UNIVERSITÀ DI PISA
Facoltà di Scienze Matematiche, Fisiche e Naturali

Laurea Specialistica in Tecnologie Informatiche

A Study on the Simulation of the Humanoid Robot

Candidate
Federico Mazzanti

Supervisor
Dr. Antonio Cisternino

Academic year 2006 / 2007

Acknowledgements

So, at last we are here ... what can I say in these occasions? We have finally reached the end of a cycle and we are going to open a new one that will definitely be more difficult than this. But cut the cackle! Now I can thank same people who have been important for the achievement of this goal.

I would like to thank Dr. Antonio Cisternino for the essential help, the precious co-operation in the project development and for having aided my professional formation.

Besides I would like to thank Diego Colombo, Crisitan Dittamo, Donatella Ermini, Daniele Mazzei, Marco Minerva, my laboratory friends and my exam mates Nicola, Grazia, Giudo, Giulio, Giorgio, Andrea for all those funny moments spent in these months and for the nice chats during the break,

Elena e Pietro for the precious contribution given during this wok drawing up,

Dr. Cianetti and Dr. Fantò for their help given during the most difficult moment of my life,

To all friends of Savino's bar and to the barman-s *who have filled with joy and happiness the after dinner moments, to the karaoke that week-end and after week-end has played us deferring music.*

In the end:

I'd like to thank Sara, my life's mate, who has helped in a strong for the achievement of this goal to the patience she has had in this last year, to the powerful contribution she has given me in the hardest moment of my life,

I'd like to thank my family for the comprehension, the support, the love displayed once more in this particular moment of my life.

In the end, I'd like to thank myself, Federico Mazzanti, for having been able to reach this important goal of life

And why not ... all the HOOTERS who give me the strength to fight

Federico Mazzanti – 12 October 2007

Ringraziamenti

Bè siamo finalmente qua, cosa dire in questi momenti? Siamo giunti finalmente alla chiusura di un ciclo e all'apertura di un altro che sicuramente risulterà essere più arduo rispetto all'attuale. Bando alle ciance, passo ora a ringraziare alcune persone alle quali devo molto per il raggiungimento di questo traguardo.

Desidero ringraziare il Dr. Antonio Cisternino per il fondamentale aiuto, la preziosa collaborazione nello svolgimento del progetto e per aver contribuito alla mia formazione professionale.

Desidero ringraziare, inoltre, Diego Colombo, Cristian Dittamo, Donatella Ermini, Daniele Mazzei e il Marco Minerva, gli amici di laboratorio e compagni di esame Nicola, Grazia, Guido, Giulio, Giorgio, Andrea per tutti quei momenti divertenti passati in questi mesi e per le allegre chiaccherate nelle pause,

Elena e Pietro per il notevole contributo offertomi durante la stesura di questo lavoro,

Dr. Cianetti e il Dr. Fantò per il loro aiuto datomi nel momento più difficile della vita,

a tutti gli amici del Savino's bar e ai baristi che hanno colmato di allegria e di gioia i dopo cena, a quel karaoke che fine settimana dopo fine settimana ha sfracassato timpano dopo timpano a tutti noi.

Infine:

Desidero ringraziare Sara, la mia compagna di vita, che ha contribuito in maniera notevole al raggiungimento di questo traguardo, alla pazienza dimostratami in quest'ultimo anno, al FORTE contributo datomi nel momento più difficile della mia vita.

Desidero ringraziare la mia famiglia per la comprensione, il sostegno e l'affetto dimostratomi ancora una volta in questo momento particolare della mia vita.

Infine desidero ringraziare Federico Mazzanti per esser riuscito a raggiungere questo traguardo importante della vita,

e perchè no.....anche a tutti i GUFI che mi hanno dato la forza di combattere.

Federico Mazzanti – 12 Ottobre 2007

Table of Contanis

Foreword on the Robotics

The idea of the robot coined to oldest times, when some myths had been told about mechanical creatures capacities to life. Automatons’ simulation was realized in the carillon of various medieval churches while in XVIII century some became famous in order to have produced ingenious and complex self-moving mockups. Today the term automaton is applied, in the common language, to the handcraft devices, usually mechanic that electronic realizes in order to imitate the living movements of the human being. The term robot derives from the Czech dictionary “*robota*”, that means "forced job". In the past it was used in the 1921 R.U.R. work (abbreviation of Rossum's Universal Robot), by the writer Czech Karel Capek, in order to designate a mechanical device that seemed human but, lacking the human sensibility, could only carry out automatic and mechanical operations. The robots, as they are understood today in technical field, are not imitations of human beings or other living shapes, except for the aspect limited to reproduce mobility. The roots of their development are in search for the effort of automation, as a whole or in parts, the operations demanded from the industrial production, above all as far as operating in positions or dangerous conditions for the human health go from particular zones of the production systems, to the oceanic depths, the land extra space. True robots, anyway, have not been realizable until the invention, in the 40s, of the computers and until the progressive miniaturization of their members. One of the first true robot experienced was the so called "*shakey*", planned by the investigators of the Stanford Research Institute in the last 60s.

It consisted in some moving blocks activated by the use of one video camera like visual sensor. Pioneer was the "*Unimate robot*", inserted in the production of the welding spot in the 60s. In the half 60s, the General Motors financed a program of development in which "*Victor Scheinman*", investigator of the MIT, perfected a motorized mechanical arm of his invention, in order to produce what was called "manipulating programmable universal for assembly". This device crowned a period of remarkable development of necessary theoretical bases for modeling robots: cinematic, dynamics, control and perception. However, in order to see the application of the robots on immense scale, it is necessary to wait for the 80s, when their cost began to be competitive regarding the traditional solutions. Later on, with the introduction of the concept of total quality, for a product, having been realized from robot became a factor of merit; in fact they co-occur to increase productivity, to reduce errors and the production costs.

These further developments have carried to the birth of the parallel robots, the independent robotics and the mobile robotics. In particular, applying systems to feedback, the robots can modify in varied degrees their operation in answer to such atmosphere variations. The trade use of the robots has gone diffusing with growth of the automation in the places of production; for a long time they have become essential in many procedures of laboratory. Japan is in front line between the nations that are exploring the possibilities of the robots. It is not possible to foretell if and when the androids of the science will become truth; in fact, also the imitation of an apparently simple action as walking has been demonstrated extremely difficult for the humanoid robots; between those we cite Honda Asimo Robot. Soon we will see one robot definition.

The robotics is not only the study of the techniques by which the robotics have been created: rather new science, related with the computer is a science, whose goal is the study of integration between movement and perception and, in last analysis, of the solution of problems in the real world, considering therefore all dynamics, also complex and unforeseeable, of the real situations.

A key passage in saying how much over is the passage from the robot understandings like manipulating to the robot like will be described later on. An industrial robot is considered like a programmable multifunctional manipulating item with automatic commands. A mobile robot is a system able to move in more or less structured atmospheres through several motor apparatuses (wheels or legs). Usually, these robots are equipped with sensors allowing them to explore the atmosphere. Navigation science concerns to direct a mobile robot through a sure atmosphere. Its goal of navigation is to catch up the destination without the robot gets lost, or crashes against some obstacle. These objects can be caught up in three tasks: mapping, planning, driving:

- Mapping concerns the ability to represent the atmosphere through a map dictates or that the robot is created exploring the atmosphere
- Planning concerns the geometric activity geometric to construct a map with opportune characteristics (of avoiding obstacles, optimal distance, etc).
- Driving concerns the ability to guide the long robot through the established distances.

Usually a robot is constructed on a fixed base or on a mobile base. For the manipulating robot, on the base there is the arm of the robot, constituted by having been fixed with same joints and linked to its extremity where the hand of manipulating is placed. Usually close to the hand same joints are placed in order to allow the hand to catch up every guideline; this group of joints is said wrist. Consequently the movements of the mechanics have the movement of the robot regarding a relative system of coordinates to the joint that has moved. It is possible to choose an absolute system of coordinates to which express the movements of the robot operating opportune conversions. It would be moreover favorable to consider the system robot as reference that is to operate with the same human modalities that the human being uses.

Nobody of us, in fact, needs to express its own movements in absolute coordinates for being able to carry out them and we do not need systems of reference for the objects that we must manipulate. This is possible because we easily know to report to objects and actions regarding same we in the space. We see in our the main types of joints and links that have been met in the robots. The joints that compose a robot can be of spin or translation. The angle between them and the spin and the link can be whichever.

The robotics is the discipline that proposed of integrating, in an intelligent way, perception and action. The action consists in the ability to move itself in the world and to move objects. It is this characteristic that distinguishes the robot from the calculating. The perception consists in interpreting given sensory. This ability is necessary for being able to have an independent behavior of the robot. [8]

The artificial intelligence studies the ways to simulate the human behavior by means of calculating, can supply many methodological instruments, to explain the reason why the robot works in a physical world, whose existence is quite uncertain and unprecise.

Introduction

The robotics domain is very important for the actual world. This argument treated in the foreward part on this thesis where we have treated the generic robotics argument. Now we focus the attention on this work, individualizing the thesis’ structure.

We start talking about the Microsoft Company Group has created, about one year ago, a robotics framework to develop robotics application. The framework’s name is *“Microsoft Robotics Studio”* and when this work has been written the framework was at version “*1.5”*.

We can develop with some programming languages, same examples that are C#, Visual Basic and Phyton. We have used a C# programming language for developing this project.

The dissertation goal is that of modeling and inserting a system for the running of a humanoid robot within the Microsoft robotics studio simulation. Robovie-M is the humanoid robot used. With this modeling and insertion we want to find out an alternative way by which developing robotics applications would be possible without the necessity to return to the real humanoid robot.

By the setting of this target, the system has to be patterned in a way to allow that the same command that has to be carried out, first on the real robot and then on the mock robot, would produce the same movement sequence. Consequently, there has to be a correspondence of 1 ~ 1 (one to one) between the real robot and the mock one.

We could also add the fact that testing one's own applications on a simulator makes the work quicker and more fluent and allows to economize on time and money…money because it's not necessary to buy a robot as the one we are going to model.

This work will deal with the modeling and the implementation of the software necessary to satisfy this sort of exigency.

Creating a humanoid robot model we need a 3D model. This model is created with Maya application. Then we have created an entity using a Microsoft robotics studio specification and inserted it on the simulation.

The thesis's structure is the following: the thesis is subdivided in two parts, one dedicated to Microsoft robotics studio framework and one dedicated to Robovie-M visual entity.

In the first chapter we will treat Microsoft robotics studio framework, and its properties. The arguments are Microsoft robotics studio runtime, Representational State Transfer (REST), Concurrency and Coordination Runtime (CCR), Decentralized System Service (DSS) and about applications model. We will tell more details information about REST, CCR and DSS.

In the second chapter we will treat Microsoft Visual Programming Language, it is an application development environment designed on a graphical data-flow based programming model.

In the third chapter we will treat Microsoft Simulation Environment and we will specify the simulation architecture, simulation runtime, simulation programming and we will have a part with same screenshots.

In the fourth chapter we will treat Robovie-M humanoid robot and its implementation. We will tell of real humanoid robot, visual entity humanoid robot and humanoid robot implementation and we will view same implementation's code.

In the fifth chapter we will treat services for managing a humanoid robot. We will start from general architecture and we will enter in a more detailed for all services and their architecture.

In the sixth chapter we will treat Visual Programming Language model, and we will view an application use services defined in the fifth chapter. In this chapter we will show the way to create a modular application.

In the seventh chapter we will show a DEMO. The DEMO will run a string command on the real humanoid robot and simulation humanoid robot.

PART I

MICROSOFT ROBOTICS STUDIO FRAMEWORK

Chapter 1

Microsoft Robotics Studio framework

1.1. Introduction

Microsoft® Robotics Studio is a Windows-based environment for hobbyist, academic and commercial developers to create robotics applications for a variety of hardware platforms. The Microsoft Robotics Studio includes a lightweight REST-style, service-oriented runtime, a set of visual authoring and simulation tools, as well as tutorials and sample code to help you get started. The Visual Programming Language enables anyone to create and debug robotics programs very easily. Just drag and drop blocks that represent services, and connect them up. You can even take a collection of connected blocks and re-use them as a single block elsewhere in your program. Simulate your robotics applications using realistic 3D simulated models.

The Microsoft Robotics Studio simulation tool includes AGEIA™ PhysX™ Technology from AGEIA Technologies Inc., a pioneer in hardware-accelerated physics, enabling real-world physics simulation for robot models. PhysX simulations can also be accelerated using AGEIA hardware. Create applications that enable you monitor or control a robot remotely using a Web-browser and send it commands using existing Web technologies, such as HTML forms and JavaScript; mount cameras on the robots and control them to survey a remote location. Microsoft Robotics Studio includes a .NET-based REST-style, services-oriented runtime. The Decentralized Software Services (DSS) application model makes it simple to access, and to respond to a robot's state, using a Web-browser or Windows-based application. The Microsoft Robotics Studio programming model can be applied to a variety of robot hardware platforms, enabling users to transfer their skills across platforms.

The programming interfaces can be used to develop applications for robots using 8, 16 or 32-bit processors, either single or multi-core. Third parties can extend the functionality of Microsoft Robotics Studio by providing additional libraries and services. Hardware or software vendors can make their products easily compatible with Microsoft Robotics Studio. With Microsoft Robotics Studio, robotics applications can be developed using a selection of programming languages, including those in Microsoft Visual Studio® and Microsoft Visual Studio Express (C# and VB.NET), as well as scripting languages such as Microsoft Iron Python. Third-party languages that support the Microsoft Robotics Studio services-based architecture are also supported.

Microsoft Robotics Studio Runtime

The runtime supports a wide variety of hardware ranging from robots connected directly to a PC (using the serial port, Bluetooth, USB, etc.) to robots that has an on-board PC, to simulated robots that can be manipulated as they operate in a simulated world. In addition, the runtime has been designed to support any type of robotics application ranging from basic observation of sensory input, to drive-by-wire and remote presence, to autonomous operation, to cooperation between multiple autonomous robots, and beyond. The Microsoft Robotics Studio Runtime environment has been designed to accommodate the development of applications with a broad set of requirements including:

1. It must be possible to monitor state and interact with individual components while the application is running.
2. It must be possible to discover, create, terminate, and restart components while the application is running.
3. It must be possible to deal with inputs from multiple sensors concurrently and to orchestrate such inputs as tasks without risk of unintended interference between the tasks.
4. It must be possible to handle autonomous as well as controlled robotics applications both locally and across the network.
5. The runtime must be lightweight enough to be executed in a wide variety of environments.

6. The application environment must be extensible and flexible enough to accommodate interaction with a wide variety of hardware and software environments.

Microsoft Robotics Studio Runtime supports a variety of Windows platforms supported by .NET framework as well as Windows CE using .NET Compact Framework. The runtime consists of two main components that facilitate building, monitoring, deploying, and running a broad set of applications with such requirements.

Concurrency and Coordination Runtime (CCR) provides a highly concurrent, message oriented programming model featuring orchestration primitives enabling coordination of messages without the use of manual threading, locks, semaphores, etc. CCR addresses the need of service-oriented applications by providing a programming model that facilitates managing asynchronous operations, dealing with concurrency, exploiting parallel hardware and handling partial failure. It enables you to design your application so that its software modules or components can be loosely coupled; that is, so they can be developed independently and make minimal assumptions about their runtime environment and other components. This approach changes the way of thinking how you think of programs from the beginning of the design process and facilitates dealing with concurrency, failure and isolation in a consistent way. The CCR runtime is a self-contained .NET DLL accessible from any language targeting the .NET 2.0. Common Language Runtime (CLR).[2]

Decentralized System Services (DSS) provides a lightweight, service oriented application model that combines key aspects of traditional Web-based architecture (commonly known as REST) with pieces of Web Services architecture. The application model defined by DSS builds on the R.E.S.T. model by exposing services through their state and a uniform set of operations over that state but extends the application model provided by HTTP by adding structured data manipulation, event notification, and service composition. The primary goal of DSS is to promote simplicity, interoperability, and loose coupling. This makes it particularly suited for creating applications as compositions of services regardless of whether these services are running within the same node or across the network. The result is a highly flexible yet simple platform for writing a broad set of applications. DSS uses HTTP and DSSP as the foundation for interacting with services. DSSP is a lightweight SOAP-based protocol that provides support for manipulation of structured state and for an event model driven by changes to the structured state. DSSP is used for manipulating and subscribing to services and hence complements HTTP in providing a simple, state-driven application model. The DSS runtime is built on top of CCR and does not rely on any other components in Microsoft Robotics Studio. It provides a hosting environment for managing services and a set of infrastructure services that can be used for service creation, discovery, logging, debugging, monitoring, and security.[2]

Application Model

The primary task of robotics applications is to consume sensory input from a variety of sources and orchestrate a set of actuators to respond to the sensory input in a manner that achieves the purpose of the application. As an example, the robotics application data-flow diagram shown below contains a simple bumper (sensor) that reports when it is hit, a message box (actuator) that controls the display, and an orchestrator that connects the pieces together.

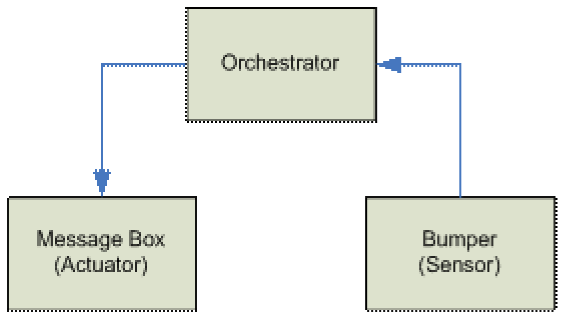


FIGURE 1

This orchestration is simple but as applications grow, there may be any number of sensors, actuators, and orchestrators communicating with each other to perform complex operations. An example of a more advanced robotics application data-flow is illustrated below:

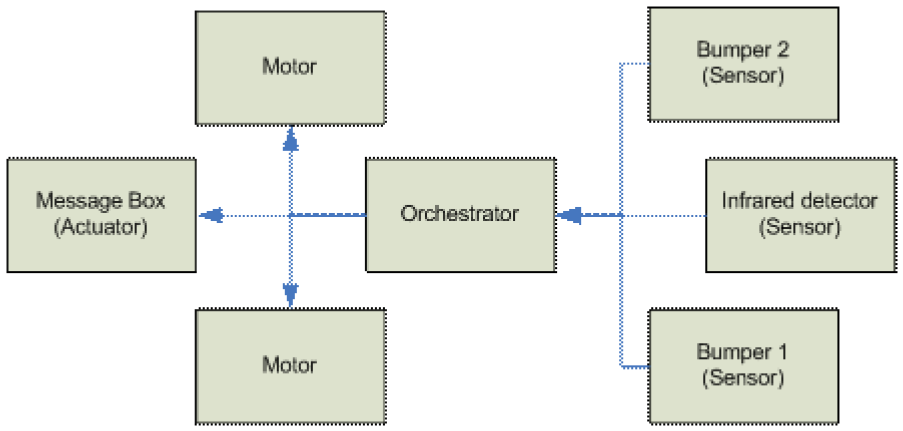


FIGURE 2:

While each of the sensors and actuators are similar to the first example, the orchestrator now has to manage more components. In addition to the added complexity of orchestrating a growing number of components (that each may also be orchestrations), there are several things that distinguish the application data-flows above from many other "Hello world" applications:

1. Handling of sensory input and controlling actuators must be dealt with concurrently as otherwise actuators can get starved and sensors ignored.
2. Orchestration and composition is a critical part of the application, especially as the number of sensors and actuators grow and the orchestration becomes more complex.
3. Autonomous and collaborative orchestration requires that components can be distributed and made accessed over the network.

In order to meet these requirements, Microsoft Robotics Studio Runtime provides a highly concurrent, service oriented application model which combines key aspects of traditional Web-based architecture with pieces from Web Services to provide a flexible yet lightweight distributed application model. Naturally, this way of viewing an application is not unique to robotics and sensors and actuators are not limited to traditional robotics components like motors and bumpers.

The primary focus of Web architecture is on simplicity, interoperability, and loose coupling. HTTP promotes this model by allowing applications to take advantage of a simple, state-oriented application model commonly known as "REST". Web based applications have through HTTP demonstrated that this model scales well, provide interoperability, and is flexible enough to accommodate a wide variety of scenarios. Despite the success of HTTP, however, there are well-known aspects that it doesn't capture. In particular, the three limitations that Web applications run up against repeatedly are:

1. No support for manipulation of structured data. That is, the only operations allowed on a resource are at the "resource level"; not at the substructure of a resource.
2. No support for event notification. HTTP is a request/response protocol and does not support an event notification model based on pushing data to subscribers.
3. No support for expressing the kind of relationships between components illustrated in the diagrams above.

The Microsoft Robotics Studio Runtime provides a REST-based application model but augments it with pieces from Web services for handling structured data manipulation, event notification, and partner management between services. By extending the REST model in this manner, applications can take advantage of an event notification model and structured manipulation of data without loosing interoperability with existing REST infrastructure. The result is much more interactive and dynamic applications constructed as compositions of services organized both within nodes and across the network. Microsoft Robotics Studio Runtime adopts the notion that a resource can be a structured entity and defines a set of operations that enables service state to be added, deleted, updated, and queried without requiring a common data model across all services.

Manipulating state in this manner is not new but by combining the structured view of Web services with REST the result is a much more flexible way for services to interact.

The second piece that Microsoft Robotics Studio Runtime adopts from the Web Services world is that of event notifications. By modeling an event as a state change in a service, it becomes straightforward to introduce events into the REST model. For example, in the case of an UPDATE operation on a service, the state of that service changes as a direct result of that UPDATE operation. Furthermore, the UPDATE operation itself directly represents the state change and so it is natural to think of the event notification as simply being the UPDATE operation.

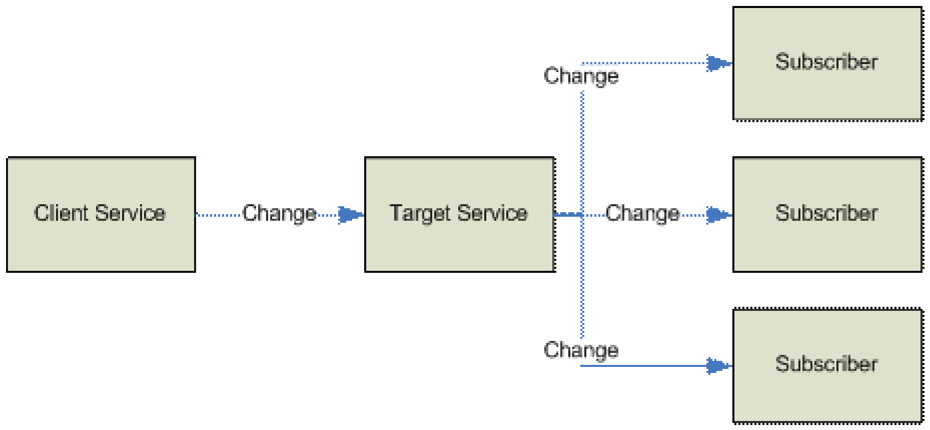


FIGURE 3

By expressing event notifications in terms of state changes in a service, subscribers have a common way of monitoring publishers independent of the semantics of a particular service and publishers have a common way of representing event notifications by simply relaying all state changing operations to subscribers.

Microsoft Robotics Studio Service Model

In the Microsoft Robotics Studio Runtime, an application is a composition of services where each service is a REST style service with added support for event notification and structured data manipulation as described above.

By orchestrating loosely coupled and lightweight services on a single host or across the network, applications of varying complexity can be created and composed with each other to create always more complex applications. Because of the relationship to HTTP, the state of each service can be monitored and manipulated directly through HTTP using a Web browser, or through a simple SOAP-based protocol called DSSP (Decentralized Software Services Protocol):

- HTTP enables support for operations such as GET, PUT, and POST, and DELETE. This enables a service to be used within a rich UI context such as a Web browser using existing infrastructure and existing data formats.
- DSSP enables support for structured data manipulation of the service state supporting semantics like UPDATE, INSERT, and QUERY. In addition, DSSP provides an event notification model that is tied to changes in the state of that service.

An example of simple Microsoft Robotics Studio application is a composition of three services that can be used to control a basic robot. The services are: A drive service (actuator) that can move the robot in different directions at varying speeds; a bumper server (sensor) that signals when another object is hit, and an explorer service (orchestration) that changes the speed and direction of the robot depending on input from the bumper. The Explorer service listens for event notifications from the bumper and determines the speed and direction of the drive service.

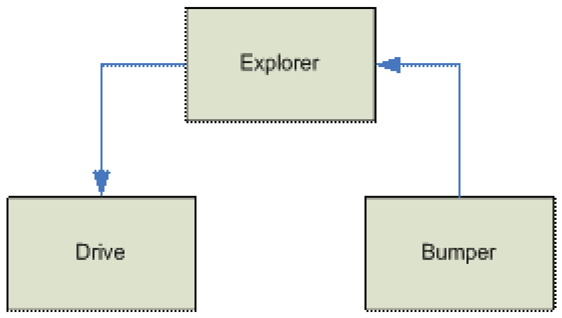


FIGURE 4

In this exaample application the Explorer service subscribes to the Bumper service and sends instructions to the Drive service. Such composition is called partner. A partnership represents a relationship that a service has to another service. A partner is described by the type of the partner and the partner service instance. Partner information is also exposed as state and can be inspected using the LOOKUP operation which is part of DSSP.

1.2. Representational State Transfer (REST)

In the previous section the concept of REST has been introduced, that is a structure characterize entire framework for control and inter-operability. Web architecture services that REST model used will be introduced. REST is the acronym for *REpresentational State Transfer*.

The first idea of REST was worked out to point 1994 in the shape of architecture series deriving from study of the greater part of constituent web elements and their interaction.

By its specifying architecture model REST is not a standard or a single protocol, therefore we will not never see a RFC or a Recommendation W3C. It just represents a guide, a reference card, to realize a web systems formulated. There is a specific architecture REST model, that varies in bringing back as much as it can from soma source and doesn't describe the null care realization of web services. The first fundamental aspect of the architecture REST model regards the functions separating the several quantity of the inside elements. This concept is very important because concerns planning principles and can be seen like the decomposition of a problem into simpler small problems.

This important it concerns the maintenance of the communication's state. According to REST the communication must rigorously be stateless, that is without state. This only limits the server side, while client can maintain the communication state. Once chosen, this type can be realized making message containing all the necessary information to the whole understanding. That would carry to focus the attention on the type of messages based on XML. In architecture model REST, a mechanism of caching is also previewed, that is previewing the possibility to distinguish answers part of the server like cacheable or not cacheable information.

The cacheable information in a generalized manner stretches to remain static in the course of the time. An example can be people's nationality information. Another important model REST's focus regards detail lists on the interfaces. REST separates objects heading from implementations by means it uses one series of generic interfaces.

The REST Architecture emphasizes the importance to subdivide the system in to levels of organized structure. Every level is made up of elements each elements has a position to seeing the single level with which it will have to interact. The way in which level uses the services is discovered and its turn supplies services. This tie serves in order to impose an advanced complexity limit of general system and to favour independence of several levels.

The last tie introduced by REST regards the extendable. It implies that a system must be able to extend the functionalities by means.

In this way is possible to guarantee realization of the system in short times the members being equipped of minimal equipment software. Subsequently several members could be part of the extended necessity system. [10]

1.3. Concurrency and Coordination Runtime (CCR)

The Concurrency and Coordination Runtime (CCR) is a managed code library (DLL) accessible from any language targeting the .NET 2.0. Common Language Runtime. The CCR addresses the need of service-oriented applications to manage asynchronous operations, deal with concurrency, exploit parallel hardware and deal with partial failure. It enables you to design your application so that its software modules or components can be loosely coupled; that is, they can be developed independently and make minimal assumptions about their runtime environment and other components. This approach changes the way of thinking of programs from the start of the design process and deals with concurrency, failure and isolation in a consistent way.

Problem Areas

- **Asynchronous Programming** - When communicating between loosely coupled software components, like programs running across the network, or UI code communicating with the user input and the file to subsystem, asynchronous operations enable the code to scale better, be more responsive, and deal with failure across multiple operations. Asynchronous programming however considerably reduces the readability of user code, since logic is often split up between "callbacks" and the code that originates the operation. Also, handling failure across multiple outstanding operations is an almost impossible task to get right.
- **Concurrency** - Code that needs to better utilize multiple execution resources, must be split up in independent logical segments, that can run in parallel, and communicate when necessary to produce results from the combined execution. Often, that logical segment is captured by the Thread OS primitive, that is nothing more than a long lived iteration. Because of Thread performance implications on thread startup, the thread stays active for long periods of time, forcing a particular pattern: Code is structured as long sequences that use blocking or synchronous calls, and only deal with one thing at a time. Furthermore, threads assume that the primary communication between them is shared memory, forcing the programmer to use very explicit, error-prone methods to synchronize access to that shared memory.
- **Coordination and Failure Handling** - Coordinating between components is where most of the complexity in large software programs lies. A mismatch of interaction patterns leads to unreadable code, where the runtime behavior changes drastically between coordination approaches. More importantly, the error handling approaches are ill-defined and again vary drastically.

Application Model

The CCR is appropriate for an application model that separates components into pieces that can interact only through messages. Components in this model need means to coordinate any messages, to deal with complex failure scenarios, and effectively cope with asynchronous programming. This application model is also very similar to the way in which heterogeneous hardware is integrated and network applications are built. Most software programs have the same needs, from traditional client PC programs to server applications, to applets running in the browser. The software needs to coordinate user input, storage input/output and UI presentation. Although disguised in layers of mostly synchronous APIs, asynchrony is inevitable since the devices operate at different speeds, have large differences in available resources, and in general we know how to use queues to isolate them. The following sections introduce the CCR programming model and its implementation that addresses the above areas in an efficient, robust and extensible way. The CCR implementation has three main categories of functionality:

1. The **PORT** and **PORTSET** queueing primitives. The **PORT** class is a FIFO queue of items, and in most cases also a queue of receivers.
2. The coordination primitives also called **ARBITERS**. These are classes that execute user code, often a delegate to some method, when certain conditions are met. The primitives can be nested and can also be extended by the programmer.
3. The **DISPATCHER**, **DISPATCHERQUEUE** and **TASK**. The CCR isolates the scheduling and load balancing logic from the rest of the implementation by using these classes to abstract the way in which user tasks execute, what they contain, and on what resources they will run. The CCR avoids a single, process wide execution resource, like the CLR thread pool and instead allows the programmer to have multiple, completely isolated pools of OS threads that abstract any notion of threading behind them. In the most common case, hundreds of software components can share just a single Dispatcher resource which can load balance millions of tasks across some small number of OS threads. The **DISPATCHERQUEUE** class is the only way to interact with the **DISPATCHER** class, and multiple queues per dispatcher allow for a fair scheduling policy.

CCR Port and PortSet

The CCR port is the most common primitive and it is used as the point of interaction between the two components. The port is implemented as the combination of two interfaces: ***IPortReceive*** and ***IPort***. The interfaces logically separate methods that add items to the port, and methods that retrieve items, or attach code that removes items and executes asynchronously. The interfaces are not strongly typed but the default implementation, the *Port*<> class, has one generic type argument.

Note: The notation <> next to a class indicates its a generic class (similar to templates in C++). For example *Port<int>* is a port implementation with one CLR type argument of *System.Int32*.

Posting Items

Adding an item to the port is an asynchronous operation. When there are no arbiters attached to the port, the item is added to a queue. If there are arbiters present, each one will be called to evaluate the item and decide if a Task should be created and scheduled for execution. This evaluation is a fast, non-blocking operation so the *Post* method returns control to the caller as soon as possible. This is the source of the asynchronous behavior in the CCR: Programs post items quickly to ports, which can then cause Tasks to get scheduled in some thread other than the current one, if certain conditions are met.

```
// Create port that accepts instances of System.Int32
Port<int> portInt = new Port<int>();

// Add the number 10 to the port
portInt.Post(10);

// Display number of items to the console
Console.WriteLine(portInt.ItemCount);
```

CODE 1

In the example above, we create a port typed to accept only integers. We then add one item (the number 10) to the port and then check the port ItemCount property which should read 1.

Retrieving Items

There are two scenarios when retrieving items from a port:

1. The port is used as a "passive" queue: no tasks are scheduled when items are posted. In this scenario items can be retrieved by calling the *Test* method. The method will never block. If no item is present it will return false and set the out parameter to null. This scenario is useful when you have some other mechanism to execute code due to some event, and you simply want to use the ports as efficient FIFO queues. It provides the flexibility of later attaching a receiver to the port, that can asynchronously execute tasks for any item already present
2. The port is used as an "active" queue: tasks are scheduled for execution when items are posted, due to one or more arbiter being registered with the port. This is the most common case of using CCR ports. It is the primary source of concurrency since any user code that gets scheduled due to an item being posted, can potentially execute in parallel for every item posted.

```
// Create port that accepts instances of System.Int32
Port<int> portInt = new Port<int>();

// Add the number 10 to the port
portInt.Post(10);

// Display number of items to the console
```

```
Console.WriteLine(portInt.ItemCount);

// retrieve the item using Test
int item;
bool hasItem = portInt.Test(out item);
if (hasItem)
{
    Console.WriteLine("Found item in port:" + item);
}
```

Code 2

In the example above (an extension of Code 1), we use the *Test* method to retrieve the item posted. The item is removed in a thread-safe manner and **ItemCount** should read zero.

```
// Create port that accepts instances of System.Int32
Port<int> portInt = new Port<int>();

// Add the number 10 to the port
portInt.Post(10);

// Display number of items to the console
Console.WriteLine(portInt.ItemCount);

// create dispatcher and dispatcher queue for scheduling tasks
Dispatcher dispatcher = new Dispatcher(0, "sample dispatcher");
DispatcherQueue taskQueue = new DispatcherQueue("sample queue", dispatcher);

// retrieve the item by attaching a one time receiver
Arbiter.Activate(taskQueue,
    Arbiter.Receive(false, // one time
        portInt, // port to register on
        delegate(int item) // user delegate
        {
            // this code executes in parallel with the method that
            // activated it
            Console.WriteLine("Received item:" + item);
         }
    ));

// any code below runs in parallel with delegate
```

Code 3

In the example above we follow the same steps as Code 1, but instead of using the *Test* method, we use the ***Arbiter.Activate*** method, to register a simple receiver arbiter to the port. The receiver is associated with a user delegate, in this case an anonymous method defined inline.

The delegate will execute in one of the dispatcher threads associated with the *DispatcherQueue* instance supplied. Note that the delegate always runs in parallel with the main body of this example. In this example, after the delegate runs, the receiver is automatically removed from the port. More on receivers and arbiters in a later section.

Inspecting Port State

The following methods and properties on the *Port<>* class are useful for inspecting its state at runtime, either for debugging or for regular code logic

- *ToString* - This method overrides the default ToString() implementation and will produce, in human readable form, the number of items in the port and the hierarchy of the receivers associated with the port.
- ItemCount - Number of items currently queued. Note that anytime one or more persisted Arbiters (they stay attached even after consuming one item) is registered with the port, this count will be zero. Most Arbiters cause items to be immediately converted to Tasks, scheduled for execution, so the items never actually show up in the port item queue.

Port Sets

Since the *Port<>* class only takes one generic type argument, it is often convenient to associate multiple, independent instances of the *Port<>* class, under a single container itself that can be treated as one entity. The *PortSet<>* generic class allows this grouping and is the most common way to define the "interface" to a CCR software component: Instead of having a set of public methods, CCR components expose only a strongly typed *PortSet*, with N independent *Port<>* instances underneath. The components can now coordinate the way in which different message types execute in relation to each other. Furthermore the code always runs concurrently and independently from the code that posts the messages. This programming model is very familiar to authors of web services, traditional server processes, or kernel mode I/O processors.

Constructing a PortSet instance

There are two ways to create a new instance of *PortSet*:

1. Use the generic type arguments to define at compile time, the number and type of *Port<>* instances the *PortSet* will support. The CCR provides implementations of the *PortSet* of up to twenty generic type arguments on the desktop CLR, and up to eight generic arguments on the .NET Compact Framework (due to a JIT limitation). Using this approach gives you the best type safety and best runtime performance when posting. However it limits the number of types the port set can support.
2. Use the initialization constructor that takes a parameter list of type arguments. You can supply an arbitrary number of types, and the *Port<>* instances will be created at runtime. Some methods, like the *Post* for each message type is not available (*PostUnknownType* or *TryPostUnknownType* must be used) and there is a minor performance hit since the item type has to be checked and compared at runtime, against a table of valid types.

```
// Create a PortSet using generic type arguments
PortSet<int, string, double> genericPortSet = new PortSet<int, string, double>();

genericPortSet.Post(10);
genericPortSet.Post("hello");
genericPortSet.Post(3.14159);

// Create a runtime PortSet, using the initialization
// constructor to supply an array of types
PortSet runtimePortSet = new PortSet(
   typeof(int),
   typeof(string),
   typeof(double)
);

runtimePortSet.PostUnknownType(10);
runtimePortSet.PostUnknownType("hello");
runtimePortSet.PostUnknownType(3.14159);
```

CODE 4:

In the example above we show how to create a *PortSet* with three different types, first using the generic type arguments, and then using the type array at runtime. The strongly typed *Post* methods can be added to a class that derives from the non generic *PortSet*, providing the same compile time safety as the generic *PortSet*<>.

```
/// <summary>
/// PortSet that accepts items of int, string, double
/// </summary>
public class CcrConsolePort : PortSet<int, string, double>
{
}

/// <summary>
/// Simple example of a CCR component that uses a PortSet to abstract
/// its API for message passing
/// </summary>
public class CcrConsoleService
{
   CcrConsolePort _mainPort;
   DispatcherQueue _taskQueue;
   /// <summary>
   /// Creates an instance of the service class, returning only a PortSet
   /// instance for communication with the service
   /// </summary>
   /// <param name="taskQueue"></param>
   /// <returns></returns>
   public static CcrConsolePort Create(DispatcherQueue taskQueue)
   {
      CcrConsoleService console = new CcrConsoleService(taskQueue);
      console.Initialize();
      return console._mainPort;
   }

/// <summary>
/// Initialization constructor
/// </summary>
/// <param name="taskQueue">DispatcherQueue instance used for scheduling</param>
private CcrConsoleService(DispatcherQueue taskQueue)
{
   // create PortSet instance used by external callers to post items
   _mainPort = new CcrConsolePort();
   // cache dispatcher queue used to schedule tasks
   _taskQueue = taskQueue;
}

private void Initialize()
{
   // Activate three persisted receivers (single item arbiters)
   // that will run concurrently to each other,
   // one for each item/message type
   Arbiter.Activate(_taskQueue,
     Arbiter.Receive<int>(true, _mainPort, IntWriteLineHandler),
     Arbiter.Receive<string>(true, _mainPort, StringWriteLineHandler),
     Arbiter.Receive<double>(true, _mainPort, DoubleWriteLineHandler)
   );
}

void IntWriteLineHandler(int item)
{
   Console.WriteLine("Received integer:" + item);
}

void StringWriteLineHandler(string item)
{
   Console.WriteLine("Received string:" + item);
}

void DoubleWriteLineHandler(double item)
{
   Console.WriteLine("Received double:" + item);
}
```

CODE 5:

In the example above we define a simple class, *CcrConsolePort*, that derives from a *PortSet* with three type arguments. This make any subsequent use of the port set more readable since we don't have to repeat the generic type definitions. The class *CcrConsoleService*, implements a common pattern for CCR components. It has a static routine that creates the instance object and returns the private *PortSet* instance for communicating with it. It then activates a handler for each message type in the *PortSet*. For each handler activation is concurrent.

Type Safety

The *PortSet* class allows for enumeration of all the port instances and item types it supports. The generic *PortSet* also provides convenient methods that automatically invoke the correct *Port<>* instance plus implicit conversion operations that can cast a *PortSet<>* instance to a *Port<>* instance, automatically, if the type argument is valid for that *PortSet*.

This is useful when registering receivers with one of the ports in the *PortSet*, using just the type argument.

```
// instance of portset
PortSet<int,string,double> portSet = new PortSet<int,string,double>();

// the following statement compiles because of the implicit assignment operators
// that "extract" the instance of Port<int> from the PortSet
Port<int> portInt = portSet;

// the implicit assignment operator is used below to "extract" the Port<int>
// instance so the int receiver can be registered
Arbiter.Activate(_taskQueue,
   Arbiter.Receive<int>(true, portSet, delegate(int item) { })
);
```

CODE 6:

In the example above we demonstrate the use of the implicit operator in two use cases:

- Assigning the correct instance of a *Port*<> within a *PortSet*, to another *Port*<> variable
- Extracting the correct instance of a *Port*<> so it can be used to register an Arbiter

CCR Coordination Primitives

Asynchronous programming is hard because there is no simple way to coordinate between multiple operations, deal with partial failure (one of many operations fail, others succeed) and also define execution behavior of asynchronous callbacks, so they don't violate some concurrency constrain (for example they don't attempt to do something in parallel). The CCR enables and promotes concurrency by providing ways to express what coordination should happen, and which are the high level constraints between different code segments, all run due to some messages being received on ports. It’s important to realize the relationship between asynchronous behavior and concurrency: The loose coupling, fast initiation of work, and consistent use of queues for interaction, promotes software design that scales and has well defined dependencies. So if the drawbacks mentioned above can be addressed, it is an appropriate model for software components. The coordination primitives provided by the CCR can be classified based on two primary usage scenarios:

1. Coordination of inbound requests, for long lived service-oriented components. A common example is a web service listening for HTTP requests on some network port, using a CCR port to post all inbound requests and attaching handlers that wake up and server each request independent of each other. In addition, it uses some of the advanced primitives to guarantee that some handlers never run when others handlers are active
2. Coordination of responses from one or more outstanding requests, with multiple possible return types per response. One example is waiting for success or failure,

on a PortSet associated with a pending request: When the request completes a success item or a failure item will be posted, and the appropriate code should execute. Another example is scattering multiple requests at once, then collecting all the responses using a single primitive, not caring which order the responses arrive across a single or multiple response ports

Each arbiter will be described briefly, then more detailed explanation in the context of the above scenarios will be provided.

Arbiter Static Class

The Arbiter static class provides helper routines for creating instances all CCR Arbiter classes, type safe manner. All methods described below are members of this class. The Arbiter static methods are not an exhaustive list of all the possible ways the CCR arbiters can be constructed. For more advanced configurations, each Arbiter class can be constructed directly using the appropriate constructor parameters. The following list shows which CCR classes are created when invoking some of the most common Arbiter class methods (this is not an exhaustive list)

- *Arbiter.FromTask* -> Creates instance of ***Task***
- *Arbiter.Choice* -> Creates instance of ***Choice***
- *Arbiter.Receive* -> Creates instance of ***Receiver***
- *Arbiter.Interleave* -> Creates instance of ***Interleave***
- *Arbiter.JoinedReceive* ->Creates instance of ***JoinReceiver***
- *Arbiter.MultipleItemReceive* -> Creates instance ***JoinSinglePortReceiver***

Single Item Receiver

A single item receiver associates a user delegate that takes a single parameter of type T, with an instance of *Port<T>*. If the persist option is true, the receiver will execute an instance of the user delegate, for each item posted. If the persist option is false, the receiver will only execute the user delegate for one item and then un-register from the port.

Important: If items are already queued on the *Port<T>* instance, the receiver will still execute them, providing a reliable way to "catch-up" to queued items, and make the user code agnostic on when the items where actually posted.

An example is:

```
Port<int> port = new Port<int>();

Arbiter.Activate(_taskQueue,
   Arbiter.Receive<int>(
      true,
      port,
      delegate(int item) { Console.WriteLine(item); }
   )
);

// post item, so delegate executes
port.Post(5);
```

CODE 7:

The example above shows how to create a *Port<int>* instance and then activate a single item receiver, that will execute the user delegate every time an item is posted on the port. Notice that the receiver is persisted and it will be active on the port until the port instance is garbage collected.

Choice Arbiter

The Choice Arbiter only executes one of its branches, and then atomically (in one step that can't be interrupted) removes all other nested Arbiters from their ports. This guarantees that only one choice will ever run and is a common way to express branching behaviour, deal with responses that have success/failure, or guard against races.

```
// create a simple service listening on a port
ServicePort servicePort = SimpleService.Create(_taskQueue);

// create request
GetState get = new GetState();

// post request
servicePort.Post(get);

// use the helper on the Arbiter class that creates a choice
// given two types found on one PortSet. This a common use of
// Choice to deal with responses that have success or failure
Arbiter.Activate(_taskQueue,
   Arbiter.Choice(get.ResponsePort, // PortSet with ports to listen on
   delegate(string s) { Console.WriteLine(s); }, // delegate for success
   delegate(Exception ex) { Console.WriteLine(ex); } // delegate for failure
));
```

CODE 8:

The example above shows one common use of the ***Choice*** arbiter to execute two different delegates, based on messages received on a *PortSet*. Note that the *Choice* class can take an arbitrary number of receivers, and coordinate across them, not just two. The *Arbiter.Choice* method above is concise alternative to creating a *Choice* arbiter, and then creating two *Receiver* arbiters, one for each delegate.

Important: The choice arbiter is an example of a "parent" arbiter: other arbiters, such as single item receivers or joins, can be nested under a Choice. The arbiter design allows for a hierarchy of arbiters, invoking each arbiter in the hierarchy in the correct order, before determining if a user handler should execute. This allows the programmer to express complex coordination with just a few lines of code.

Joins and Multiple Item Receivers

Multiple item receiver arbiters come into two categories:

1. Also known as ***joins*** (or WaitForMultiple in OS literature) they are receivers that attempt to receive from one or more ports and if one of the attempts fail, they post any items back and wait to try again when the right conditions are met. This two phase logic provides a type safe and deadlock free mechanism. It can be used to guarantee atomic access to multiple resources, without the fear of deadlock, since the order the items are received is not important. The number of items and ports can be specified at runtime or be fixed at compile time. The fact that the number of items in the join can be specified at runtime, is an important extension the CCR provides over other forms of typed joins.
2. Receivers that eagerly remove items from each port participating in the receive, and when the total item count is satisfied, execute the user delegate. This version is very fast but should not be used as a resource synchronization primitive. It is often used for gathering results for multiple pending requests (scatter/gather scenarios)

Joins

```
//instance of ports
Port<double> portDouble = new Port<double>();
Port<string> portString = new Port<string>();

// activate a joined receiver that will execute only when one
// item is available in each port.
Arbiter.Activate(_taskQueue,
   Arbiter.JoinedReceive<double, string>(
      false,// one time
      portDouble, // first port to listen
      portString, // second port to listen
      delegate(double value, string string Value)
      {
         value /= 2.0;
         stringValue = value.ToString();
         // post back updated values
         portDouble.Post(value);
         portString.Post(stringValue);
      })
);

// post items. The order does not matter, which is what Join its power
portDouble.Post(3.14159);
portString.Post("0.1");

//after the last post the delegate above will execute
```

CODE 9:

In the example above we demonstrate a simple "static" join (specified at compile type with fixed number of ports). We activate a join receiver across two ports, and then post items in each port. The join logic will then determine it has everything it needs and schedule the delegate for execution.

Multiple Item receivers

Multiple item receivers are appropriate when no contention is expected on the ports. They can be used to aggregate responses from multiple requests.

```
// create a simple service listening on a port
ServicePort servicePort = SimpleService.Create(_taskQueue);

// shared response port
PortSet<string, Exception> responsePort = new PortSet<string, Exception>();

// number of requests
int requestCount = 10;

// scatter phase: Send N requests as fast as possible
for (int i = 0; i < requestCount; i++)
{
   // create request
   GetState get = new GetState();

   // set response port to shared port
   get.ResponsePort = responsePort;

   // post request
   servicePort.Post(get);
}

// gather phase:
// activate a multiple item receiver that waits for a total
// of N responses, across the ports in the PortSet.
// The service could respond with K failures and M successes (K+M == N)
Arbiter.Activate(_taskQueue,
   Arbiter.MultipleItemReceive<string, Exception>(
   responsePort, // port set used to gather success or failure
   requestCount, // total responses expected
   delegate(ICollection<string> successes, ICollection<Exception> failures)
   {
      Console.WriteLine("Total received:" + successes.Count + failures.Count);
   })
);
```

CODE 10:

The example above shows a common case for dealing with multiple pending asynchronous operations, using a single delegate to gather the results. Assuming that for any N operations, K can fail, M can succeed, and K+M = N, the CCR ***MultipleItemReceiver*** gives a concise way to gather all the results, arriving in any order and in any combination across the types. A single delegate will be called, with two collections, containing the K failures and M successes. The *Arbiter.MutipleItemReceive* method can be used for two discrete types but the underlying ***MultipleItemGather*** CCR arbiter can work with an arbitrary number of types.

Persisted Single Item Receivers

CCR was motivated from the beginning as the runtime capable of efficiently executing components that listen on some queues for messages, and activate handlers to process inbound messages. The simplest case is to use the Receiver arbiter, in persisted mode, to listen on a port and activate a handler whenever an item is posted.

```
/// <summary>
/// Base type for all service messages. Defines a response PortSet used
/// by all message types.
/// </summary>
public class ServiceOperation
{
   public PortSet<string, Exception> ResponsePort = new PortSet<string, Exception>();
}

public class Stop : ServiceOperation
{
}
public class UpdateState : ServiceOperation
{
   public string State;
}

public class GetState : ServiceOperation
{
}

/// <summary>
/// PortSet that defines which messages the services listens to
/// </summary>
public class ServicePort : PortSet<Stop, UpdateState, GetState>
{
}

/// <summary>
/// Simple example of a CCR component that uses a PortSet to abstract
/// its API for message passing
/// </summary>
public class SimpleService
{
   ServicePort _mainPort;
   DispatcherQueue _taskQueue;
   string _state;

   public static ServicePort Create(DispatcherQueue taskQueue)
   {
      SimpleService service = new SimpleService(taskQueue);
      service.Initialize();
      return service._mainPort;
   }
```

```
    private void Initialize()
    {
       // using the supplied taskQueue for scheduling, activate three
       // persisted receivers, that will run concurrently to each other,
       // one for each item type
       Arbiter.Activate(_taskQueue,
           Arbiter.Receive<UpdateState>(true, _mainPort, UpdateHandler),
           Arbiter.Receive<GetState>(true, _mainPort, GetStateHandler)
       );
    }

    private SimpleService(DispatcherQueue taskQueue)
    {
       // create PortSet instance used by external callers to post items
       _mainPort = new ServicePort();

       // cache dispatcher queue used to schedule tasks
       _taskQueue = taskQueue;
    }

    void GetStateHandler(GetState get)
    {
       if (_state == null)
       {
          // To demonstrate a failure response,
          // when state is null will post an exception
          get.ResponsePort.Post(new InvalidOperationException());
          return;
       }
       // return the state as a message on the response port
        get.ResponsePort.Post(_state);
    }

    void UpdateHandler(UpdateState update)
    {
       // update state from field in the message
       _state = update.State;

       // as success result, post the state itself
       update.ResponsePort.Post(_state);
    }
}
```

CODE 11:

In the example above we show a class implementing the common CCR pattern for a software component:

- Definitions of message types used to interact with the component
- Definition of a *PortSet* derived class that accepts the message types defined. It's not necessary to derive from PortSet, but it's a convenient way to reuse a *PortSet* with a particular number of types.
- A static *Create* method that initializes an instance of the component and returns a *PortSet* instance used to communicate with the component instance

- A private *Initialize* method that attaches some Arbiters on the public *PortSet* external code will use to talk to the service

If no concurrency constraints exist between the different handlers, simple, persisted single item receivers can be used.

Interleave Arbiter

For non trivial components that listen on ports, it is often the case that a private resource used should be carefully protected from concurrent access. A data structure stored internally that required multiple updates that must be treated atomically is one case. Another scenario is a component implementing a complex multi-step process, that cannot be preempted when certain external requests arrive. The CCR helps the programmer think only about implementing the complex process, and takes care of queueing requests and handler activations, until the process is complete. The programmer uses the ***Interleave*** arbiter to essentially declare what protection segments of code require.

For programmers familiar with the **reader/writer lock** primitive in thread programming, the interleave arbiter is a similar concept (it's a writer biased reader/writer) but instead of locking a specific object, sections of code are protected from each other. Avoiding contention on a lock, interleave uses internal queues to create scheduling dependencies, and manages execution so tasks that can run concurrently, do, and tasks that run exclusively, wait for all other tasks to complete first.

```
/// <summary>
/// Simple example of a CCR component that uses a PortSet to abstract
/// its API for message passing
/// </summary>
public class ServiceWithInterleave
{
    ServicePort _mainPort;
    DispatcherQueue _taskQueue;
    string _state;

    public static ServicePort Create(DispatcherQueue taskQueue)
    {
        ServiceWithInterleave service = new ServiceWithInterleave(taskQueue);
        service.Initialize();
        return service._mainPort;
    }

    private void Initialize()
    {
        // activate an Interleave Arbiter to coordinate
        //how the handlers of the service
        // execute in relation to each other and to their own parallel activations
        Arbiter.Activate(_taskQueue,
           Arbiter.Interleave(
              new TeardownReceiverGroup(
                 Arbiter.Receive<Stop>(false, _mainPort, StopHandler)
              ),
              new ExclusiveReceiverGroup(
                  Arbiter.Receive<UpdateState>(true, _mainPort, UpdateHandler)
              ),
              //parallel execute
              new ConcurrentReceiverGroup(
                  Arbiter.Receive<GetState>(true, _mainPort, GetStateHandler)
              ))
            );
     }

     private ServiceWithInterleave(DispatcherQueue taskQueue)
     {
        // create PortSet instance used by external callers to post items
        _mainPort = new ServicePort();
        // cache dispatcher queue used to schedule tasks
        _taskQueue = taskQueue;
     }

     void GetStateHandler(GetState get)
     {
        if (_state == null)
        {
            // when state is null will post an exception
            get.ResponsePort.Post(new InvalidOperationException());
            return;
         }
         // return the state as a message on the response port
         get.ResponsePort.Post(_state);
    }

     void UpdateHandler(UpdateState update)
     {
        // update state from field in the message
        // Because the update requires a read, a merge of two strings
        // and an update, this code needs to run un-interrupted by other updates.
        // The Interleave Arbiter makes this guarantee since the UpdateHandler is in
the
       // ExclusiveReceiverGroup
       _state = update.State  + _state;

       // as success result, post the state itself
       update.ResponsePort.Post(_state);
    }

    void StopHandler(Stop stop)
    {
        Console.WriteLine("Service stopping. No other handlers are running or will run
after this");
    }
}
```

Code 12:

The example above extends the SimpleService class to use an Interleave Arbiter to coordinate the Receivers that execute the various handlers. Interleave is another example of a parent Arbiter that can have various other receivers nested. The example shows how the programmer can concisely state his/her intent in terms of concurrency: certain handlers can run independently, others cannot. The CCR doesn't need to know what resource or multi step process needs exclusive access. It only needs to know what code handler to protect. The handlers are very simple in this example, but in a later section, *iterator* handlers demonstrate how Interleave can protect complex code that runs in multiple steps.

CCR Task Scheduling

The third major component of the CCR is how tasks, generated when messages arrive on ports with active receivers, get load balanced among the execution resources of the machine. There are three important classes that implement or abstract scheduling in the CCR:

- The ***ITask*** interface and the *Task* and *IterativeTask* implementations. Only classes that implement *ITask* can be scheduled. Arbiters also implement *ITask* so they can be scheduled and properly activate.
- The ***DispatcherQueue*** class. *DispatcherQueue* is a FIFO queue of Tasks. Dispatcher queues can use the CLR thread pool for scheduling tasks (very uncommon) or an instance of a CCR Dispatcher
- The ***Dispatcher*** class. The dispatcher manages OS threads and load balances tasks de-queued from one or more *DispatcherQueue* instances.

```
1:      Dispatcher dispatcher = new Dispatcher(
            0, // zero means use one thread per CPU, or 2 if only one CPU present
            "sample dispatcher" // friendly name assigned to OS threads
            );

2:      DispatcherQueue taskQueue = new DispatcherQueue(
            "sample queue", // friendly name
            dispatcher // dispatcher instance
            );

3:      Port<int> port = new Port<int>();

4:      Arbiter.Activate(taskQueue,
           Arbiter.Receive<int>(
               true,
               port,
               delegate(int item) { Console.WriteLine(item); }
           )
        );
        // post item, so delegate executes

5:      port.Post(5);
```

CODE 13:

The example above shows how a dispatcher and dispatcher queue are created and then used to schedule tasks. The following is a step-by-step description of the example:

1. An instance of a *Dispatcher* is created, using 0 as the number of threads. This makes the CCR choose a number of threads based on the number of CPU cores reported by the OS. The number of threads per CPU, used for the default, is controlled by the static property **ThreadsPerCpu** on the Dispatcher class.
2. An instance of a *DispatcherQueue* is created, supplying the Dispatcher instance we created in step 1. This "attaches" the dispatcher queue on
3. An instance of *Port<int>* is created. We will use this port to post items and also attach a receiver with delegate.
4. The *Arbiter.Activate* method is called passing the instance of the dispatcher queue we created earlier, plus a new *Receiver* arbiter with the port it needs to listen on, plus the delegate to execute when an item is posted on the port
5. An item of type *int* is posted on the port.

When item is posted on a port with a receiver attached, the following happens within the port implementation:

1. A container is created for the value being posted. The container class, *IPortElement***,** allows the CCR to queue items and also assign them to Task instances, without caring about the item type.
2. The container instance is queued
3. If the list of receivers is not null and there is at least one receiver, the port will call the *ReceiverTask.Evaluate* method so the receiver and its Arbiter hierarchy can determine if the item can be consumed. In this example, the receiver will return true from Evaluate and also create a new instance of *Task<int>* using the item and the user delegate as parameters
4. The port logic calls *taskQueue.Enqueue* with the task returned from the *Evaluate* method on the receiver. Note that when a receiver is first activated, it is associated with the dispatcher queue instance supplied in the *Arbiter.Activate* method

After step 4 above, the generated *Task* instance is now dealt by the scheduling logic.

```
Task<int> task = new Task<int>(
   5,
   delegate(int item)
   {
      Console.WriteLine(item);
   });
taskQueue.Enqueue(task);
```

CODE 14:

In the example above we show the equivalent code for scheduling the same delegate as in example 16 but without posting anything on a port. Creating a *Task* instance explicitly is useful when data is available and code can be immediately executed to process it. The CCR does a similar task creation when a receiver is invoked in the context of a *Port.Post* call.

Once an item is queued in the dispatcher queue, the following happens:

1. The dispatcher queue signals the dispatcher instance its associated with, that a new task is available for execution
2. The dispatcher notifies one or more instances of its ***TaskExecutionWorker*** class. Each task execution worker manages one OS thread. When items are available for scheduling it puts the thread in an efficient sleep state, waiting for a signal from the dispatcher
3. An instance of *TaskExecutionWorker* calls the *DispatcherQueue**.Test*** method to retrieve a task from the queue. If a task is available (and not already picked up by another worker) the worker calls ***ITask.Execute.***
4. The *Task**.Execute*** invokes the delegate associated with the task, passing it one or more parameters associated with the task. In our example a single parameter with the value 5 is passed to the delegate that writes to the console

CCR Iterator

Iterators are a C# 2.0 language feature that is used in a novel way by the CCR: Instead of using delegates to nest asynchronous behavior (also known as callbacks), the programmer can write code in a sequential fashion, yielding to CCR arbiters or other CCR tasks. Multi-step asynchronous logic can then be all written in one iterator method, vastly improving readability of the code, maintaining the asynchronous behavior, and scaling to millions of pending operations since no OS thread is blocked during a yield.[1]

```
void StartIterator()
{
   // create an IterativeTask instance and schedule it
   Arbiter.Activate(_taskQueue,
      Arbiter.FromIteratorHandler(IteratorExample)
   );
}

/// <summary>
/// Iterator method scheduled by the CCR
/// </summary>
IEnumerator<ITask> IteratorExample()
{
   // accumulator variable
   int totalSum = 0;
   Port<int> portInt = new PortSet<int>();
   // using CCR iterators we can write traditional loops
   // and still yield to asynchronous I/O !
   for (int i = 0; i < 10; i++)
   {
      // post current iteration value to a port
      portInt.Post(i);
      // yield until a delegate executes in some other thread
      yield return Arbiter.Receive<int>(false, portInt, delegate(int j)
      {
         // this delegate can modify a stack variable, allowing it
         // communicate the result back to the iterator method
```

```
            totalSum += j;
        });
    }
    Console.WriteLine("Total:" + totalSum);
}
```

CODE 15:

In the example above the *StartIterator* method uses the *Arbiter* class to create a task from an iterator delegate and then submits it for scheduling using *Arbiter.Activate*. The second method is the iterator method, a method that can use the ***yield return*** and ***yield break*** C# statements in its body to control execution. What distinguishes this method from more common C# methods is the return value:

```
IEnumerator<ITask> IteratorExample()
{
```

The return value indicates this is a C# iterator over CCR *ITask* instances and informs the compiler that the method body might contain yield statements.

```
    yield return Arbiter.Receive<int>(false, portInt,
        delegate(int j)
        {
```

The yield return statement above returns control to the CCR scheduler. It also returns an instance of the ITask interface, implemented by the ***ReceiverTask*** created when calling the *Arbiter.Receive* helper. The CCR scheduler activates the returned Task and also associates the iterator with the task. When the task completes execution, it can choose to progress the iterator to the next code statement after the yield. When the programmer thinks about this code, they can think of the yield as a synchronization point: the iterator method will stop execution until the yield is satisfied.

Important: Never yield to persisted receivers within an iterator. In the yield statement above, Arbiter.Receive is called with **persist = false** (the first argument). If the receiver is persisted, the yield is never considered satisfied, so the iterator will never progress to the next statement after the yield

Implicit parameter passing through local variables

The power of using the CCR with the C# iterators comes from two other C# language features:

1. Anonymous methods - We have been using this feature in our examples to define the body of a handler as an in-lined delegate.
2. The compiler "captures" all local variables in the iterator method that are referenced inside the body of the anonymous method. This allows the delegate to use local variables, defined in the parent method. The delegates, that always run in

some other thread, can communicate results back to the parent iterator, with no explicit parameter passing

```
       // this delegate can modify a stack variable, allowing it
       // communicate the result back to the iterator method
       totalSum += j;
```

The body of the delegate from example 18 is shown again above. The **totalSum** variable is modified within the delegate but was defined in the parent iterator. The variable can be thought of as a result variable which can then be used at the end of the iteration (the end of the iterator method), to show the result of the multi-step asynchronous operation:

```
       Console.WriteLine("Total:" + totalSum);});
```

The line above is from the body of the iterator method *IteratorExample,* in example 18. Its the last statement in the method and it implicitly marks the end of the iteration. Notice it uses the variable modified by the delegate.

Yielding to coordination primitives

```
void StartIterator2()
{
   Port<string> portString = new Port<string>();
   Arbiter.Activate(
      _taskQueue,
      Arbiter.ReceiveWithIterator(false, portString, StringIteratorHandler)
   );
}

IEnumerator<ITask> StringIteratorHandler(string item)
{
   Console.WriteLine(item);
   yield break;
}
```

CODE 16:

The example above shows how to specify an iterator method as the outcome of a receive operation. Up to now we have been using traditional methods for the methods that execute when an arbiter is satisfied. The Arbiter class contains methods that expect an iterator delegate for all the major coordination primitives (*JoinReceiver*, *MultipleItemGather*, *Choice*, *Receiver*, etc) giving the programmer the choice between regular methods or iterator methods for every CCR coordination primitive. The arbiter implementations work with instances of *ITask*, which hides the type of the user delegate, so they work with iterator or non-iterator methods.[1]

```
IEnumerator<ITask> IteratorWithChoice()
{
   // create a service instance
   ServicePort servicePort = ServiceWithInterleave.Create(_taskQueue);
   // send an update request
   UpdateState updateRequest = new UpdateState();
   updateRequest.State = "Iterator step 1";
   servicePort.Post(updateRequest);

   string result = null;

   // wait for EITHER outcome before continuing
   yield return Arbiter.Choice(
      updateRequest,
      delegate(string response) { result = response; },
      delegate(Exception ex) { Console.WriteLine(ex); }
   );

   // if the failure branch of the choice executed, the result will be null
   // and we will terminate the iteration
   if (result == null)
      yield break;

   // print result from first request
   Console.WriteLine("UpdateState response:" + result);

   // now issue a get request
   GetState get = new GetState();
   servicePort.Post(get);

   // wait for EITHER outcome
   yield return Arbiter.Choice(
      get.ResponsePort,
      delegate(string response) { result = response; },
      delegate(Exception ex) { Console.WriteLine(ex); }
   );

   // print result from second request
   Console.WriteLine("GetState response:" + result);
}
```

CODE 17:

The example above shows a common use case for iterators: multiple asynchronous requests to a service, where the outcome of each request is success of failure. In real world examples there is usually some data dependency of request N, from the result of request N-1, which is why they are done in sequence. The ability to yield to the return value of ***Arbiter.Choice*** *(*which creates an instance of the Choice arbiter), allows the iterator to concisely handle multiple outcomes of an asynchronous I/O operation, all in-line. Note that the iterator method continues execution when **either** branch of the choice executes. It also uses the *result* stack variable to retrieve the result of the requests.

Nesting of iterators

When an iterator method grows too large to be easily maintainable, it can be decomposed further to smaller iterator methods. The CCR scheduler can determine when an iterator has exhausted all its steps which means the iterator reached a yield break or finished executing the last step of the iteration.

```
IEnumerator<ITask> ParentIteratorMethod()
{
   Console.WriteLine(
      "Yielding to another iterator that will execute N asynchronous steps"
   );

   yield return Arbiter.ExecuteToCompletion(
      _taskQueue,
      new IterativeTask<int>(10, ChildIteratorMethod)
   );

   Console.WriteLine("Child iterator completed");
}

IEnumerator<ITask> ChildIteratorMethod(int count)
{
   Port<int> portInt = new Port<int>();
   for (int i = 0; i < count; i++)
   {
       portInt.Post(i);
       yield return Arbiter.Receive(false, portInt, delegate(int j) { });
    }
}
```

CODE 18:

In the example above we show the nesting of iterators using the ***Arbiter.ExecuteToCompletion*** method:

- The parent iterator method yields execution to the task instance returned by Arbiter.ExecuteToCompletion
- *Arbiter.ExecuteToCompletion* schedules the IterativeTask instance on the dispatcher queue instance supplied. *Arbiter.FromIteratorHandler* can also be used instead of explicitly creating an IterativeTask
- The child iterator method executes in one of the dispatcher threads, and yields 10 times to a simple receive operation. A loop is used to showcase how iterators make looping around asynchronous operations very easy and readable. It also shows that the parent iterator can yield to arbitrarily complex child iterators, without knowing how many asynchronous steps they execute

Note that *Arbiter.ExecuteToCompletion* can execute non iterator tasks as well.

Important: If an exception is thrown in a handler executed in the context of a *Arbiter.ExecuteToCompletion* call, the parent iterator will still be called. Exceptions.

CCR Failure Handling

Traditional failure handling schemes follow the following patterns:

1. For synchronous invocations of methods, the caller checks one or more return values from the method. The method uses the callers execution context (most often this is a thread) to run
2. Using structured exception handling, the caller wraps synchronous method invocations in try/catch/finally statements and either relies exclusively on the catch{} block for error handling or uses a combination of the catch{} block plus explicit checks on the results from the method
3. Transactions relying on a variety of compensation implementations by the components being called, OS infrastructure and usually expensive mechanisms to flow context across threads

All methods above can't easily apply to concurrent, asynchronous execution, especially the first two which are simply not available for asynchronous programming. Methods execute in arbitrary context, potentially in parallel with the caller.

The CCR addresses failure handling with two approaches:

1. **Explicit or local** failure handling using the Choice and MultipleItemGather arbiters. Combined with iterator support they provide a type safe, robust way to deal with failure since they force the programmer to deal with the success and failure case in two different methods and then also have a common continuation. **Examples code 10 and 19** show explicit error handling using *Choice*, *MultipleItemGather* and *Choice* in an iterator, respectively.
2. **Implicit or distributed** failure handling, referred to as ***Causalities*** that allows for nesting and extends across multiple asynchronous operations.It shares in common with transactions the notion of a logical context or grouping of operations and extends it for a concurrent asynchronous environment. This mechanism can also be extended across machine boundaries

1.4. Decentralized System Services (DSS)

Choosing the right programming language that suits you is an important task. Microsoft Robotics Studio supports a wide variety of programming languages ranging from traditional .NET languages like C# and VB.Net to scripting languages such as Python to Microsoft Visual Programming Language (VPL) which is a brand-new visual programming language provided as part of the Microsoft Robotics Studio.

Microsoft Visual Programming Language (VPL)

VPL Introduction is an application development environment designed around a graphical dataflow-based programming model rather than control flow typically found in conventional programming. Rather than series of imperative commands sequentially executed, a dataflow program is more like a series of workers on an assembly line, who do their assigned task as the materials arrive. As a result VPL is well suited to programming a variety of concurrent or distributed processing scenarios.VPL is targeted for beginning programmers with a basic understanding of concepts like variables and logic. However, VPL is not limited to novices. The compositional nature of the programming language may appeal to more advanced programmers for rapid prototyping or code development. In addition, while the contents of its toolbox is tailored developing robot applications, the underlying architecture is not limited to programming robots and could be applied to other applications. As a result, VPL may appeal to a wide audience of users including students, enthusiasts/hobbyists, as well as possibly web developers and professional programmers.[2]

Service Component

Service is the basic building block for writing applications using Microsoft Robotics Studio and is a key component of the DSS application model. Services can be used to represent anything including but not limited to:

1. Hardware components such as sensors and actuators
2. Software components such as User Interface, storage, directory services, etc.
3. Aggregations: Sensor fusion, mash-ups, etc

Services are executed within the context of a DSS Node. A DSS node is a hosting environment that provides support for services to be created and managed until they are deleted or the DSS node is stopped. Services are inherently network enabled and can communicate with each other in a uniform manner regardless of whether they are executed within the same DSS node or across the network.

The DSS service model has been designed to facilitate reuse of services by making them easy to use and compose with each other while enforcing a very loose coupling between them. All DSS services consist of a common set of components as described in this section.

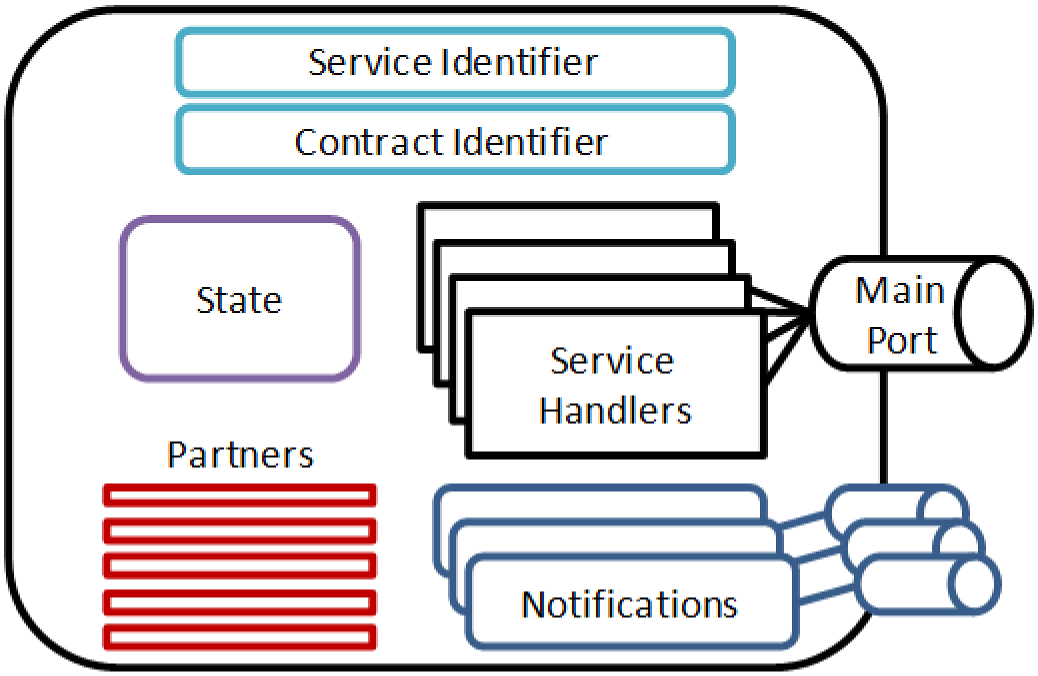


FIGURE 5:

Service Identifier

When a service instance is created within a DSS Node, it is dynamically assigned a URI by the Constructor Service. This service identifier refers to a particular instance of a service identifier running on a particular DSS node.

The service identifier is what enables other services to communicate with that service as well as Accessing Services through a Web Browser. The service identifier only provides identity of a service instance; it does not convey information about the service state, behaviour, or context. In other words, it is never safe to guess about what a service is based on the service identifier.

The service identifier is determined by the following steps:

1. If a service instance name is indicated in Service Manifests then that name is used
2. If a service instance name is indicated in a Create request sent directly to the System Services then that name is used
3. If a service implementation has a ServicePort attribute looking like this
4. `[ServicePort("/mysample")]`, then the name is the host and port name of the DSS Node with "/mysample" appended. For example http://machine:port/mysample

5. If a service implementation has a ServicePort attribute with a `AllowMultipleInstances = true` like this
6. `[ServicePort("/mysample", AllowMultipleInstances = true)]`, then the name of the service instance is the same as above but with a GUID appended to further differentiating this instance from other instances: `http://machine:port/mysample/d88441c9-8319-407c-b554-0b0bfd90050b`

Contract Identifier

A contract is a condensed description of a service implementation that describes its behavior so that other services can compose and reuse services with a given contract. The contract of a service can be inspected using the DSS Contract Information Tool (DssInfo.exe). Contracts are used to generate DSS proxy DLLs which is what services link against when talking to other services rather than linking directly with each other.

A contract identifier is a URI that uniquely identifies the contract of a service. A contract identifier is automatically created when creating a service project, for example using DSS New Service Generation Tool (DssNewService.exe). The contract identifier can be modified but by it is of the form:

```
http://schemas.tempuri.org/[year]/[month]/[name].html
```

Service State

The service state is a representation of a service at any given point in time. One way to think of the service state is as a document that describes the current content of a service. Examples of state documents include:

1. The state of a service representing a motor may consist of rotations per minute, temperature, oil pressure, and fuel consumption.
2. A service representing a work queue may contain a list of all queued work items and their current status. The work items themselves may be services allowing the work queue to simply refer to them using their identity.
3. A service representing a keyboard may contain information about which keys have been pressed.

Any information that is to be retrieved, modified, or monitored as part of a DSS service must be expressed as part of the service state.

In addition to capturing the current share of a service, the state also can be used to drive the UI of a service.

Service Partners

A critical part of the DSS application model is to enable services to compose with each other to provide higher level functions. This means that in order for services to effectively take advantage of each other, they must be able to discover and establish communication with each other in an efficient and predictable manner. However, as services are loosely coupled, a service does not know apriori whether other services that it depend on are available or even where these other services are. To address this issue, the DSS application model provides the notion of partner services.

Partner services are other services that a service interacts with and possibly depends on in order to function properly. By declaring a set of other services as partners, a service can indicate to the runtime that it wants to be wired up with these services as part of the creation process of the service itself. A partner is declared using the Partner attribute which has a rich set of options controlling exactly what the partner relationship is with each of the declared partners. For example, a service can declare that a partner is required in order for the service to run and if that partner can't be found then there is no need for the service to start at all. Partners can also be declared as being optional in which case the runtime attempts to wire up the service with the partner but if it doesn't succeed then the creation process continues regardless.

Main Port

The main port is a CCR port where messages from other services arrive (a.k.a. the operations port). Because service implementations do not link against each other directly, a service can only talk to another service by sending a message to its main port. The main port itself is a private member of the service class and is identified by the ServicePort attribute. An example of a main port declaration is:

```
[ServicePort("/mysample")]
private MySampleOperations _mainPort = new MySampleOperations();
```

The messages accepted on the main port are defined by the type of the port, in the case above by the type MySampleOperations. All operations defined on the main port must either follow the message operations defined by DSSP or by HTTP. An example of a main port definition that supports the DSSP operations LOOKUP, DROP, GET, and REPLACE, as well as the HTTP operations GET and POST is

```
public class MySampleOperation
   : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, Replace, HttpGet, HttpPost>
{

}
```

Service Handlers

For each of the DSSP operations defined on the main port, service handlers need to be registered to handle incoming messages arriving on that port. The only exception is the handling of the operations DsspDefaultLookup and DsspDefaultDrop for which the DSS runtime registered default handlers. For example, in the case of the MySampleOperations definition above, the MySample service would register handlers for Get, Replace, HttpGet, and HttpPost.

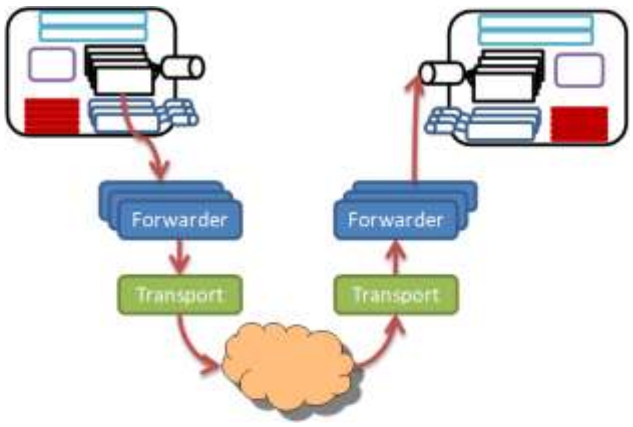
Service handlers can be registered declaratively using the ServiceHandler attribute. This example registers a service handler for the DSSP GET operation:

```
[ServiceHandler]
public IEnumerator<ITask> GetHandler(Get get)
{
    get.ResponsePort.Post(_state);
    yield break;
}
```

Within the context of a service handler a service can send messages to other services. There are two ways a service can send messages:

1. Unsolicited in the form of a request message sent to another service.
2. Solicited in the form of an event notification sent to a subscriber as a result of a state change within the service generating the event notification.

In either case, messages are sent through a **SERVICE FORWARDER** which is a local CCR Port representing the main port of the remote service. When a message is sent through the service forwarder, it will get forwarded down through the runtime until it reaches a transport that will route the message, possibly through the network, to the transport of the other service. Here the message will get forwarded back up through the runtime until it reaches the main port of the receiving service.



:

FIGURE 6:

Event Notifications

A common pattern used in DSS services is that of subscribing to other services. A service generates event notifications as a result of changes to its own state. For each subscription that a service has established with other services, the service will receive event notifications on separate CCR ports. By using different ports for each subscription, it is possible to differentiate event notifications and determine which subscription each event notification is associated with. Furthermore, because event notifications arrive on ports, it is possible to orchestrate event notification messages using the full spectrum of CCR primitives.

Because event notifications are generated as a result of changes to the state of a service, and state changes happen as a result of DSSP operations such as DELETE, INSERT, UPDATE, etc. the actual event notification messages are exactly the set of operations on the main port that affect state change. As a result, a local notification port where event notifications arrive is of the same type as the main port of the service associated with that particular subscription. For example, if we introduce a service called MySample2 and MySample subscribes to that service. The type of the local notification port in MySample service where notifications arrive as a result of state changes in the MySample2 service is the same as that of the MySample2 main port.

Service Data Model

The service state is a representation of a service at any given point in time. One way to think of the service state is as a document that describes the current content of a service. In addition to capturing the current share of a service, the state is also used to drive the Web-based UI of the service.

When defining the state of a service you must decorate the CLR classes, fields, and properties using the DataContractAttribute, DataMemberAttribute, and DataMemberConstructorAttribute in order to include the types in the generated DSS proxy DLL. The DSS proxy generation uses an "opt-in" model where anything not explicitly decorated will not be included in the generated proxy DLL.

DataContractAttribute

The DataContractAttribute is used on Classes, Structs, and Enumerations to indicate that these types have to be included in the DSS proxy DLL generated for this service implementation as part of the public contract.

DataMemberAttribute

The DataMemberAttribute is used on Fields and Properties to indicate that these members have to be included in the DSS proxy DLL generated for this service implementation as part of the public contract.

DataMemberConstructorAttribute

The DataMemberConstructorAttribute can be used to indicate that a constructor overload containing initialization parameters should be automatically generated in the proxy DLL. The attribute can be used in combination with either the DataContractAttribute or the DataMemberAttribute as follows:

- Used in combination with DataContractAttribute this attribute causes the generation of a constructor with every field or property marked with the DataMemberAttribute.
- Used in combination with DataMemberAttribute this attribute controls the order in which the fields or properties are listed in the generated constructor. If the order is set to 0 (default), the lexical order is preserved and if set to -1 the field or property is explicitly not included in the generated constructor.

Application Configuration File

Several DSS runtime features including message logging, generation of debug messages, and handling of service request timeouts can be configured through an application configuration file. An application configuration is an XML file that must have the same name as the application, with a `.config` extension appended, and be located in the same folder as the application itself.

When running services using the DSS Host Tool (DssHost.exe), the application configuration file is called `dsshost.exe.config`. However, the controls can also be applied to an executable that starts the DSS runtime using the DssEnvironment static class. In this case, the name of the application configuration file is that of the application executable with `.config` appended.

Message Capturing and Logging

The DSS runtime can be configured to serialize messages between services and store them in log files named after the URI of the service instance sending or receiving each messages. To control the level of message capturing and logging, open the application configuration file and look for the section named `appSettings` and identify the line containing a key named `Microsoft.Dss.Services.Forwarders.MessageCapture`:

```
<appSettings>
   <add key="Microsoft.Dss.Services.Forwarders.MessageCapture" value=""/>
</appSettings>
```

The available values for this option are:

CaptureInbound

Using the `CaptureInbound` value, all messages received by a service, are serialized to XML, and saved to disk.

```
<add
   key="Microsoft.Dss.Services.Forwarders.MessageCapture"
   value="CaptureInbound"/>
```

CaptureOutbound

Using the `CaptureOutbound` value, all messages sent by a service are serialized to XML, and saved to disk.

```
<add
   key="Microsoft.Dss.Services.Forwarders.MessageCapture"
   value="CaptureOutbound"/>
```

CaptureInboundOutbound

Using the `CaptureInboundOutbound` value, all messages received or sent by a service are serialized to XML, and saved to disk.

```
<add
   key="Microsoft.Dss.Services.Forwarders.MessageCapture"
   value="CaptureInboundOutbound"/>
```

Log files are placed in the `store\logs` directory and can be accessed while the DSS node is running. Services that are set up to have a fixed URI will always use the same log file. Services that are set up to use different URIs for each instance will use different log files located in a directory named after the service prefix.

Debug and Trace Messages

The DSS Runtime uses the *`System.Diagnostics.Trace`* class to log a variety of status messages while running. Trace messages are categories according to a set of trace switches that can be configured to show messages according to their level of severity ranging from **VERBOSE**, **INFORMATIONAL**, **WARNING**, to **ERROR**. Messages written to the trace switches can be inspected in one of two ways:

1. When a DSS node is running, the set of debug and trace messages can be seen by accessing the Console Output service which is a System Services and accessible from a Web browser.
2. Messages can be captured by various tools such as DebugView

The trace switches defined by Microsoft Robotics Runtime are as follows:

Microsoft.Ccr.Core

Contains messages generated by CCR which is part of the Microsoft Robotics runtime

Microsoft.Dss.Core

Contains messages generated by DSS which is part of the Microsoft Robotics runtime

Microsoft.Dss.Services.TestBase

Contains messages generated by the Service Testing Infrastructure.

Microsoft.Dss.Services

Contains messages generated by the System Services such as the directory and contract directory.

Microsoft.Dss.Services.Transports

Contains messages generated by the network transport layer of the DSS runtime

Microsoft.Dss.Services.Forwarders

Contains messages generated by the message forwarders which sit between a DSS service and the transport.

To configure the trace switches, open the application configuration file and look for the section named switches which is a part of the larger section named System.Diagnostics:

```
<switches>
   <add name="Microsoft.Ccr.Core"   value="2" />
   <add name="Microsoft.Dss.Core"   value="3" />
   <add name="Microsoft.Dss.Services.TestBase"   value="3" />
   <add name="Microsoft.Dss.Services"   value="3" />
   <add name="Microsoft.Dss.Services.Transports"   value="2" />
   <add name="Microsoft.Dss.Services.Forwarders"   value="2" />
</switches>
```

The value of each switch can be one of the following values:

Value	Description
0	Trace switch is turned off.
1	Only messages with trace level **ERROR** are shown.

2	Messages with trace level **ERROR** and **WARNING** are shown
3	Messages with trace level **ERROR**, **WARNING**, and **INFORMATIONAL** are shown
4	All messages with trace level **ERROR**, **WARNING**, **INFORMATIONAL**, and **VERBOSE** are shown

Service authors can define their own trace switches using the *System.Diagnostics.TraceSwitch* .Net Framework class. By adding them to the application configuration file in a similar manner to that of the trace switches provided by Microsoft Robotics Studio, they can be controlled in exactly the same manner as described above.

Accessing Services through a Web Browser

The DSS Service model is an extension of the classic Web architecture model and is compatible with existing Web infrastructure allowing for integration with common tools such as Web browsers. For example, services can generate UI in a manner that allows users to both access and update services directly through a Web browser.

Responding to HTTP GET Requests

Services can respond to HTTP GET requests in one of three ways depending on the HTTP request itself and which service handlers the service has registered to handle requests arriving on its main port.

HTTP Get Service Handler

If the service has registered a service handler for the *Microsoft.Dss.Core.DsspHttp.HttpGet* DSSP Operation on the main port of the service, it will respond to requests where the HTTP Request URI does **not** contain a query component. That is, HTTP GET requests where the request URI does not contain a question mark "?" will be handled by this service handler.

An HTTP GET request is a request for the entire state of a service which means that a successful HTTP GET response contains a snapshot of the entire service state. The service state can be represented in an open-ended set of data formats ranging from XML, HTML, image formats, video, audio, etc. Most of the System Services generate XML with an associated XSLT that converts the state into HTML for display in a browser. However, the WebCam Service is an example of how to generate multiple data formats within an HTTP Get handler.

HTTP Query Service Handler

If the service has registered a service handler for the *Microsoft.Dss.Core.DsspHttp.HttpQuery* DSSP Operation on the main port of the service, it will respond to requests where the HTTP Request URI **does** contain a query component. That is for URIs that do contain a question mark "?" like for example

```
http://example.org/myservice?query=keyword
```

An HTTP Query request is a request for parts of the state of a service where the query is encoded as name-value pairs in the URI. Like for the HTTP Get Service handler, the response to a query can be represented in an open-ended set of data formats ranging from XML, HTML, image formats, video, audio, etc.

DSSP GET Service Handler

If a service has a service handler registered for the *Microsoft.Dss.ServiceModel.Dssp.Get* DSSP Operation on the main port of the service but no HTTP Get or Query service handler it will still respond to regular HTTP requests. While DSSP GET semantically is identical to HTTP GET, the difference from the responses generated by HTTP Get and Query handlers is that the response is an actual SOAP envelope containing the state of the service within the SOAP Body element.

Responding to HTTP POST Requests

Services can respond to HTTP POST requests if it has a service handler registered for the *Microsoft.Dss.ServiceModel.Dssp.HttpPost* DSSP Operation on the main port. The service handler can either read the HTTP request entity as an unstructured stream or as a name value collection.

Responding to HTTP PUT Requests

Services can respond to HTTP PUT requests if it has a service handler registered for the *Microsoft.Dss.ServiceModel.Dssp.HttpPut* DSSP Operation on the main port. The service handler would typically read the entire HTTP request entity as an unstructured stream and feeds it to some other stream acting as a data sink. An HTTP PUT request is a request for replacing the state of a service with the state included in the HTTP PUT request itself. HTTP PUT requests are supported by the Mount Service where the data is persisted to disk allowing to be retrieved later using a GET request.

Well-known Service URIs

When a service instance is created within a DSS Node, it is dynamically assigned a URI by the Constructor Service. This service identifier refers to a particular instance of a service identifier running on a particular DSS node. The service identifier is what enables other services to communicate with that service as well as Accessing Services through a Web Browser. The service identifier only provides identity of a service instance; it does not convey information about the service state, behaviour, or context. In other words, it is never safe to guess about what a service is based on the service identifier.

Directory Service

While the `ServicePaths` class defined in the DSS runtime contains constants for some common service instances, in most cases, service URIs should be looked up dynamically in the Directory Service which is one of the System Services. Within a DSS Node the Directory Service is guaranteed to have the relative name `/directory` like for example:

```
http://localhost:50000/directory
```

Accessing Services through DSSP

DSS uses HTTP and DSSP as the foundation for interacting with services. DSSP is a lightweight SOAP-based protocol that provides support for manipulation of structured state and for an event model driven by changes to the structured state. DSSP is used for manipulating and subscribing to services and hence compliments HTTP in providing a simple, state-driven application model. By enabling services to be accessed both as regular HTTP resources or as DSSP services, it is possible to extend the HTTP model with support for structured data manipulation, event notification, and partner management. Because HTTP and DSSP have inherently different protocol characteristics, DSSP is complementary to HTTP and not intended as a replacement.

Responding to CREATE Requests

The CREATE operation is used to create a new instance of a service and is normally only supported by the Constructor service. While it is possible to write additional constructor services, most scenarios are typically better covered by writing services on top of the existing Constructor service. The Manifest Loader service is an example of a service that uses the Constructor service to create new instances of services listed in a manifest.

As part of the service creation process, you can provide the service with information about its partners including the initial state partner which can be used to initialize the service with a known state.

Responding to DROP Requests

The DROP operation is used to shut down a service. When a service accepts a DROP request it can no longer be reached by other services. If a service does not have any particular shutdown logic it can use the `DefaultDropHandler` which removes the service from the directory service and shuts down the service instance. This handler is automatically registered so a service implementation does not need to provide anything explicit in order to enable the default behavior.

Services that do have special shutdown logic can provide their own DROP handler which augment the basic shutdown logic as follows:

```
[ServiceHandler(ServiceHandlerBehavior.Teardown)]
public virtual IEnumerator<ITask> DropHandler(DsspDefaultDrop drop)
{
    // Perform service specific shutdown logic including shutting
    // down dependent service instances

    // Perform default shutdown logic
    base.DefaultDropHandler(drop);

    // Shutdown complete
    yield break;
}
```

Code 19:

This handler will get picked up automatically and registered when calling *DsspServiceBase.Start()*.

Note: Service instances may share a CCR dispatcher so issuing a DROP on a service does not guarantee that CLR threads are removed. This means that if your service started some CCR logic that periodically does something, that will continue after the DROP operation has completed.

Responding to LOOKUP Requests

The LOOKUP operation provides information about a service including what contract (or contracts) the service is implementing and what partners this particular service instance has. All services must support the LOOKUP operation so it is always present on the service operations port.

The DSS runtime already has all relevant information about a service instance to be able to respond to a LOOKUP request automatically so the *DefaultLookuphandler* is typically sufficient to handle LOOKUP requests. While the default behavior can be overridden by explicitly registering a LOOKUP handler, this is not recommended under normal circumstances.

Responding to State Retrieval Requests

DSSP defines two operations for service state retrieval: GET and QUERY. While the two operations are similar, the QUERY request contains a service-specific, structured query against the service state whereas a GET request is an service-independent request for the entire service state.

Because the GET request is a request for the entire state and the request type is pre-defined by DSSP, there can be at most one GET handler. However, there may well be multiple queries against the state of a service that may warrant multiple QUERY handlers supporting different query types.

Note: As both GET and QUERY requests are defined by DSSP as not having any side-effects, GET and QUERY handlers should be marked as ServiceHandlerBehavior.Concurrent using the ServiceHandler attribute so that the infrastructure knows that these handlers can run concurrently.

Responding to GET Requests

The DSSP GET operation has a pre-defined GET request type but a service-specific GET response containing a representation of the state of the service. A very common GET handler looks like this:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> GetHandler(Get get)
{
    get.ResponsePort.Post(_state);
    yield break;
}
```

CODE 20:

Responding to QUERY Requests

A QUERY operation is a query over the service state meaning that the state included in a QUERY response is a subset of the entire state of a service available in a GET response. A service can have any number of QUERY handlers supporting different QUERY request types. A typical QUERY handler looks like the following:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> QueryHandler(Query query)
{
    // Create filtered view of service state using the parameters provided in the
query

    query.ResponsePort.Post(_filteredState);
    yield break;
}
```

CODE 21:

DSSP does not limit the way in which queries can be expressed leaving it up to the service to determine the most appropriate way for querying its state. Common examples include name-value-based queries, or by the service defining specific query types that generate views over the service state.

Responding to SUBSCRIBE Requests

The SUBSCRIBE operation allows other services to subscribe to state changes. Subscriptions are widely used in DSS services to enable subscribers to track state changes.

Service implementing support for subscriptions can use the Subscription Manager service for managing their subscriptions rather than doing it themselves. The following is an example of a typical SUBSCRIBE handler using a Subscription Manager service previously set up as a partner service.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<iTask> SubscribeHandler(Subscribe subscribe)
{
    yield return Arbiter.Choice(SubscribeHelper(_submgrPort, subscribe.Body,
subscribe.ResponsePort),
        delegate(SuccessResult success)
        {
            // Send notification (or notifications) using
DsspServiceBase.SendNotification()
            // allowing the newly subscribed party to catch up to the current state of
this service.
            // In this example we use Replace to catch up the newly subscribed party.
            base.SendNotification<Replace>(_submgrPort, subscribe.Body.Subscriber,
_state);
        },
        delegate(Exception failure)
        {
            LogError(null, "Unable to subscribe", failure);
        }
    );
    yield break;
}
```

CODE 22:

Responding to State Changes Requests

The following cover DSSP operations that change the state of a service. State changes can result in the service state being inserted, updated, or deleted and causes event notifications to be generated as a result.

Note: As state changes by the very nature have side-effects, the operations described in this section should be marked as ServiceHandlerBehavior.Exclusive using the ServiceHandler attribute so that the infrastructure knows that these handlers must run exclusive to all other handlers registered as part of the same interleave.

Responding to INSERT Requests

The INSERT operation is used to add new state to the service. Below is a typical INSERT operation handler that performs the following sequence of instructions:

- Check whether the data is valid or generate a OperationFailed fault message.
- Check the data for any uniqueness constraints and if not met then generate a DefaultInsertResponse fault message.
- Add data included in insert request to service state

- Send an INSERT event notification using the Subscription Manager service previously created
- Send a DefaultInsertResponseType.Instance message

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> InsertHandler(Insert insert)
{
    // Validate insert data
    if ( /* data is not valid */ )
    {
        insert.ResponsePort.Post(
            Fault.FromCodeSubcodeReason(
                FaultCodes.Sender,
                DsspFaultCodes.OperationFailed));
        yield break;
    }

    // Check uniqueness of data
    if ( /* uniqueness constraint not met */)
    {
        insert.ResponsePort.Post(
            Fault.FromCodeSubcodeReason(
                FaultCodes.Sender,
                DsspFaultCodes.DuplicateEntry));
        yield break;
    }

    // Add data included in insert request to service state

    // Send notification
    base.SendNotification(_submgrPort, insert);

    // Send response
    insert.ResponsePort.Post(DefaultInsertResponseType.Instance);
    yield break;
}
```

CODE 23:

Responding to UPDATE Requests

The UPDATE operation is used to update the current service state with new data included in the request. The typical UPDATE handler looks very similar to the INSERT handler above but with a change from Insert to Update message types:

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<iTask> UpdateHandler(update update)
{
    // Validate update data
    if (false)
    {
        update.ResponsePort.Post(
            Fault.FromCodeSubcodeReason(
                FaultCodes.Sender,
                DsspFaultCodes.OperationFailed));
        yield break;
    }

    // Check existence of data
    if (false)
    {
        update.ResponsePort.Post(
            Fault.FromCodeSubcodeReason(
                FaultCodes.Sender,
                DsspFaultCodes.UnknownEntry));
        yield break;
    }

    // Update service state with data included in update request

    // Send notification
    base.SendNotification(_submgrPort, update);

    // Send response
    update.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}
```

CODE 24:

Responding to UPSERT Requests

The UPSERT operation is used to either update part of the current service state if the data is already present or insert the data included in the request state as new state. Logically the UPSERT operation merges the INSERT and UPDATE semantics but with the benefit that the entire state changed can be done as a single operation. The typical UPSERT handler follows the same pattern as indicated for INSERT and UPDATE above.

Responding to DELETE Requests

The DELETE operation is used to delete existing state. The typical handler for DELETE operations also follows the same pattern as already laid out above in terms of steps that it goes through.

Responding to REPLACE Requests

The REPLACE operation can be used to replace the entire service state regardless of the existing service state. An example of a typical REPLACE handler is shown below where the event notification is managed by the Subscription Manager services.

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<iTask> ReplaceHandler(Replace replace)
{
    // Assign new state received in request to service state
    _state = replace.Body;

    // Send REPLACE event notification
    base.SendNotification(_submgrPort, replace);

    // Send response to the request
    replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
    yield break;
}
```

CODE 25:

Responding to SUBMIT Requests

The SUBMIT operation can be used to represent semantics that is not possible or practical to represent as state changes. The SUBMIT operation is explicitly defined to allow computations that do not change the state of a service and as a result do not generate any event notifications.

Chapter 2

Microsoft Visual Programming Language

2.1. Introduction

Microsoft Visual Programming Language (VPL) is an application development environment designed on a graphical data-flow-based programming model rather than control flow typically found in conventional programming. Rather than series of imperative commands sequentially executed, a data-flow program is more like a series of workers on an assembly line, who do their assigned task as the materials arrive. As a result VPL is well suited to programming a variety of concurrent or distributed processing scenarios. VPL is targeted for beginner programmers with a basic understanding of concepts like variables and logic. However, VPL is not limited to novices. The compositional nature of the programming language may appeal to more advanced programmers for rapid prototyping or code development. In addition, while its toolbox is tailored developing robot applications, the underlying architecture is not limited to programming robots and could be applied to other applications as well.

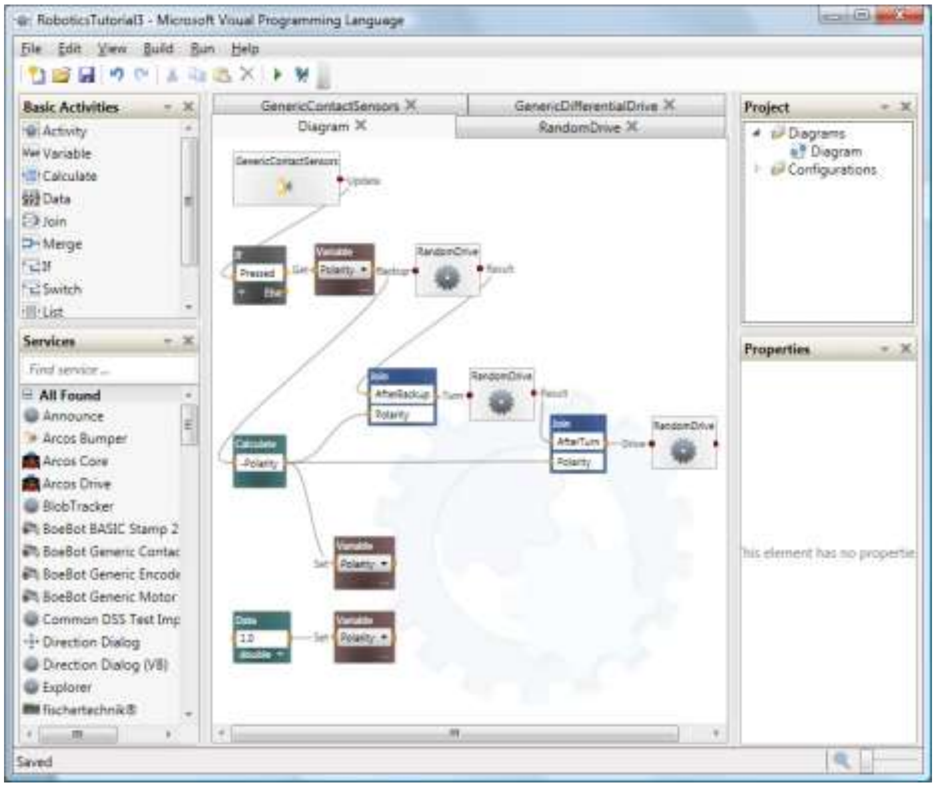


FIGURE 7: SAMPLE MICROSOFT VISUAL PROGRAMMING LANGUAGE (VPL) DIAGRAM FOR SIMPLE BUMP TURN GO WANDER BEHAVIOR

A Microsoft Visual Programming Language data-flow consists of a connected sequence of *activities* represented as blocks with inputs and outputs that can be connected to other activity blocks.

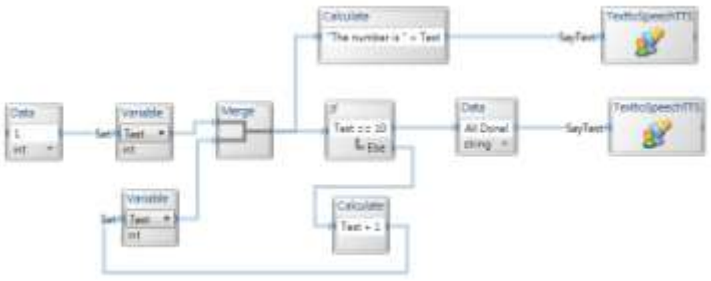


FIGURE 8: THE LINK ARROWS BETWEEN THE BLOCK REPRESENT MESSAGES THAT SEND DATA FROM ONE ACTIVITY TO ANOTHER.

2.2. Activities

Activities can represent pre-built services, data-flow control, functions, or other code modules. The resulting application is therefore often referred to as *orchestration*, the sequencing of separate processes. Activities can also include compositions of other of activities. This makes it possible to compose activities and reuse the composition as a building block. In this sense an application built in VPL is itself an activity. Activity blocks typically include the activity's name and borders that represent its connection points. An activity block may also include graphics to illustrate the purpose of the activity as well as user interface elements that may enable the user to enter values, assignments, or transformations for data used in an activity.

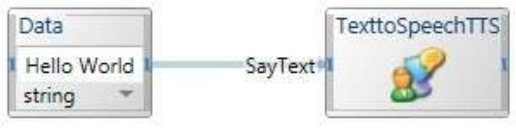


FIGURE 9:

Activities are connected through their connection pins. A connection pin on the left side of an activity represents connection point for incoming/input messages and a pin on the right represents its connection point for outgoing/output messages. An activity receives messages containing data through its input connection pins. An activity's input pins are connection points to its predefined internal functions known as actions or handlers (which can be either as functions provided by a service or nested data-flows). An activity block activates and processes the incoming message data as soon as it receives a valid incoming message. All data sent to the activity is consumed by the activity. For data to be forwarded on through the activity's output, the receiving activity must replicate the data and put it on its output connection. An activity's input pins are connection points to its predefined internal functions known as actions or handlers. An activity may have multiple input connection pins and each with its own set of output connection pins.

Output connection pins can be one of two kinds: a *result* output or *notification* output (sometimes also referred to as an event or publication output). Result outputs are displayed as rectangular connection pins while publication outputs have round connection pins.

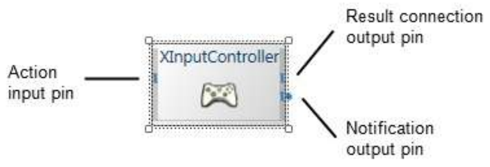


FIGURE 10:

A response output pin is used in situations where an outgoing message (data) is sent as the result of a specific incoming action message. Notification pins can send information resulting from an incoming message, but more typically fire a message as they change their internal state. They can also generate messages multiple times, whereas a result pin only sends out a single message on the receipt of an incoming message. So notification output pins are used for sending message data without having to repeatedly request or poll for the state of an activity.

Chapter 3

Microsoft Simulation Environment

3.1. Introduction

Microsoft Robotics Studio targets a wide audience in an attempt to accelerate robotics development and adoption. An important part of this effort is the simulation runtime. It was immediately obvious that PC and Console gaming has paved the way when it comes to affordable, widely usable, robotics simulation. Games rely on photo-realistic visualizations with advanced physics simulation running within real time constraints. This was a perfect starting point for our effort. We designed the simulation runtime to be used in a variety of advanced scenarios with high demands for fidelity, visualization, and scaling. At the same time, a novice user with little to no coding experience can use simulation; developing interesting applications in a game-like environment. Our integration of the AGEIA PhysX Technologies enables us to leverage a very strong physics simulation product that is mature and constantly evolving towards features that will be invaluable to robotics. The rendering engine is based on Microsoft XNA Framework.

In this part, we discuss:

- Simulation Architecture
- Simulation Runtime
- Simulation Programming
- Simulation Screenshots

3.2. Simulation Architecture

The simulation's architecture is represented in the following figure. It's composed by:

- An user interface
- Simulation engine service
- AGEIA Physics
- XNA Graphics Library
- Display Hardware

User interface is used in order to modify entities at runtime.

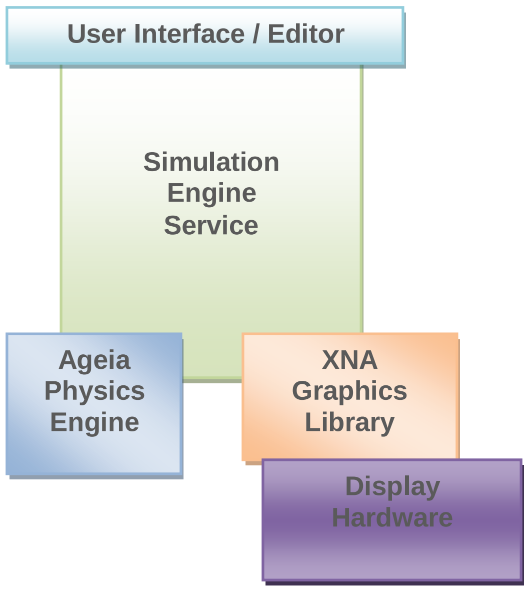


FIGURE 11

When we define one or more entities any examples are: *Sky, Ground, Robot and Camera;* we must insert them into level simulation engine as seen in figure 12.

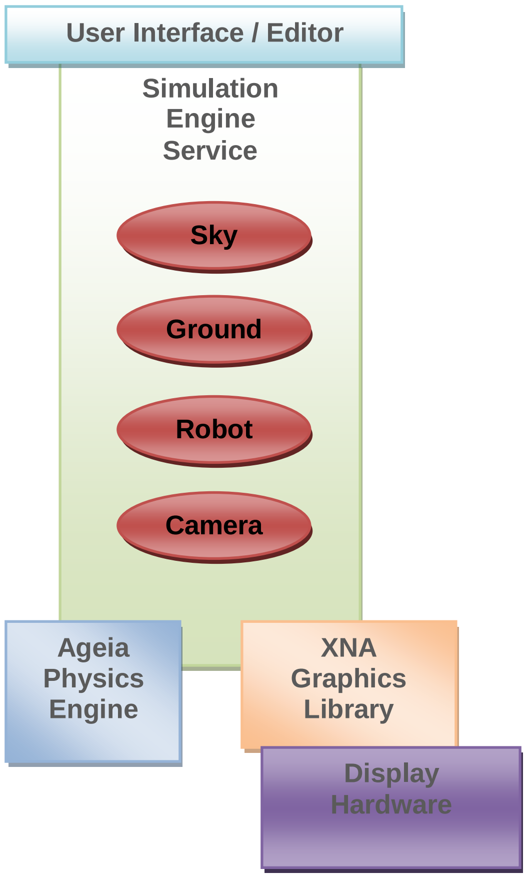


FIGURE 12

The controller service is connected to any entities, examples Robot and Camera. They are controlled by Drive Service and Webcam Service. It’s seen into figure 13.

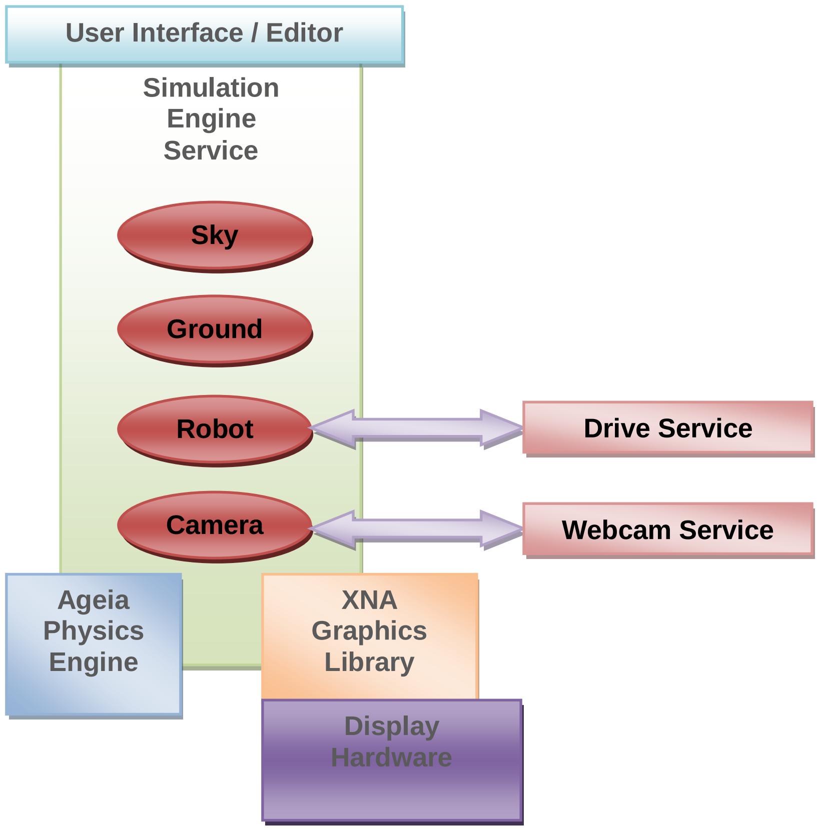


FIGURE 13

An example can run a simulation service is seen in to following code:

```
[DisplayName("Simulation Tutorial 2")]
[Description("Simulation Tutorial 2 Service")]
[Contract(Contract.Identifier)]
public class SimulationTutorial2 : DsspServiceBase
{
   State _state = new State();

   // partner attribute will cause simulation engine service to start
   [Partner("Engine", Contract = engineproxy.Contract.Identifier,
      CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
   private engineproxy.SimulationEnginePort _engineServicePort =
          new engineproxy.SimulationEnginePort();
…
}
```

CODE 26:

3.3. Simulation Runtime

The simulation runtime is composed of the following components:

- The **Simulation Engine Service**, is responsible for rendering entities and progressing the simulation time for the physics engine. It tracks of the entire simulation world state and provides the service/distributed front end to the simulation.
- The **Managed Physics Engine Wrapper**, abstracts the user from the low level physics engine API, provides a more concise, managed interface to the physics simulation.
- The **Native Physics Engine Library,** enables hardware acceleration through AGEIA PhysX Technology, which support hardware acceleration through the AGEIA PhysX Technology processor.
- **Entities**, represent hardware and physical objects in the simulation world. A number of entities come predefined with the Microsoft Robotics Studio and enable users to quickly assemble them and build rich simulated robot platforms in various virtual environments.

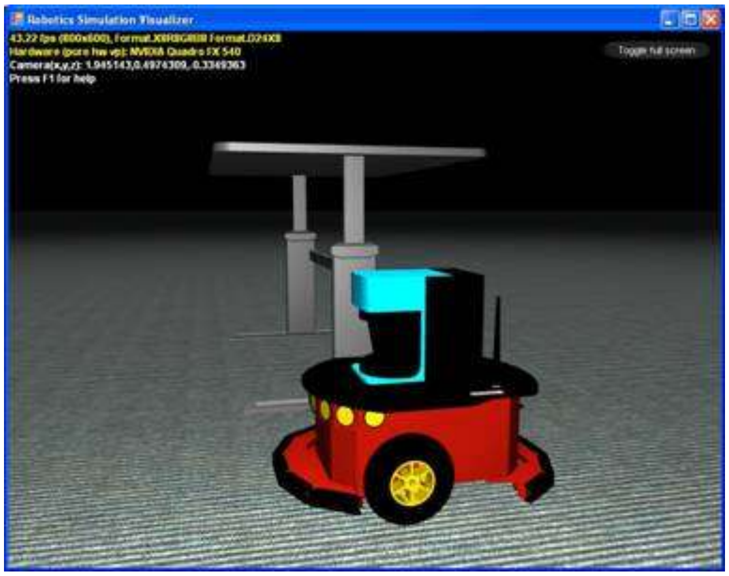
The rendering engine uses the programmable pipeline of graphics accelerator cards, conforming to Directx9 Pixel/Vertex Shader standards. In the simulation tutorials we will talk how the simulation runtime makes it easy to supply your own effects and have the engine manage loading, rendering, updates to effect state, etc.

3.4. Simulation Programming

Two software components are usually involved for simulating a physical component and its service:

- An **Entity**, is the software component that interfaces with the physics engine and the rendering engine. It exposes the appropriate high level interfaces to emulate hardware and hide the specific use of physics APIs.
- A **Service**, that uses the same types and operations as the service it is simulating and provides the distributed front end to the entity, just like robotics services provide the front end to robot hardware.

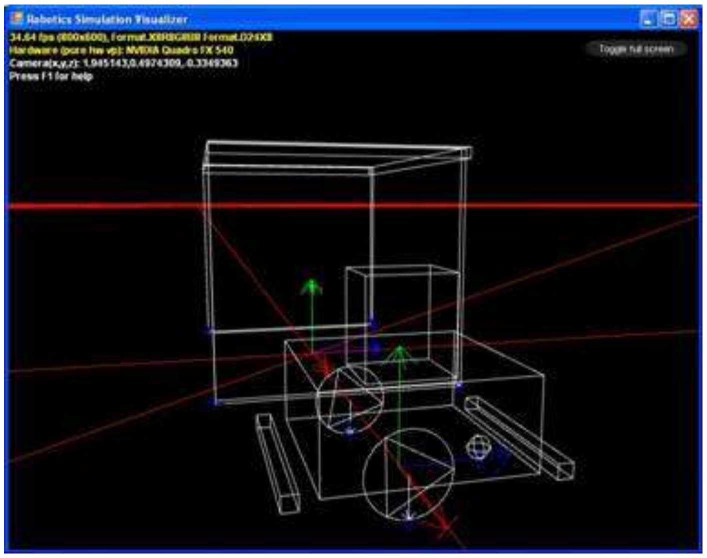
3.5. Simulation Screenshots



```
43.22 fps (800x600), Format.X8R8G8B8 Format.D24X8
Hardware (pure hw vp): NVIDIA Quadro FX 540
Camera(x,y,z): 1.945143,0.4974309,-0.3349363
Press F1 for help
```

FIGURE 14:

The first image represents a robot base view with differential drive, laser range finder and bumper array. The second image is of the physics primitive view, which shows how the table and robot are approximated by solid shapes.



```
34.64 fps (800x600), Format.X8R8G8B8 Format.D24X8
Hardware (pure hw vp): NVIDIA Quadro FX 540
Camera(x,y,z): 1.945143,0.4974309,-0.3349363
Press F1 for help
```

FIGURE 15:

These images represent close-up of a multi shape environment entity and its physics model too.

FIGURE 16:

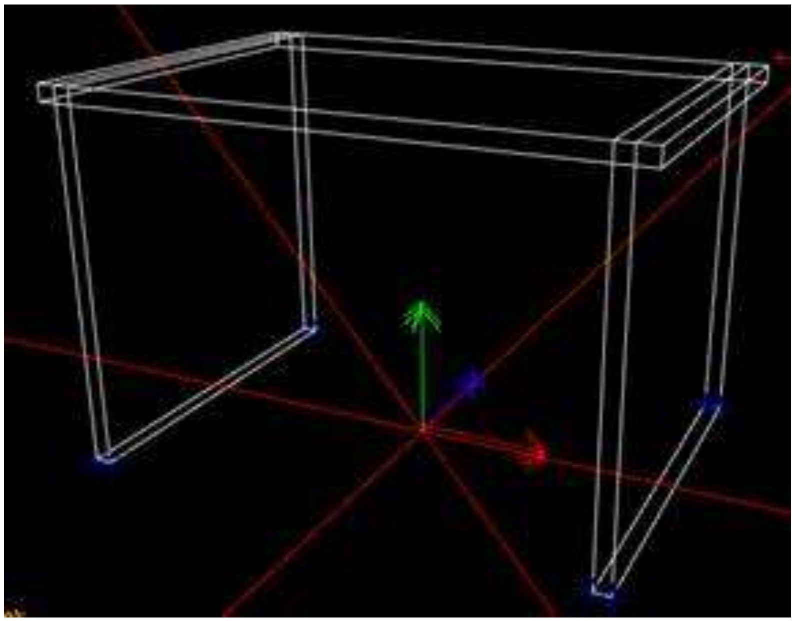
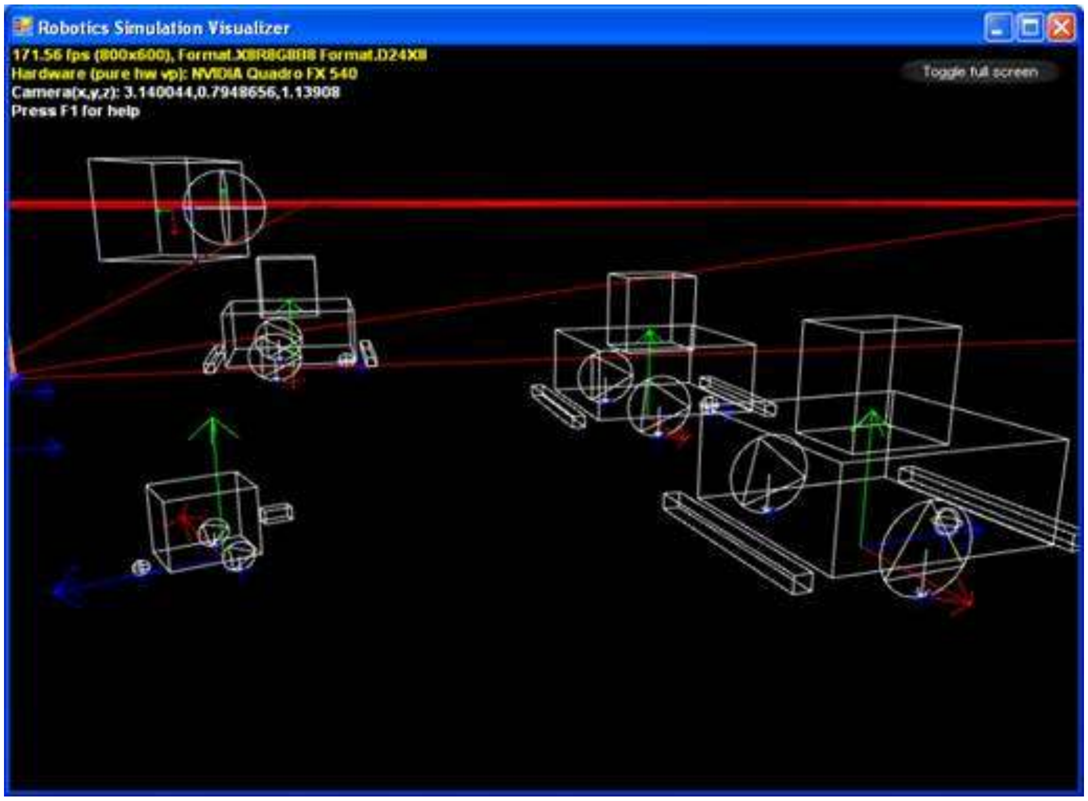


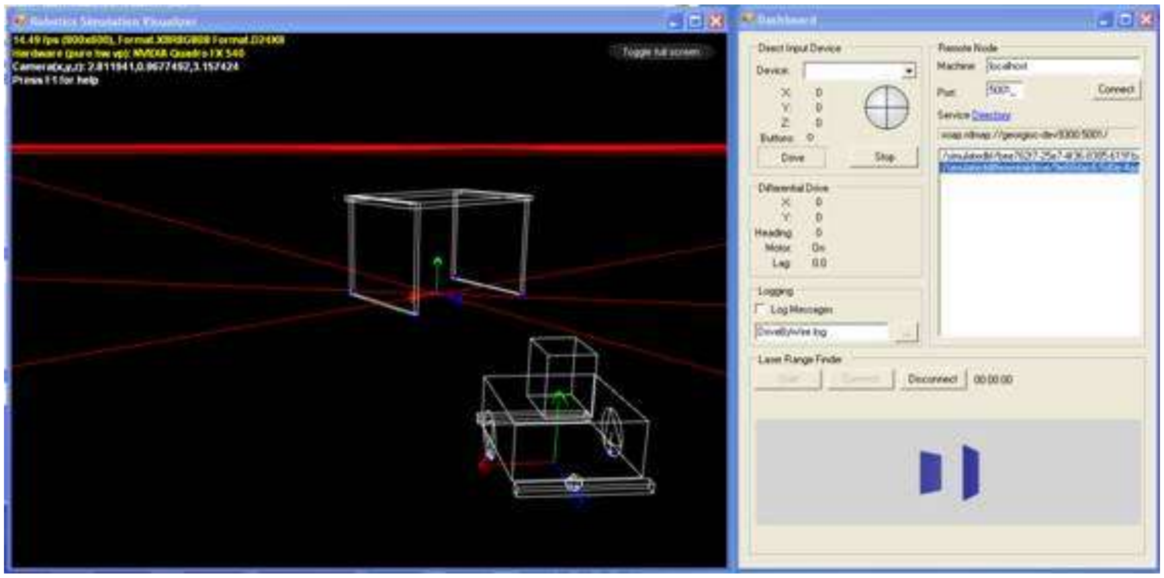
FIGURE 17:

The follow images are a simple Dashboard monitoring simulated laser in physics view and the last two images viewed a generic collection of any entity.



```
171.56 fps (800x600), Format.X8R8G8B8 Format.D24X8
Hardware (pure hw vp): NVIDIA Quadro FX 540
Camera(x,y,z): 3.140044,0.7948656,1.13908
Press F1 for help
```

FIGURE 18:



```
14.49 fps (800x600), Format.X8R8G8B8 Format.D24X8
Hardware (pure hw vp): NVIDIA Quadro FX 540
Camera(x,y,z): 2.811941,0.9677492,3.157424
Press F1 for help
```

Direct Input Device — X: 0, Y: 0, Z: 0, Buttons: 0

Remote Node — Machine: localhost, Port: 5001

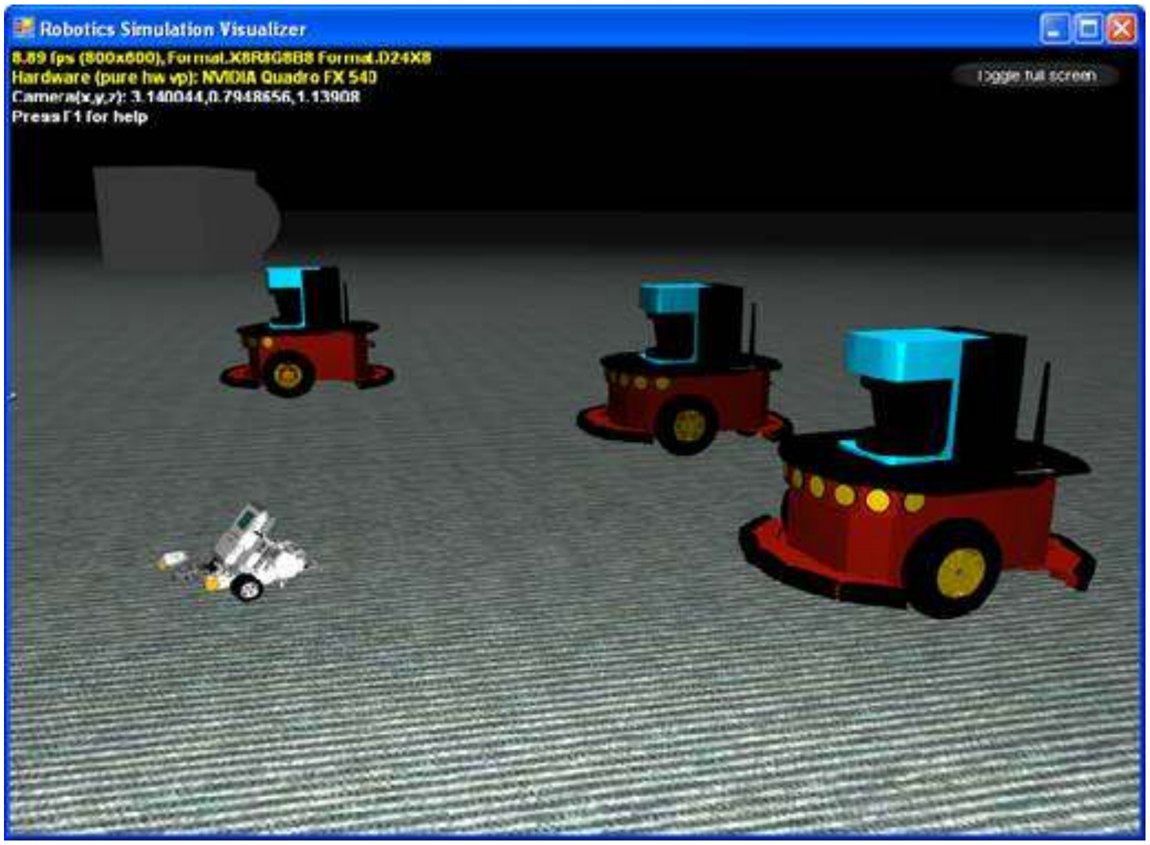
soap.tcp://georgioc-dev:5000 5001/

Differential Drive — X: 0, Y: 0, Heading: 0, Motor: On, Lag: 0.0

Logging — DriveByWire.log

Laser Range Finder — 00:00:00

FIGURE 19:



```
8.89 fps (800x600),Format.X8R8G8B8 Format.D24X8
Hardware (pure hw vp): NVIDIA Quadro FX 540
Camera(x,y,z): 3.140044,0.7948656,1.13908
Press F1 for help
```

FIGURE 20:

PART II

ROBOVIE-M VISUAL ENTITY

Chapter 4

Robovie-M Humanoid Robot

4.1. Introduction

Simulating and creating a robotic behavior has different important applications. Firstly, it allows researchers to develop and test programs in a virtual environment, without the physical presence of the robot. This may be extremely useful, for instance, in situation in which there are many people who are working on different tasks related to the same robot.

In this case, each person can work independently without interfering with the others. Moreover, simulators are very useful in education: Students can develop different algorithms to study the robots reaction, without been physically in the laboratory or without the risk to damage the robot.

Generally, a Humanoid Robot is a complex unit that is built with expensive electronic and mechanical parts.

4.2. Real Robovie-M Model

Robovie-M is a humanoid robot produced by the Japanese Vstone. We used the version three of this robot. It has 22 DOF (degrees of freedom), and consequently 22 servomotors, so distributed:

- 6 for each inferior art (legs)
- 4 for each superior art (arms)
- 2 for the trunk

Its dimensions are 290x240x65mm, with a complexive weight of 1.9 Kg.

The robot is distributed without the camera, and its control board does not support a camera device. We used the control board with the CPU Renesas SH2/7054, previously described. This is not only more powerful than the previous one, but it also allows a camera's supporting for image acquisitions. The control board's power supply is given by a set of batteries (five batteries of 1.2 V and 2300 mA) that gives 6 V as output, and that is able to provide a current of 6 A, necessary to run the 22 servomotors of the robot.

The virtual model of the Robovie-M robot has been developed using the program 3DStudio. We took the size and weight measures of each single part of the Robovie-M, including the servomotors.

Then, we have drawn the virtual model of the robot with 3DStudioMAX. In the following part there are same figures representing real Robovie-M model version 3:

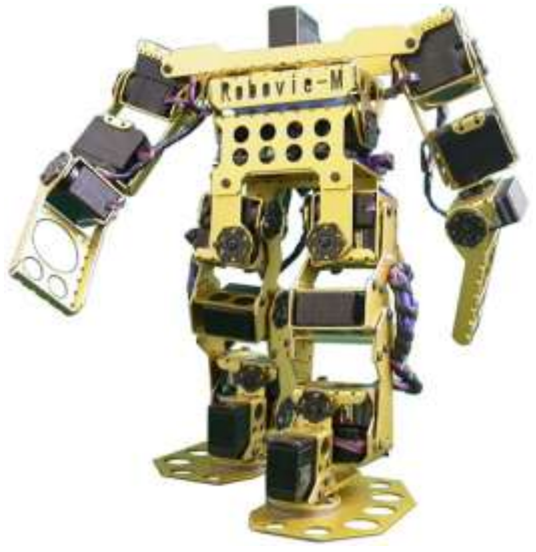


FIGURE 21:

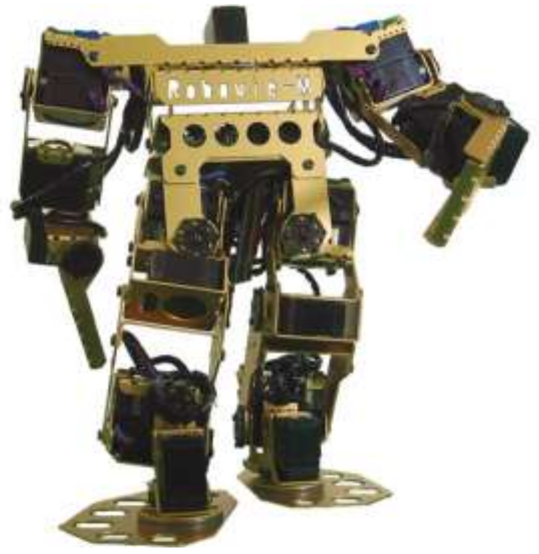


FIGURE 22:

Usually this humanoid robot model used to play soccer and other simulation sport. An example is represented in the following figure:

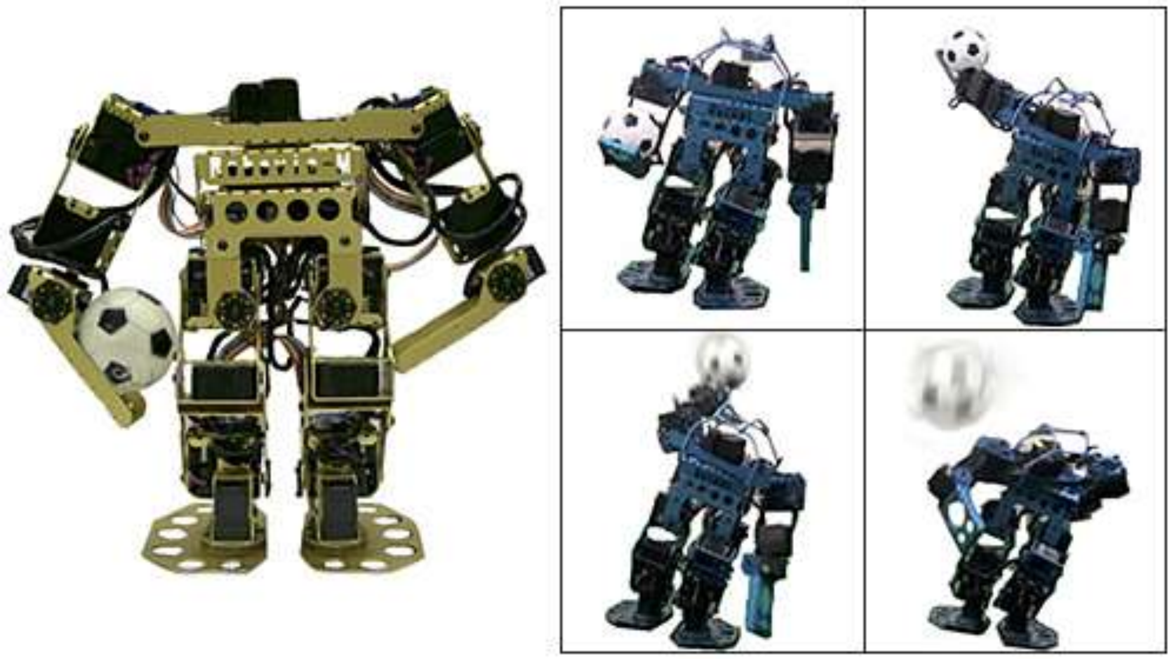


FIGURE 23:

On the right hand side of figure 23 the robot throws a ball with the right arm and on the left side of figure 23 the robot holds the ball in the right hand.

In the following part we will describe a virtual Robovie-M for Microsoft Robotics Studio implementation.

We will start to create a 3D model with 3DStudio MAX and in a second moment we will create a Robovie-M *VisualEntity* for inserting it into MsRS simulation, considering code's structure. We will view any parts of code used for implementation it. The parts will be selected with a *#region* and *#endregion* (C# keyword to group parts of code) inside the project's code.

4.3. Virtual Robovie-M VisualEntity for MsRS

When we want to insert an entity into Microsoft Robotics Studio simulation we must execute the following steps:

- Take a real model
- Create 3D model and its single components.
- Create a visual entity and approximate physics for each 3D model component
- Insert model on the simulation.

The following diagram-flow steps represent the concept just said:

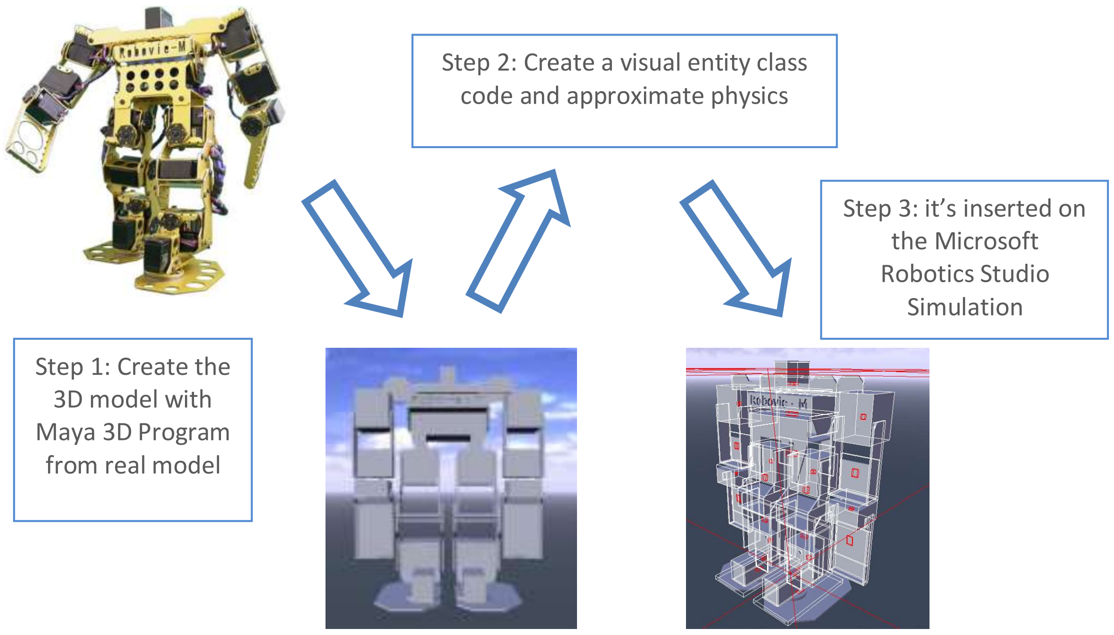

Step 2: Create a visual entity class code and approximate physics
Step 3: it's inserted on the Microsoft Robotics Studio Simulation
Step 1: Create the 3D model with Maya 3D Program from real model

FIGURE 24

Firstly *Robovie-M* implementations view a namespace structure and it's represented in the following code part:

```
using Microsoft.Robotics.Simulation.Engine;
using Microsoft.Robotics.Simulation.Engine.RobovieM.RobovieMAbstract;
using Microsoft.Robotics.Simulation.Engine.RobovieM.RobovieMBody;
using Microsoft.Robotics.Simulation.Engine.RobovieM.RobovieMArm;
using Microsoft.Robotics.Simulation.Engine.RobovieM.RobovieMLeg;
using Microsoft.Robotics.Simulation.Engine.RobovieM.RobovieServoMotors;
```

CODE 27

The generic using of the each source files are:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing.Design;
using System.IO;

using Microsoft.Dss.Core.Attributes;
using Microsoft.Robotics.PhysicalModel;
using Microsoft.Robotics.Simulation;

using physic = Microsoft.Robotics.Simulation.Physics;
using xna = Microsoft.Xna.Framework;
using xnag = Microsoft.Xna.Framework.Graphics;
```

CODE 28

When we want to insert a new entity into Microsoft Robotics Studio simulation, we must extend *VisualEntity* class. For explicit concept in the following part we view the most important visual entity class' information:

```
public class VisualEntity
   :Entity, IVisualEntity, IDisposable, ICloneable, IDssSerializable
{
  public VisualEntity();

  public VisualEntity Parent { get; set; }
  public virtual void AddShapeToPhysicsEntity(Shape, VisualEntityMesh);
  public override object Clone();
  public void CopyTo(IDssSerializable);
  protected void CreateAndInsertPhysicsEntity(PhysicsEngine);
  public virtual void Initialize(GraphicsDevice, PhysicsEngine);
  public virtual void InsertEntity(VisualEntity);
  public virtual void RemoveEntity(VisualEntity);
  public void Render(MatrixTransforms, CameraEntity);
  public virtual void Render(RenderMode, MatrixTransforms, CameraEntity);
  protected void Render(RenderMode, MatrixTransforms, VisualEntityMesh);
  public virtual void Update(FrameUpdate);
}
```

CODE 29

The *Robovie-M* 's project entity base has the following structure:

```
/// <summary>
/// RobovieVisualEntity, rapresent a abstract class to indentify a Robovie-M basic
structure
/// </summary>
[DataContract(Name = "RobovieVisualEntity")]
public abstract class RobovieVisualEntity : VisualEntity { }

/// <summary>
/// RobovieShapesEntity is a abstract class that defined a list of BoxShapes basic for
a Robovie-M internal component
/// </summary>
[DataContract(Name = "RobovieShapesEntity")]
public abstract class RobovieShapesEntity : RobovieVisualEntity { }

/// <summary>
/// RobovieArmVisualEntity is a abstract class to represent a VisualEntity..that have
got a two pose for connect other entity.
/// The pose are UpperPose and LowerPose.
/// </summary>
[DataContract(Name = "RobovieArmVisualEntity")]
public abstract class RobovieArmVisualEntity : RobovieShapesEntity { }
```

CODE 30

Where the components are:

***RobovieVisualEntity*,** represens the base class where we have defined the necessary components for visual entity implementation. It has all the variables to manage the visual entity's structure. It has a variable SCALE to estabilish the dimension in scale; and variable PATHMODEL that defineds a localpath including a 3D obj file model. We have defined all the variables that will create a visual entity's approximate physics, examples are: BORDER, Dimension, Depth, Weight, Height and BoundingBoxDimension that repesents a shape border, shape dimension, shape depth, shape weight, shape height and shape bounding box dimension respectively.

There are other variables to manage with axamples are: ROTATE_(x | y | z) representing a generic joint's rotation axis; NORMAL_(x | y | z) defined joint's normal and specificating 3D models's dimension for each submodel, examples are: DIMENSION_HEAD, DIMENSION_BELLY, DIMENSION_SHOULDER with model's mass specification with signature MASS_HEAD and MASS_SHOULDER etc.

There are functions defined for generic joint creation, an example is in the following code:

```
/// <summary>
/// Create a PhysicJoint instance with default Angular property
/// </summary>
/// <param name="jointname">Joint's name</param>
/// <param name="dof">Joint's Degrees of freedom</param>
/// <returns>PhysicJoint instance created</returns>
protected PhysicsJoint CreatePhysicsJoint(String jointname, JointDOFMode dof)
{
  PhysicsJoint jointInstance = null;
  JointAngularProperties commonAngular = new JointAngularProperties();

  commonAngular.TwistMode = dof;
  commonAngular.TwistDrive = new JointDriveProperties(
    JointDriveMode.Position,
      new SpringProperties(500000, 100000, 0), 1000000);
  jointInstance = CreatePhysicsJoint(jointname, dof, commonAngular);
  return jointInstance;
}
```

CODE 31

Where *jointname* is joint's name and *dof* is joint's degrees of freedom, it can be *Free*, *Limited* and *Locked* value. In this case the *dof* value for each joints are *JointDOFMode.Free* value. There are methods that connect two entities with specific details. It has two positions for connecting another entity and positions are *LowerPose* and *UpperPose* for each entity. An example code for connecting two entities is viewed in the following code:

```
/// <summary>
/// create a PhysicJoint to connect two entities where first is upper entity /// and
second is lower entity
/// </summary>
/// <param name="upperentity">RobovieArmVisualEntity upperentity</param>
/// <param name="lowerentity">RobovieArmVisualEntity lowerentity</param>
/// <param name="dof">Joint's degrees of freedom</param>
/// <param name="normal">Joint's normal</param>
/// <param name="axis">Joint's axis rotate</param>
/// <returns>Created instance of PhysicJoint to connect entities</returns>
protected PhysicsJoint ConnectorEntities(RobovieArmVisualEntity upperentity,
  RobovieArmVisualEntity lowerentity,
  JointDOFMode dof,
  Vector3 normal,
  Vector3 axis)
{
  upperentity.State.Pose.Orientation =
    TypeConversion.FromXNA(xna.Quaternion.CreateFromAxisAngle(
      new  xna.Vector3(axis.X, axis.Y, axis.Z), 0));
  lowerentity.State.Pose.Orientation =
    TypeConversion.FromXNA(xna.Quaternion.CreateFromAxisAngle(
       new xna.Vector3(axis.X, axis.Y, axis.Z), 0));

  PhysicsJoint pj =
    this.CreatePhysicsJoint(
      upperentity.State.Name + lowerentity.State.Name, dof);

  EntityJointConnector ejc0 = new EntityJointConnector(
    upperentity,
    normal,
    axis,
    upperentity.LowerPose.Position);
  EntityJointConnector ejc1 = new EntityJointConnector(
    lowerentity,
    normal,
    axis,
    lowerentity.UpperPose.Position);
  pj.State.Connectors[0] = ejc0;
  pj.State.Connectors[1] = ejc1;

  upperentity.LowerJoint = pj;
  lowerentity.UpperJoint = pj;

  return pj;
}
```

Code 32

The recent method is used for a specific entities connection. There are methods for other entities connection; method’s signature is viewed in the following code:

```
protected PhysicsJoint ConnectorEntities(
   RobovieHeadEntity upperentity,
   RobovieBreastEntity lowerentity,
   JointDOFMode dof,
   Vector3 normal,
   Vector3 axis);

protected PhysicsJoint ConnectorEntities(
   RobovieHeadEntity head,
   RobovieShoulderLeftEntity left,
   JointDOFMode dof,
   Vector3 normal,
   Vector3 axis);

protected PhysicsJoint ConnectorEntities(
   RobovieHeadEntity head,
   RobovieShoulderRightEntity right,
   JointDOFMode dof,
   Vector3 normal,
   Vector3 axis);

protected PhysicsJoint ConnectorEntities(
   RobovieBellyEntity upperentity,
   HipLeftEntity lowerentity,
   JointDOFMode dof,
   Vector3 normal,
   Vector3 axis);

protected PhysicsJoint ConnectorEntities(
   RobovieBellyEntity upperentity,
   HipRightEntity lowerentity,
   JointDOFMode dof,
   Vector3 normal,
   Vector3 axis);
```

Code 33

The signatures are used for the connection Head with Breast, Head with Shoulder Left, Head with Shoulder Right and Belly with Hip Left and Hip Right. The methods are created for a specific use where an entity has not only specific *LowerPose* and *UpperPose*, but it has another pose to connect joint; examples are: *LeftPose* and *RightPose*. We remember that *RobovieVisualEntity* has only two positions to connect joint; they are *LowerPose* and *UpperPose*.

RobovieShapeEntity, is used for to adding more information for a single entity or single model. It has a list of shapes that represent model's approximate physics on the Microsoft Robotics Studio simulation. It has same properties like central position that is model's epicenter and same methods to insert shape in to data structure. Class structure is in the following code:

```
/// <summary>
/// BoxShape's specific mass density
/// </summary>
protected float MASS = 0.00001f;

/// <summary>
/// Center pose 3D model
/// </summary>
protected Pose _center;

/// <summary>
/// List of BoxShapes to represent aproxmatily physic 3D model
/// </summary>
protected List<BoxShape> _boxshapes;

/// <summary>
/// Property, Get or Set a List of BoxShapes
/// </summary>
[DataMember]
[Description("List of BoxShapes for physic entity defined")]
public List<BoxShape> BoxShapes;

/// <summary>
/// Added list of BoxShapes in to PhysicPrimitives
/// </summary>
protected virtual void AddBoxShapesToEntity()
{
  if (this._boxshapes != null)
  {
    foreach (BoxShape bs in _boxshapes)
      base.State.PhysicsPrimitives.Add(bs);
  }
}

/// <summary>
/// abstract method to use for creating a aproximate phisys 3D model.
/// </summary>
protected abstract void CreatePhysicsShapes();
```

Code 34

RobovieArmVisualEntity, represents a base class for all entity's models. It has the same properties like: *LowerPose*, *LowerJoint*, *UpperPose* and *UpperJoint* used for connect two entities in a specific point. A diagram to represent this concept is:

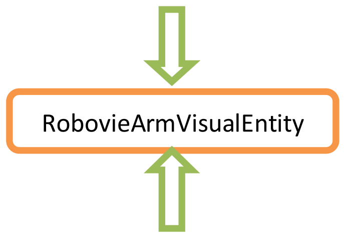


FIGURE 25

The class' structure is specific in the following part:

```
/// <summary>
/// Visible mesh[0] pose if it's exists
/// </summary>
private Pose _visibleMeshPose = new Pose();

/// <summary>
/// Property, Get a center pose of 3D model
/// </summary>
[DataMember]
[Description("Center Mesh pose")]
public Pose CenterMeshPose;

/// <summary>
/// Property, Get a UpperPose
/// </summary>
[DataMember]
[Description("Upper Joint pose")]
public Pose UpperPose;

/// <summary>
/// Property, Get a LowerPose
/// </summary>
[DataMember]
[Description("Lower Joint pose")]
public Pose LowerPose;

/// <summary>
/// Property, Get or Set UpperJoint
/// </summary>
[DataMember]
[Description("Upper Joint")]
public Joint UpperJoint;

/// <summary>
/// Property, Get or Set LowerJoint
/// </summary>
[DataMember]
[Description("Lower Joint")]
public Joint LowerJoint;

/// <summary>
/// Property, Get or Set ServoMotor in to VisualEntity
/// </summary>
[DataMember]
```

```
[Description("Robovie ServoMotor Visual Entity")]
public RobovieServoMotorEntity ServoMotor;

/// <summary>
/// Default Constructor
/// </summary>
protected RobovieArmVisualEntity();
protected RobovieArmVisualEntity(String name, String mesh, Vector3 initPose);

/// <summary>
/// override Initialize VisualEntity. Call a base initialize and calculate model's
center, create physic, create pose to joint
/// and added physic in to visual entity
/// </summary>
/// <param name="device">GraphicsDevice</param>
/// <param name="physicsEngine">PhysicsEngine</param>
public override void Initialize(
   Microsoft.Xna.Framework.Graphics.GraphicsDevice device,
   PhysicsEngine physicsEngine);

/// <summary>
///
/// </summary>
/// <param name="renderMode">render mode</param>
/// <param name="transforms">transforms</param>
/// <param name="currentCamera">current camera</param>
public override void Render(
   VisualEntity.RenderMode renderMode,
   MatrixTransforms transforms,
   CameraEntity currentCamera);

/// <summary>
/// abstract method to define the joint's position in the visual entity
/// </summary>
protected abstract void CreatePoseToJoints();
```

Code 35

When we want to create an object of this type, we need: a name, a mesh's path and the initial position.

The *RobovieArmVisualEntity*'s constructor is defined here:

```
protected RobovieArmVisualEntity(String name, String mesh, Vector3 initPose)
{
   this.State.Name = name;
   this.State.Assets.Mesh = mesh;
   this._visibleMeshPose = new Pose(initPose);
   this._boxshapes = new List<BoxShape>();
}
```

Code 36

It's protected because this class is not directly instance, but it has to be extended by another class for instancing an object of this type. For each entity the same implementation override method is used to initialize the entity. The following code is an example:

```
public override void Initialize(
   Microsoft.Xna.Framework.Graphics.GraphicsDevice device,
   PhysicsEngine physicsEngine)
{
  base.Initialize(device, physicsEngine);

  if (BoxShapes.Count == 0)
  {
    float x = base.EntityBoundingSphere.Center.X;
    float y = base.EntityBoundingSphere.Center.Y;
    float z = base.EntityBoundingSphere.Center.Z;

    this._center = new Pose(new Vector3(x, y, z));

    this.CreatePhysicsShapes();
    this.CreatePoseToJoints();
    this.AddBoxShapesToEntity();

    if (base.State.PhysicsPrimitives.Count > 0)
      base.CreateAndInsertPhysicsEntity(physicsEngine);
  }
}
```

CODE 37

When shapes are not defined in to entities (model approximate physics), in the specific case when *BoxShapes.Count* is 0, steps are the following: determinate mesh center pose, creating an approximate physics, and create a position to insert same joints. The last step is to insert all shapes in to visual entity parent. The Render method allows mesh rendering in to Microsoft Robotics Studio simulation. The simple code is shown below:

```
public override void Render(
   VisualEntity.RenderMode renderMode,
   MatrixTransforms transforms,
   CameraEntity currentCamera)
{
  transforms.World = World;
  base.Render(renderMode, transforms, Meshes[0]);
}
```

CODE 38

In the next section we will to describe the classes structure and we will specific all signatures it. In following image are class structures specific:

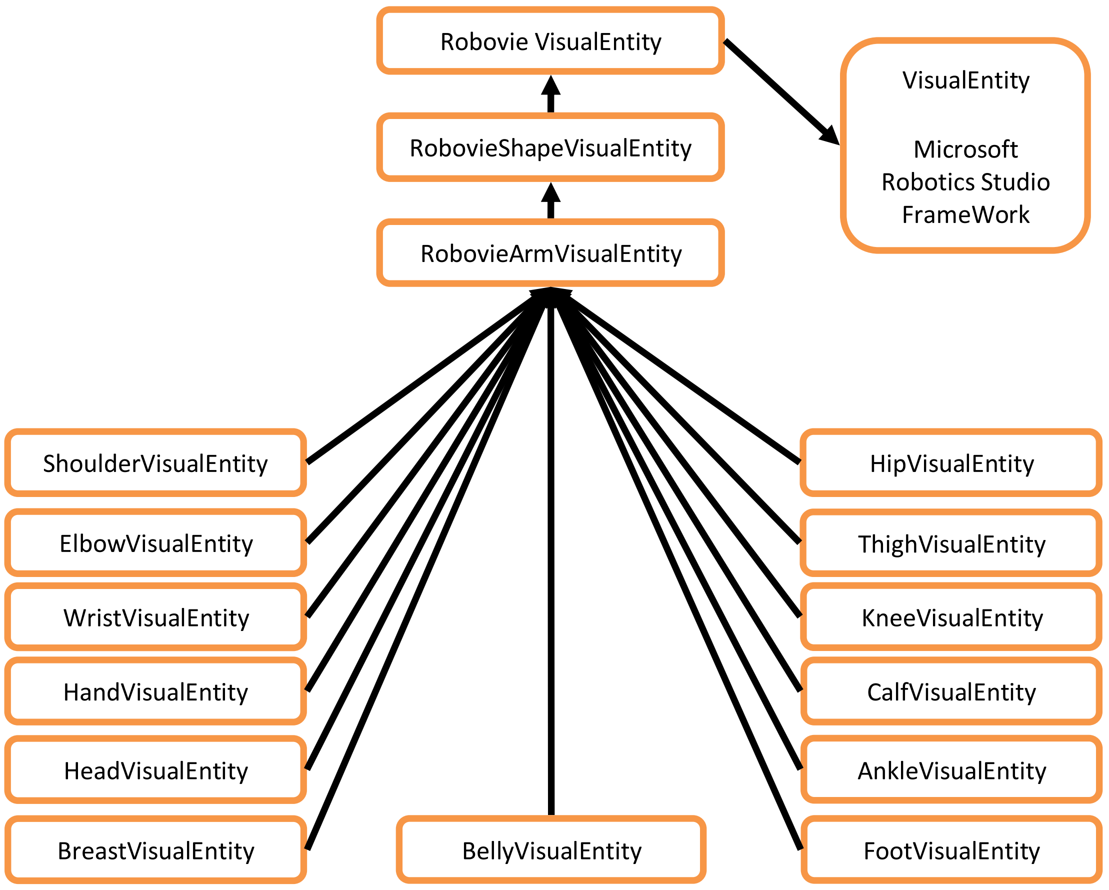


Figure 26

The image shown enters in the specific details for parts that have Left and right visual entity implementations. The parts are: shoulders, elbows, wrists, hands, hips, thighs, knees calves, ankles and feet entities. The following image represents a generic structure of Visual Entity type mode. We take as example *XXXVisualEntity* and *XXXLeftVisualEntity* and *XXXRightVisualEntity*. This concept is valid for each type of entities.

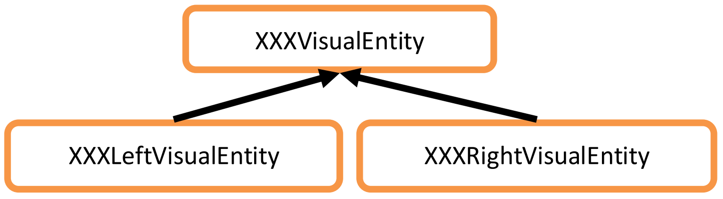


Figure 27

The substring *XXX* in the preview image can be: Shoulders, Elbows, Wrists, Hands for robovie-m’s arms and Hips, Thighs, Knees, Calves, Ankles and Feet for robovie-m’s legs.

We will now treat in more detailed way single entities, starting from *HeadVisualEntity* up to *FootRightVisualEntity*. We don't cut & paste from project's code, but we will illustrate an important code's part.

4.4. Robovie Body Visual Entities

In this section we will treat the part build *Robovie-M*'s body. It contains the following visual entities: *RobovieHeadVisualEntity*, *RobovieBreastVisualEntity* and *RobovieBellyVisualEntity*. For all entities we will to describe the important part the composition it.

RobovieHeadVisualEntity

This entity represents a Visual Entity to robovie's head. Class' structure is defined in to following part of the code:

```
[DataContract(Name = "RobovieHeadEntity")]
public class RobovieHeadEntity : RobovieArmVisualEntity // Testa
{
   //body
}
```

CODE 39

It's different from the other entities but it is possible to connect an entity on *LeftPose* and *RightPose* in to it for another two joint. In the following code there are same head's properties.

```
/// <summary>
/// Property, Get Joint's Left Pose
/// </summary>
public Pose LeftPose;

/// <summary>
/// Property, Get Joint's Right Pose
/// </summary>
public Pose RightPose;

/// <summary>
/// Property, Get or Set Head's joint left
/// </summary>
public Joint LeftJoint;

/// <summary>
/// Property, Get or Set Head's joint right
/// </summary>
public Joint RightJoint;

/// <summary>
/// Constructor
/// </summary>
/// <param name="initPose">Head Visual Entity's initial position</param>
public RobovieHeadEntity(Vector3 initPose)
  : base("HeadEntity", PATHMODEL + @"Head.obj", initPose)
{
   this._dimension = base.DIMENSION_HEAD;
}
```

CODE 40

For this entity we just treat only two methods *CreatePhysicShapes()*,creating an approximate physics with shape definition for the specific entity and *CreatePoseToJoint()* method creating the positions to connect the other entity.

Usually positions of course are *LowerPose*, *UpperPose* and in the same case, an example is head entity, *LeftPose* and *RightPose*. The *CreatePhysicShapes()* code's part is:

```
protected override void CreatePhysicsShapes()
{
Pose center = base._center;

BoxShape centerbox = new BoxShape(
  new BoxShapeProperties(
    base.MASS, center,
    new Vector3(this.Depth * 2 - BORDER * 6, this.Height * 2 - BORDER, BORDER * 10)));

Pose goundpose = new Pose(
   center.Position + new Vector3(0, -this.Height + BORDER, 0));

BoxShape groundbox = new BoxShape(new BoxShapeProperties(
   base.MASS, goundpose, new Vector3(this.Depth * 2 - BORDER, BORDER, this.Weight)));

Pose behindpose = new Pose(
   center.Position + new Vector3(-this.Depth + (BORDER / 2), -this.Height / 2, 0));

BoxShape behindbox = new BoxShape(new BoxShapeProperties(
   base.MASS, behindpose, new Vector3(BORDER, this.Height, this.Weight * 2)));

Pose beorepose = new Pose(
   center.Position + new Vector3(+this.Depth - (BORDER / 2), -this.Height / 2, 0));

BoxShape beforebox = new BoxShape(new BoxShapeProperties(
   base.MASS, beorepose, new Vector3(BORDER, this.Height, this.Weight * 2)));

base.BoxShapes.Add(centerbox);
base.BoxShapes.Add(groundbox);
base.BoxShapes.Add(behindbox);
base.BoxShapes.Add(beforebox);

base.State.MassDensity.Mass = base.MASS_HEAD;
base.State.MassDensity.CenterOfMass = base.CenterMeshPose;
}
```

CODE 41

The method creating a list of Shape and relative position and insert it in the list of shapes. The shapes will load in the base *VisualEntity* when we write *base.BoxShapes.add(shape)*.

The *CreatePoseToJoints()* method code's part is defined below:

```
protected override void CreatePoseToJoints()
{
Pose center = base._center;

this._upperpose = new Pose(
   center.Position + new Vector3(0, this.Height, 0));

this._rightpose = new Pose(
   center.Position + new Vector3(0, -this.Height + BORDER, -(this.Weight - BORDER)));

this._leftpose = new Pose(
   center.Position + new Vector3(0, -this.Height + BORDER, (this.Weight - BORDER)));

this._lowerpose = new Pose(
   center.Position + new Vector3(0, -this.Height, 0));
}
```

CODE 42

The method creates points to connect the other entities and a specific position for the joints. The entity's diagram is defined here:

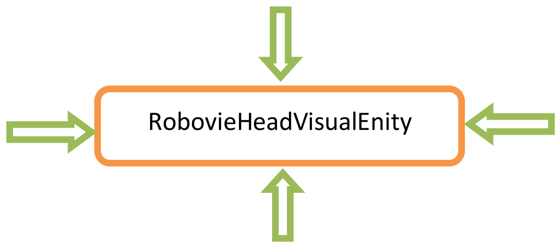


FIGURE 28

RobovieBreastEntity

The entity represents a robovie's breast and its definition code is:

```
public class RobovieBreastEntity : RobovieArmVisualEntity //Petto
{
  /// <summary>
  ///
  /// </summary>
  /// <param name="initPose"></param>
  public RobovieBreastEntity(Vector3 initPose)
    : base("BreastEntity", PATHMODEL + @"Breast.obj", initPose)
  {
    base._dimension = base.DIMENSION_BREAST;
  }
/// <summary>
///
/// </summary>
protected override void CreatePhysicsShapes()
{
  Pose center = base._center;

  Pose centerpose = new Pose(center.Position + new Vector3(0, this.Height / 2, 0));
  BoxShape centerbox = new BoxShape(new BoxShapeProperties(
    base.MASS,
    centerpose,
    new Vector3(this.BoundingBoxDimension.X, this.BoundingBoxDimension.Y / 2,
this.BoundingBoxDimension.Z)));

  Pose leftpose = new Pose(new Vector3(
    center.Position.X,
    center.Position.Y - this.Height / 2,
    center.Position.Z - (this.Weight - BORDER / 2)));

  BoxShape leftbox = new BoxShape(new BoxShapeProperties(
    base.MASS,
    leftpose,
    new Vector3(this.Depth * 2, this.Height, BORDER)));

  Pose rightpose = new Pose(new Vector3(
    center.Position.X,
    center.Position.Y - this.Height / 2,
    center.Position.Z + (this.Weight - BORDER / 2)));

  BoxShape rightbox = new BoxShape(new BoxShapeProperties(
    base.MASS,
    rightpose,
    new Vector3(this.Depth * 2, this.Height, BORDER)));

  base.BoxShapes.Add(centerbox);
  base.BoxShapes.Add(leftbox);
  base.BoxShapes.Add(rightbox);

  base.State.MassDensity.Mass = base.MASS_BREAST;
  base.State.MassDensity.CenterOfMass = base.CenterMeshPose;
}

/// <summary>
///
/// </summary>
protected override void CreatePoseToJoints()
{
  Pose center = base._center;

  this._upperpose = new Pose(
    center.Position + new Vector3(0, this.Height, 0));

  this._lowerpose = new Pose(
    center.Position + new Vector3(0, -this.Height + this.Height / 4, 0));
}
}//end class
```

CODE 43

The breast entity defines a default property, *LowerPose* and *UpperPose* and defined three shapes that represent an approximate physics for this entity. The logic breast's diagram is shown in the following part:

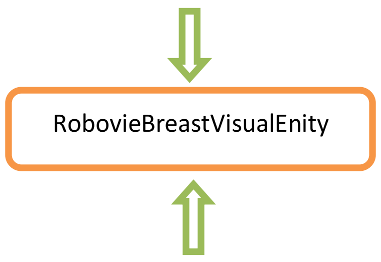


FIGURE 29

RobovieBellyVisualEntity

The entity represents a robovie's belly and its definition code is:

```
[DataContract(Name = "RobovieBellyEntity")]
public class RobovieBellyEntity : RobovieArmVisualEntity
{
   public Pose LowerLeftPose { get { return _lowerleftpose; } }
   public Pose LowerRightPose { get { return _lowerrightpose; } }

   public Joint LowerLeftJoint
   {
     get { return _lowerleftjoint; }
     set { this._lowerleftjoint = value; }
   }

   public Joint LowerRightJoint
   {
     get { return _lowerrightjoint; }
     set { this._lowerrightjoint = value; }
    }
   // <summary>
   /// Default Constructor
   /// </summary>
   /// <param name="initPose"></param>
   public RobovieBellyEntity(Vector3 initPose)
     : base("BellyEntity", PATHMODEL + @"Belly.obj", initPose)
   {
      base._dimension = base.DIMENSION_BELLY;
   }

   protected override void CreatePhysicsShapes()
   {
     Pose center = base._center;

     #region CenterBox Definition Code
     #region BehindLeftBox Definition Code
     #region BehindRightBox Definition Code
     #region BeforeLeftBox Definition Code
```

```
      #region BeforeRightBox Definition Code

      base.BoxShapes.Add(centerbox);
      base.BoxShapes.Add(behindleftbox);
      base.BoxShapes.Add(behindrighbox);
      base.BoxShapes.Add(beforeleftbox);
      base.BoxShapes.Add(beforerightbox);

      base.State.MassDensity.Mass = base.MASS_BELLY;
      base.State.MassDensity.CenterOfMass = base.CenterMeshPose;
    }

    protected override void CreatePoseToJoints()
    {
       Pose center = base._center;

       this._upperpose = new Pose(
          center.Position + new Vector3(0, (this.Height * 2 / 3), 0));

       this._lowerleftpose = new Pose(
          center.Position + new Vector3(0, -this.Height / 2 - this.Height / 4,
(this.Weight - BORDER * 6)));

       this._lowerrightpose = new Pose(
          center.Position + new Vector3(0, -this.Height / 2 - this.Height / 4, -
(this.Weight - BORDER * 6)));
   }
}
```

CODE 44

The belly entity defines a default property, *UpperPose* but defines two *LowerPose* type as well: *LowerLeftPose* to connect the left leg and *LowerRightPose* to connect the right leg. It defines five shapes that represent an approximate physics for this entity. The logic belly's diagram is shown in the following part:

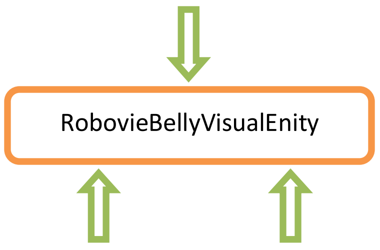


FIGURE 30

The three entities described are connected as specified in the following diagram:

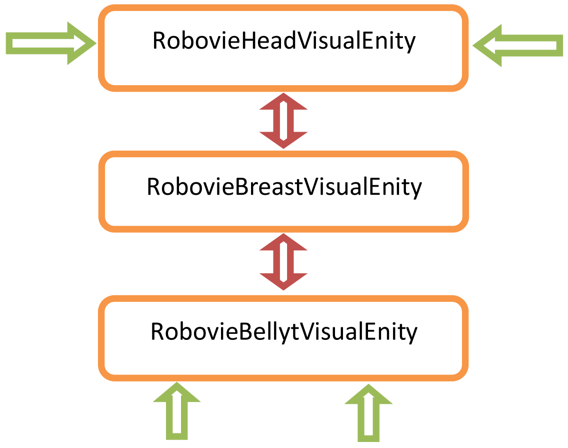


FIGURE 31

4.5. Robovie Arms Visual Entity

A single arm is composed by four entities and they are: Shoulder, Elbow, Wrist and Hand. There are two arms:

- *ArmLeft*, composed by ShoulderLeftVisualEntity, ElbowLeftVisualEntity, WristLeftVisualEntity and HandLeftVisualEntity.
- *ArmRight*, composed by ShoulderRightVisualEntity, ElbowRightVisualEntity, WristRightVisualEntity and HandRightVisualEntity.

Shoulder Left and Right VisualEntity

The following code represents the shoulder's structure. In general the shoulder has a lower pose for lower joint connection with other entities and that shoulder left has right pose and right joint for connection with other shoulder that has left pose and left joint connection. The shoulder structure is represented in following part code:

```
/// <summary>
/// ShoulderEntity to represent a generic abstract shoulder component
/// </summary>
[DataContract(Name = "ShoulderEntity")]
public abstract class ShoulderEntity : RobovieArmVisualEntity { … }

/// <summary>
/// ShoulderLeftEntity to represent a abstract left shoulder.
/// </summary>
[DataContract(Name = "ShoulderLeftEntity")]
public abstract class ShoulderLeftEntity : ShoulderEntity { … }

/// <summary>
/// ShoulderRightEntity to represent a abstract right shoulder.
/// </summary>
[DataContract(Name = "ShoulderRightEntity")]
public abstract class ShoulderRightEntity : ShoulderEntity { … }

/// <summary>
/// RobovieShoulderLeftEntity is a Robivie-M's specific class to represent a
shoulderleft entity.
/// </summary>
[DataContract(Name = "RobovieShoulderLeftEntity")]
public class RobovieShoulderLeftEntity : ShoulderLeftEntity { … }

/// <summary>
/// RobovieShoulderRightEntity is a Robivie-M's specific class to represent a
ShoulderRight entity.
/// </summary>
[DataContract(Name = "RobovieShoulderRightEntity")]
public class RobovieShoulderRightEntity : ShoulderRightEntity { … }
```

CODE 45

The shoulders' diagrams are shown in the following image. It views all the points by which other entities can be connected by shoulder.

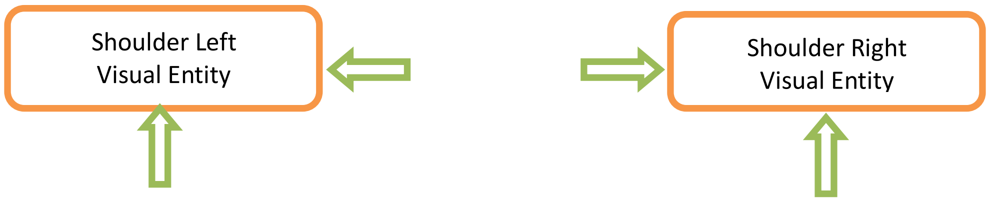


FIGURE 32

Robovie Elbow Left and Right VisualEntity

Robovie elbows can only have Robovie Arm Visual Entity properties, it has an upper pose and upper joint connection with another entities. The elbows' structure is shown in the following part:

```
/// <summary>
/// ElbowEntity abstract class
/// </summary>
[DataContract(Name = "ElbowEntity")]
public abstract class ElbowEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieElbowLeftEntity is a Robovie-M's specific class to represent a ElbowLeft
/// </summary>
[DataContract(Name = "RobovieElbowLeftEntity")]
public class RobovieElbowLeftEntity : ElbowEntity { … }

/// <summary>
/// RobovieElbowRightEntity is a Robovie-M's specific class to represent a ElbowRight
/// </summary>
```

```
[DataContract(Name = "RobovieElbowRightEntity")]
public class RobovieElbowRightEntity : ElbowEntity { … }
```

CODE 46

The elbows' diagrams are shown in the following image. It views all points by which other entities can be connected by elbow

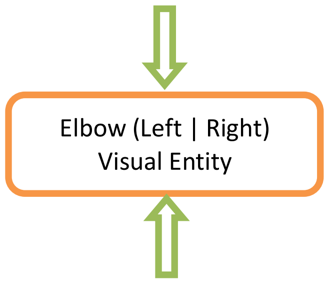


FIGURE 33

Robovie Wrist Left and Right VisualEntity

Robovie wrists can only have Robovie Arm Visual Entity properties, it has an upper pose and upper joint connection with other entities. The wrists' structure is shown in the following part:

```
/// <summary>
/// WristEntity a abstract class
/// </summary>
[DataContract(Name = "WristEntity")]
public abstract class WristEntity :  RobovieArmVisualEntity { … }

/// <summary>
/// RobovieWristLeftEntity to represent a Robivie-M's wristleft
/// </summary>
[DataContract(Name = "RobovieWristLeftEntity")]
public class RobovieWristLeftEntity : WristEntity { … }

/// <summary>
/// RobovieWristRightEntity to represent a Robivie-M's wristright
/// </summary>
[DataContract(Name = "RobovieWristRightEntity")]
public class RobovieWristRightEntity : WristEntity { … }
```

CODE 47

The wrists' diagrams are shown in the following image. It views all points by which other entities can be connected by wrist.

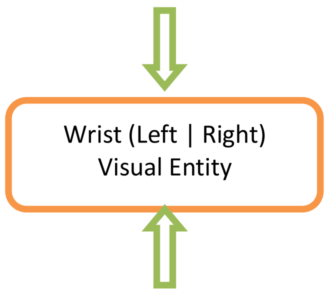


FIGURE 34

Robovie Hand Left and Right VisualEntity

Robovie hands can only have Robovie Arm Visual Entity properties, it has an upper pose and upper joint connection with other entities. The hands' structure is viewed in the following part:

```
/// <summary>
/// HandEntity represents a abstract class Hand
/// </summary>
[DataContract(Name = "HandEntity")]
public abstract class HandEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieHandLeftEntity represents a Robovie-M's handleft
/// </summary>
[DataContract(Name = "RobovieHandLeftEntity")]
public class RobovieHandLeftEntity : HandEntity { … }

/// <summary>
/// RobovieHandRightEntity represents a Robovie-M's handright
/// </summary>
[DataContract(Name = "RobovieHandRightEntity")]
public class RobovieHandRightEntity : HandEntity { … }
```

CODE 48

The hands' diagrams are viewed in the following image. It views all the points by which other entities can it connected by hand.

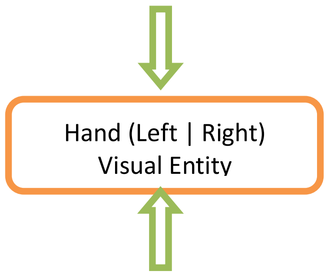
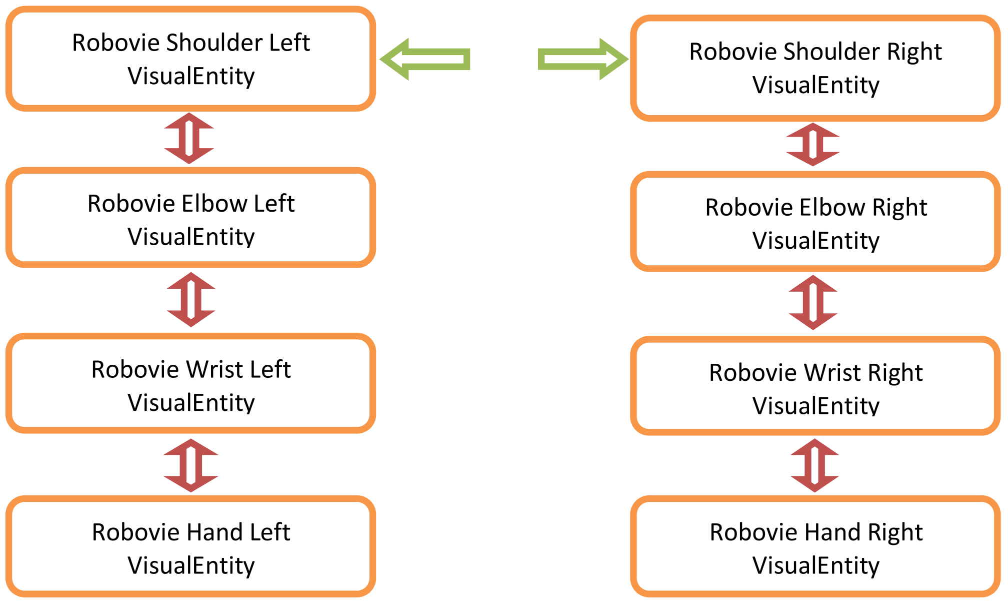


FIGURE 35

In the following picture shown how the entities are connected:



Robovie Shoulder Left VisualEntity
Robovie Shoulder Right VisualEntity
Robovie Elbow Left VisualEntity
Robovie Elbow Right VisualEntity
Robovie Wrist Left VisualEntity
Robovie Wrist Right VisualEntity
Robovie Hand Left VisualEntity
Robovie Hand Right VisualEntity

FIGURE 36

4.6. Robovie Legs Visual Entity

A single leg is composed by six entities that are: Hip, Thigh, Knee Calf, Ankle and Foot. There are two legs:

- *Leg Left*, composed by *HipLeftVisualEntity*, *ThighLeftVisualEntity*, *KneeLeftVisualEntity* and *CalfLeftVisualEntity*, *AnkleLeftVisualEntity* and *FootLeftVisualEntity*.
- *Leg Right*, composed by *HipRightVisualEntity*, *ThighRightVisualEntity*, *KneeRightVisualEntity* and *CalfRightVisualEntity*, *AnkleRightVisualEntity* and *FootRightVisualEntity*.

In this section we will describe legs' structure. The structure for all single component that compose a leg, is Robovie Arm Visual Entity structure. Now we illustrate the single component's structure and for each one we will view a diagram for connectivity with other entities.

RobovieHipVisualEntity

Hip's structure is explicit in the following code:

```
/// <summary>
/// abstract class HipEntity represents a Robovie-M's Hip
/// </summary>
[DataContract(Name = "HipEntity")]
public abstract class HipEntity : RobovieArmVisualEntity { …
}

/// <summary>
/// HipLeftEntity is abstract class represents a Robovie-M hip left
/// </summary>
[DataContract(Name = "HipLeftEntity")]
public abstract class HipLeftEntity : HipEntity { … }

/// <summary>
/// HipRightEntity is abstract class represents a Robovie-M hip right
/// </summary>
[DataContract(Name = "HipRightEntity")]
public abstract class HipRightEntity : HipEntity { … }

/// <summary>
/// RobovieHipLeftEntity is a specificy of Robovie-M's hip left
/// </summary>
[DataContract(Name = "RobovieHipLeftEntity")]
public class RobovieHipLeftEntity : HipLeftEntity { … }

/// <summary>
/// RobovieHipRightEntity is a specificy of Robovie-M's hip right
/// </summary>
[DataContract(Name = "RobovieHipRightEntity")]
public class RobovieHipRightEntity : HipRightEntity { … }
```

CODE 49

The hip's diagram by which it can be connected with another entity is:

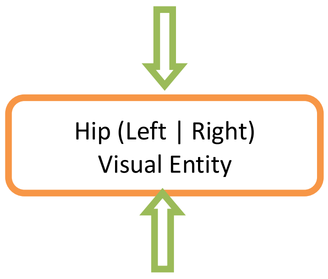


FIGURE 37

RobovieThighVisualEntity

Thigh's structure is explicit in the following code:

```
/// <summary>
/// ThighEntity is a abstract class
/// </summary>
[DataContract(Name = "ThighEntity")]
public abstract class ThighEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieThighLeftEntity is a class represents a Robovie-M's thigh left
/// </summary>
[DataContract(Name = "RobovieThighLeftEntity")]
public class RobovieThighLeftEntity : ThighEntity { … }

/// <summary>
/// RobovieThighRightEntity is a class represents a Robovie-M's thigh right
/// </summary>
[DataContract(Name = "RobovieThighRightEntity")]
public class RobovieThighRightEntity : ThighEntity { … }
```

CODE 50

The thigh's diagram by which it can be connected with another entity is:

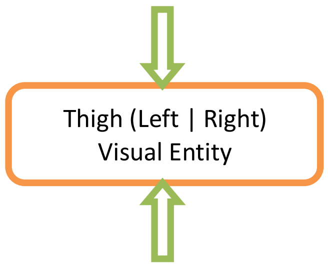


FIGURE 38

RobovieKneeVisualEntity

The knee's structure is explicit in the following code:

```
/// <summary>
/// KneeEntity is abstract class
/// </summary>
[DataContract(Name = "KneeEntity")]
public abstract class KneeEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieKneeLeftEntity is class represents a Robovie-M's Knee left
/// </summary>
[DataContract(Name = "RobovieKneeLeftEntity")]
public class RobovieKneeLeftEntity : KneeEntity { … }

/// <summary>
/// RobovieKneeRightEntity is class represents a Robovie-M's Knee right
/// </summary>
[DataContract(Name = "RobovieKneeRightEntity")]
public class RobovieKneeRightEntity : KneeEntity { … }
```

CODE 51

The knee's diagram by which it can be connected with another entity is:

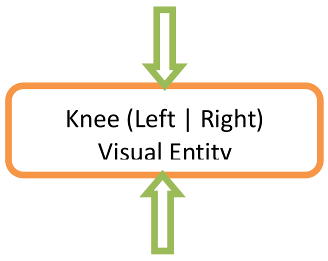


FIGURE 39

RobovieCalfVisualEntity

The calf's structure is explicit in the following code:

```
/// <summary>
/// CalfEntity is abstract class represents a generic Calf
/// </summary>
[DataContract(Name = "CalfEntity")]
public abstract class CalfEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieCalfLeftEntity is a class represents Robovie-M's Calf left
/// </summary>
[DataContract(Name = "RobovieCalfLeftEntity")]
public class RobovieCalfLeftEntity : CalfEntity { … }
```

```
/// <summary>
/// RobovieCalfRightEntity is a class represents Robovie-M's Calf right
/// </summary>
[DataContract(Name = "RobovieCalfRightEntity")]
public class RobovieCalfRightEntity : CalfEntity { … }
```

CODE 52

The calf’s diagram by which it can be connected with another entity is:

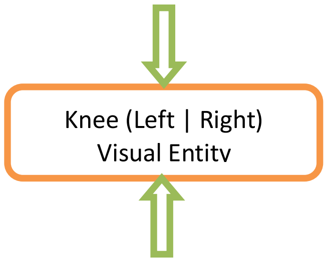


FIGURE 40

RobovieAnkleVisualEntity

The ankle’s structure is explicit in the following code:

```
/// <summary>
/// AnkleEntity is abstract class represents a generic Ankle.
/// </summary>
[DataContract(Name = "AnkleEntity")]
public abstract class AnkleEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieAnkleLeftEntity is class represents Robovie-M's ankle left
/// </summary>
[DataContract(Name = "RobovieAnkleLeftEntity")]
public class RobovieAnkleLeftEntity : AnkleEntit { … }

/// <summary>
/// RobovieAnkleRightEntity is class represents Robovie-M's ankle right
/// </summary>
[DataContract(Name = "RobovieAnkleRightEntity")]
public class RobovieAnkleRightEntity : AnkleEntity { … }
```

CODE 53

The ankle's diagram by which it can be connected with another entity is:

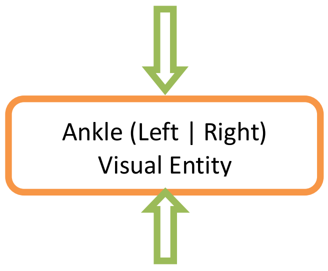


FIGURE 41

RobovieFootVisualEntity

The foot's structure is explicit in the following code:

```
/// <summary>
/// FootEntity is abstract class represents a generic Foot entity
/// </summary>
[DataContract(Name = "FootEntity")]
public abstract class FootEntity : RobovieArmVisualEntity { … }

/// <summary>
/// RobovieFootLeftEntity is a class represents Robovie-M's foot left
/// </summary>
[DataContract(Name = "RobovieFootLeftEntity")]
public class RobovieFootLeftEntity : FootEntity { … }

/// <summary>
/// RobovieFootRightEntity is a class represents Robovie-M's foot right
/// </summary>
[DataContract(Name = "RobovieFootRightEntity")]
public class RobovieFootRightEntity : FootEntity { … }
```

CODE 54

The foot's diagram by which it can be connected with another entity is:

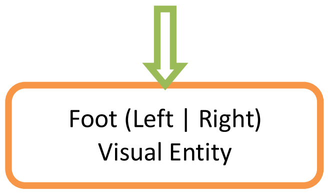


FIGURE 42

The following diagram represents legs' structure and the connection between entities. The Hip entities are free and they will be connected at the Belly entity.

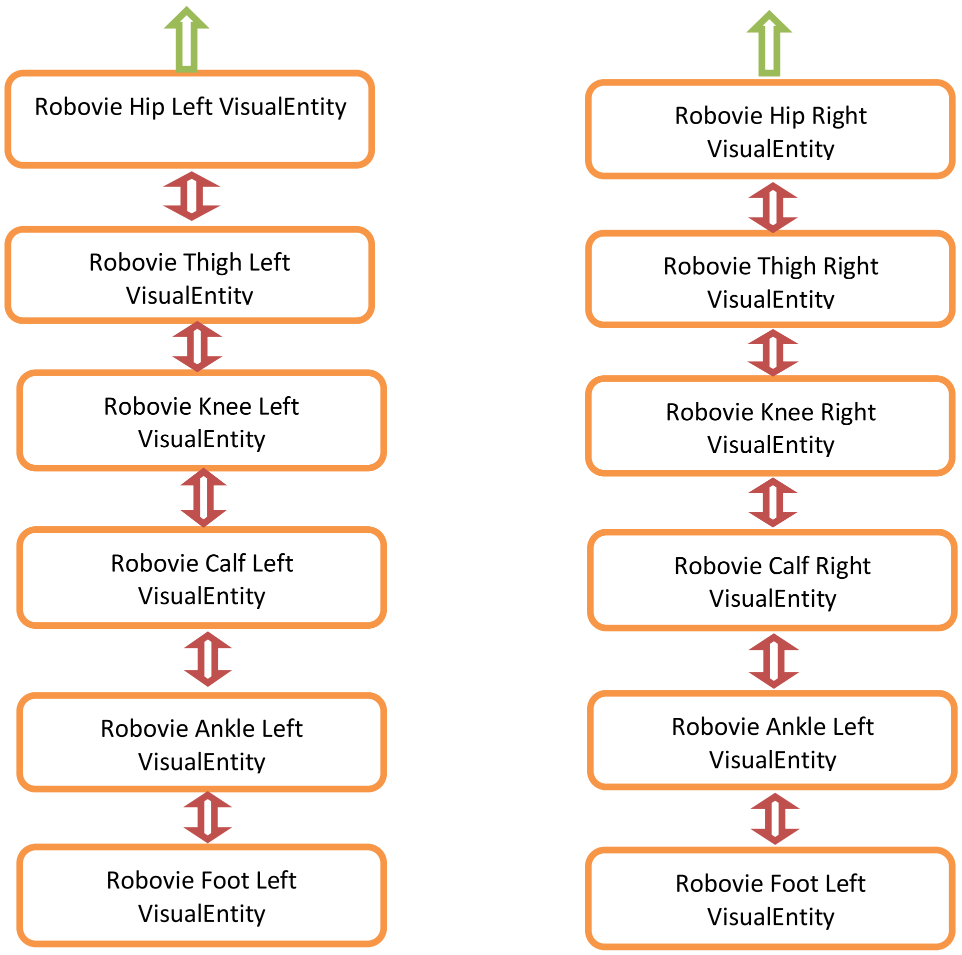

Robovie Hip Left VisualEntity
Robovie Thigh Left VisualEntitv
Robovie Knee Left VisualEntity
Robovie Calf Left VisualEntity
Robovie Ankle Left VisualEntity
Robovie Foot Left VisualEntity
Robovie Hip Right VisualEntity
Robovie Thigh Right VisualEntity
Robovie Knee Right VisualEntity
Robovie Calf Right VisualEntity
Robovie Ankle Left VisualEntity
Robovie Foot Left VisualEntity

FIGURE 43

4.7. Robovie-M base implementation

RobovieMBase is an abstract class that represents the whole humanoid robot container on the simulation environment. The *RobovieMBase* class extends *RobovieVisualEntity* and its signature is

```
/// <summary>
/// Robovie-MBase is a abstract class to represent a Robovie-M base visual /// entity.
/// <example>
/// RobovieMBase r = new RobovieM(...);
/// with RobovieM is a class to extend RobovieMBase and it's instanceble.
/// </example>
/// </summary>
[DataContract(Name = "RobovieMBase")]
public abstract class RobovieMBase : RobovieVisualEntity
{
  …
}
```

CODE 55

The class's constructor attends humanoid model initialization as shown in the following code:

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="name">Robovie-M's Name</param>
/// <param name="initPose">Robovie-M's Initial Position</param>
protected RobovieMBase(String name, Vector3 initPose)
{
  base.State.Name = name;
  base.State.Pose.Position = initPose;

  //initilizations
  this.Joints = new List<Joint>();
  this.Links = new List<RobovieArmVisualEntity>();
  this.ServoMotors =
    new Dictionary<ServoMotorName, RobovieServoMotorEntity>();

  //its creates component base for robovie visual entity instance.
  this.CreateRobovieBase(initPose);

  //create a component arm base
  this.CreateRobovieArmsBase(initPose);
}
```

CODE 56

The constructor attends to assign entity's name and entity's initial position and it creates the objects that needed as Joints, Links and ServoMotors. The Joints list concerns a joints list. A joint is a connector between two entities that have any position for connecting another entity as *LowerPose* or *UpperPose*. The Links list is a generic list of Robovie Arm that is not entity's child, but it is an entity that compose parent entity as in this case. All arm entity created, as shoulder, as elbow etc. aren't parent entities' child but they compose it. The ServoMotor list is a list of servo motor where for each servo has a particular characteristic. A servo motor can be of two types: SPECAPZ and ERGVB, one concerning the arms and the body joint and the second concerning legs joint. For this concept we created two entities that represents this concept; the follwing code show it:

```
/// <summary>
/// RobovieServoMotorEntity is a class with represent a generic Robovie-M servo motor
/// </summary>
[DataContract(Name = "RobovieServoMotorEntity")]
public class RobovieServoMotorEntity : RobovieShapesEntity { … }

/// <summary>
/// RobovieServoMotorHyperERGVBEntity represent an instance of ServoMotorHyperERGVB
/// </summary>
[DataContract(Name = "RobovieServoMotorHyperERGVBEntity")]
public class RobovieServoMotorHyperERGVBEntity : RobovieServoMotorEntity
{
  /// <summary>
  /// Constructor
  /// </summary>
  /// <param name="joint">ServoMotor's joint uses</param>
  /// <param name="rotateaxis">ServoMotor's axis rotate</param>
  public RobovieServoMotorHyperERGVBEntity(Joint joint, Vector3 rotateaxis)
    : base("Servo Motor Hyper ERG-VB", "13 kg x cm (6V)",
         new SpeedServoMotor(Degrees.D_060, 0.10f, rotateaxis), joint)
  {
    base._jointuse = joint;
    base._dimension = base.DIMENSION_SERVOMOTOR_HYPERERGVB;
    base._mass = base.MASS_SERVOMOTOR_HYPERERGVB;
  }
}

/// <summary>
/// RobovieServoMotorSPECAPZEntity represent an instance of ServoMotorSPECAPZ
/// </summary>
[DataContract(Name = "RobovieServoMotorSPECAPZEntity")]
public class RobovieServoMotorSPECAPZEntity : RobovieServoMotorEntity
{
  /// <summary>
  /// Constructor
   /// </summary>
  /// <param name="joint">ServoMotor's joint uses</param>
  /// <param name="rotateaxis">ServoMotor's axis rotate</param>
  public RobovieServoMotorSPECAPZEntity(Joint joint, Vector3 rotateaxis)
    : base("Servo Motor SPEC-APZ", "4 kg x cm (4.8V)",
         new SpeedServoMotor(Degrees.D_060, 0.20f, rotateaxis), joint)
  {
    base._jointuse = joint;
    base._dimension = base.DIMENSION_SERVOMOTOR_SPECAPZ;
    base._mass = base.MASS_SERVOMOTOR_SPECAPZ;
   }
}
```

Code 57

When a servo motor entity is created, we must set a joint that entity uses and his dimension together with his mass. The servo's characteristics are specification on *RobovieServoMotorService*. This argument will be treated in the next chapter when we will talk about RobovieM services.

The method *CreateRobovieBase* creates any base components as Head, Breast and Belly that are base components for humanoid robot entity. The following code shown this concept and creates base components:

```
/// <summary>
/// Create a Robovie-M's body base
/// </summary>
/// <param name="initPose">initial position</param>
void CreateRobovieBodyBase(Vector3 initPose)
{
  _head = new RobovieHeadEntity(initPose);
  _breast = new RobovieBreastEntity(initPose);
  _belly = new RobovieBellyEntity(initPose);

  //insert head in 0 position....
  _links.Insert(0, _head);
  _links.Add(_breast);
  _links.Add(_belly);
}
```

CODE 58

The Head entity must be inserted in the first position of the list, because this entity is a humanoid robot's principal entity.

CreateRobovieArmBase is a method to create robovie's arm part as arm left, arm right and leg left and leg right.

There are other methods for creating entity completed, the following code represents it:

```
/// <summary>
/// Create a Robovie-M's Arms Base. This method calls:
/// CreateRobovieArmLeftBase, CreateRobovieArmRightBase,
/// CreateRobovieLegLeftBase, CreateRobovieLegRightBase
/// <code>
/// CreateRobovieArmLeftBase(initPose);
/// CreateRobovieArmRightBase(initPose);
/// CreateRobovieLegLeftBase(initPose);
/// CreateRobovieLegRightBase(initPose);
/// </code>
/// </summary>
/// <param name="initPose">initial position</param>
protected void CreateRobovieArmsBase(Vector3 initPose)
{
  CreateRobovieArmLeftBase(initPose);
  CreateRobovieArmRightBase(initPose);
  CreateRobovieLegLeftBase(initPose);
```

```
   CreateRobovieLegRightBase(initPose);
}
```

CODE 59

Examples for creating robovie's arms are shown in the ollowing code:

```
void CreateRobovieArmLeftBase(Vector3 initPose)
{
  _shoulderL = new RobovieShoulderLeftEntity(initPose);
  _elbowL = new RobovieElbowLeftEntity(initPose);
  _wristL = new RobovieWristLeftEntity(initPose);
  _handL = new RobovieHandLeftEntity(initPose);

  _links.Add(_shoulderL);
  _links.Add(_elbowL);
  _links.Add(_wristL);
  _links.Add(_handL);
}

void CreateRobovieArmRightBase(Vector3 initPose)
{
  _shoulderR = new RobovieShoulderRightEntity(initPose);
  _elbowR = new RobovieElbowRightEntity(initPose);
  _wristR = new RobovieWristRightEntity(initPose);
  _handR = new RobovieHandRightEntity(initPose);

  _links.Add(_shoulderR);
  _links.Add(_elbowR);
  _links.Add(_wristR);
  _links.Add(_handR);
}

void CreateRobovieLegLeftBase(Vector3 initPose)
{
  _hipL = new RobovieHipLeftEntity(initPose);
  _thighL = new RobovieThighLeftEntity(initPose);
  _kneeL = new RobovieKneeLeftEntity(initPose);
  _calfL = new RobovieCalfLeftEntity(initPose);
  _ankleL = new RobovieAnkleLeftEntity(initPose);
  _footL = new RobovieFootLeftEntity(initPose);

  _links.Add(_hipL);
  _links.Add(_thighL);
  _links.Add(_kneeL);
  _links.Add(_calfL);
  _links.Add(_ankleL);
  _links.Add(_footL);
}

void CreateRobovieLegRightBase(Vector3 initPose)
{
  _hipR = new RobovieHipRightEntity(initPose);
  _thighR = new RobovieThighRightEntity(initPose);
  _kneeR = new RobovieKneeRightEntity(initPose);
  _calfR = new RobovieCalfRightEntity(initPose);
  _ankleR = new RobovieAnkleRightEntity(initPose);
  _footR = new RobovieFootRightEntity(initPose);

  _links.Add(_hipR);
  _links.Add(_thighR);
  _links.Add(_kneeR);
  _links.Add(_calfR);
  _links.Add(_ankleR);
  _links.Add(_footR);
}
```

CODE 60

An important concept concerns all robovie's entities that aren't inserted into *base.InsertEntity(child)*, but are inserted into Links list not as entity's child. Up to now we have created connectionless entities. For the connection of the two entities we use the following code, creating a *PhysicJoint* and connection the two entities.

```
/// <summary>
/// Create joints and servomotors for Robovie-M's body base. Connect head to breast
and breast to belly.
/// Connect any servomotor to joint created too.
/// </summary>
void CreateRobovieBodyJointsMotorsBase()
{
  //connect head visual entity to breast visual entity & from breast to belly
  PhysicsJoint pj0 =
    base.ConnectorEntities(_head, _breast, _bodyDOF,
                     base.NORMAL_X, base.ROTATE_Y);

  PhysicsJoint pj1 =
    base.ConnectorEntities(_breast, _belly, _bodyDOF,
                     base.NORMAL_Y, base.ROTATE_Z);

  //adds joint in to list of joints
  this.Joints.Add(pj0);
  this.Joints.Add(pj1);

  //create a servo motor visual entity and insert it in to head.ServoMotor
  //RunMotorTypeDirection represents mode servomotor to rotate.
  RobovieServoMotorEntity rsm0 =
     RobovieServoMotorEntity.CreateRobovieServoMotorSPECAPZ(pj0, ROTATE_Z);
  rsm0.NameServoMotor = "ServoMotorHead";
  _head.ServoMotor = rsm0;
  _head.ServoMotor.RunMotorTypeDirection = RunMotorType.Positive;

  //create a servo motor visual entity and insert it in to breast.ServoMotor
  //RunMotorTypeDirection represents mode servomotor to rotate.
  RobovieServoMotorEntity rsm1 =
    RobovieServoMotorEntity.CreateRobovieServoMotorSPECAPZ(pj1, ROTATE_Y);
  rsm1.NameServoMotor = "ServoMotorBreast";
  _breast.ServoMotor = rsm1;
  _breast.ServoMotor.RunMotorTypeDirection = RunMotorType.Positive;

  //adds servomotors in to list of servomotors
  this.ServoMotors.Add(ServoMotorName.SERVOMOTOR_V, rsm0);
  this.ServoMotors.Add(ServoMotorName.SERVOMOTOR_U, rsm1);
}
```

Code 61

This method creates two *PhysicJoint* objects: *pj0* and *pj1* and adding into joints list. Then it creates two servo motor, one with pj0 and one with pj1, sets the run motor direction that can be positive or negative and adds servo motors into list of servomotors. When we insert them in the list we set our name with *SERVOMOTOR_(A* | … … | *V)*. The servo motor's names are definite with enumeration shown in the following code:

```
public enum ServoMotorName
{
  SERVOMOTOR_V,//_HEAD,
  SERVOMOTOR_U,//_BREAST,

  SERVOMOTOR_G,//_SHOULDER_LEFT,
  SERVOMOTOR_H,//_ELBOW_LEFT,
  SERVOMOTOR_I,//_WRIST_LEFT,
  SERVOMOTOR_J,//_HAND_LEFT,

  SERVOMOTOR_Q,//_SHOULDER_RIGHT
  SERVOMOTOR_R,//_ELBOW_RIGHT,
  SERVOMOTOR_S,//_WRIST_RIGHT,
  SERVOMOTOR_T,//_HAND_RIGHT,

  SERVOMOTOR_A,//_HIP_LEFT,
  SERVOMOTOR_B,//_THIGH_LEFT,
  SERVOMOTOR_C,//_KNEE_LEFT,
  SERVOMOTOR_D,//_CALF_LEFT,
  SERVOMOTOR_E,//_ANKLE_LEFT,
  SERVOMOTOR_F,//_FOOT_LEFT,

  SERVOMOTOR_K,//_HIP_RIGHT,
  SERVOMOTOR_L,//_THIGH_RIGHT,
  SERVOMOTOR_M,//_KNEE_RIGHT,
  SERVOMOTOR_N,//_CALF_RIGHT,
  SERVOMOTOR_O,//_ANKLE_RIGHT,
  SERVOMOTOR_P,//_FOOT_RIGHT
};
```

Code 62

The precedent concept shown is generic for all the connections between two entities. Here we will show method's signature for other connections between two entities considering any typology of entity. An example is:

```
PhysicsJoint pj0 =
  base.ConnectorEntities(_shoulderL, _elbowL, _armDOF, NORMAL_Y, ROTATE_Z);

PhysicsJoint pj1 =
  base.ConnectorEntities(_elbowL, _wristL, _armDOF, NORMAL_X, ROTATE_Y);

PhysicsJoint pj2 =
  base.ConnectorEntities(_wristL, _handL, _armDOF, NORMAL_Y, ROTATE_Z);
```

```
this.Joints.Add(pj0);
this.Joints.Add(pj1);
this.Joints.Add(pj2);

RobovieServoMotorEntity rsm0 =
  RobovieServoMotorEntity.CreateRobovieServoMotorSPECAPZ(pj0, ROTATE_Z);
rsm0.NameServoMotor = "ServoMotorElbowLeft";
_elbowL.ServoMotor = rsm0;
_elbowL.ServoMotor.RunMotorTypeDirection = RunMotorType.Negative;

RobovieServoMotorEntity rsm1 =
  RobovieServoMotorEntity.CreateRobovieServoMotorSPECAPZ(pj1, ROTATE_Y);
rsm1.NameServoMotor = "ServoMotorWristLeft";
_wristL.ServoMotor = rsm1;
_wristL.ServoMotor.RunMotorTypeDirection = RunMotorType.Positive;

RobovieServoMotorEntity rsm2 =
  RobovieServoMotorEntity.CreateRobovieServoMotorSPECAPZ(pj2, ROTATE_Z);
rsm2.NameServoMotor = "ServoMotorHandLeft";
_handL.ServoMotor = rsm2;
_handL.ServoMotor.RunMotorTypeDirection = RunMotorType.Negative;

this.ServoMotors.Add(ServoMotorName.SERVOMOTOR_H, rsm0);
this.ServoMotors.Add(ServoMotorName.SERVOMOTOR_I, rsm1);
this.ServoMotors.Add(ServoMotorName.SERVOMOTOR_J, rsm2);
```

Code 63

This example shows as occurs arm left connection with three *PhysicJoint* and three servo motors. In general we can tell that for the connection of two or more entities we must write the following pseudo code:

```
For each pair entities to connect
{
  PhysicsJoint pj0 = base.ConnectorEntities(entity1, entity2,   {properties});
  Add pj0 into list of joints
  Create ServoMotor Entity with pj0;
  Add servomotor into list of servo motors
}
```

Code 64

This pseudo-code is just valid for all the entities that have *lowerpose* and *upperpose*, nothing else valid. The whole humanoid robot entity is seen in the following image. This represents all entities connections. The lines are connection between two entities by joint and the rectangles are entities.

There are some important methods used for initialization entity and rendering mode entity. The methods are: *Initialize(...)* and *Render(...)*, thay are override methods from class base.

```
/// <summary>
/// override Initialize VisualEntity
/// </summary>
/// <param name="device">GraphicsDevice</param>
/// <param name="physicsEngine">PhysicsEngine</param>
public override void Initialize(GraphicsDevice device, PhysicsEngine
     physicsEngine)
{
  base.Initialize(device, physicsEngine);

  //initialize the children visual entity
  foreach (VisualEntity ve in this.Children)
  {
    ve.Parent = this;
    ve.Initialize(device, physicsEngine);
  }

  //initialize the non-childern visual entity
  foreach (RobovieArmVisualEntity link in _links)
  {
    link.Parent = this;
    link.Initialize(device, physicsEngine);
  }

  //create joints for connection arms and legs entity
  this.CreateRobovieJointsMotorsBase();

  //adds arms and legs to entity base robot
  this.AddRobovieArmsToBodyBase();

  foreach (Joint joint in Joints)
    physicsEngine.InsertJoint((PhysicsJoint)joint);

  foreach (RobovieServoMotorEntity rsm in this.ServoMotors.Values)
    base.InsertEntity(rsm);

  // Compute the bounding sphere
  xna.Vector3 center = xna.Vector3.Zero;
  EntityBoundingSphere = new xna.BoundingSphere(center, 2.40f);
}
```

Code 65

The *initialize(...)* method initializes the base class, child entities, no-child entities, for each servo motor entity created a joint represent a point of connection, adds arms and legs to robovie's body and inserts list of joint in to physics engine. It inserts each servomotor entity as entity's child.

It sets a bounding sphere in pose center, so that it is comparable with the head position. This comparison is very important to know if robovie's has move or robovie's hasn't move.

The next method is Render method used for rendering mesh. The important functionality is to update the mesh pose into Microsoft Robotics Studio simulation. The following code represents a generic mesh update:

```
/// <summary>
/// override Update for a SumulatorMode is equal Edit
/// </summary>
/// <param name="update">FrameUpdate</param>
public override void Update(FrameUpdate update)
{
  base.Update(update);

  if (SimulationEngine.GlobalInstance.SimulatorMode ==
                              SimulationEngine.SimModeType.Edit)
  {
    xna.Vector3 linkPositionWS = Position;
    if (Parent != null)
    {
      linkPositionWS = xna.Vector3.Transform(linkPositionWS, Parent.World);
    }

   // Update positions if necessary (if user translates)
   if (linkPositionWS - _head.Position != xna.Vector3.Zero)
   {
     xna.Vector3 offset = linkPositionWS - _head.Position;

     foreach (RobovieArmVisualEntity link in _links)
     {
       link.Position += offset;
     }
     _head.Position = linkPositionWS;
   }

   // WorldSpace orientation of the arm
   xna.Quaternion linkOrientationWS =
                 TypeConversion.ToXNA(State.Pose.Orientation);
   if (Parent != null)
   {
      xna.Quaternion parentOrientation =
          xna.Quaternion.CreateFromRotationMatrix(Parent.World);
      linkOrientationWS = parentOrientation * linkOrientationWS;
   }

   // Update orientations if necesary (if user orients)
   xna.Quaternion orientationOffset = linkOrientationWS –
                TypeConversion.ToXNA(_head.State.Pose.Orientation);
   if (orientationOffset.LengthSquared() > .00001f)
   {
     // Compute the rotation amount
     xna.Quaternion rotation = linkOrientationWS *
 (TypeConversion.ToXNA(Quaternion.Inverse(_head.State.Pose.Orientation)));

      // -translation * rotation * translation => affine transform
      // Use this to update position
      xna.Matrix transform = xna.Matrix.CreateTranslation(-linkPositionWS) *
              xna.Matrix.CreateFromQuaternion(rotation) *
              xna.Matrix.CreateTranslation(linkPositionWS);
      for (int i = 1; i < _links.Count; ++i)
      {
        // Compute new position & orientation for the new pose
        xna.Vector3 position =
              (TypeConversion.ToXNA(_links[i].State.Pose.Position));
        position = xna.Vector3.Transform(position, transform);

       xna.Quaternion orientation =
                TypeConversion.ToXNA(_links[i].State.Pose.Orientation);
       orientation = rotation * orientation;

       Pose newPose =
           new Pose(TypeConversion.FromXNA(position),
                          TypeConversion.FromXNA(orientation));
       _links[i].State.Pose = newPose;
       _links[i].PhysicsEntity.SetPose(newPose);
     }

    // Assign orientation directly to avoid precision error
    _head.State.Pose.Orientation = TypeConversion.FromXNA(linkOrientationWS);
     _head.PhysicsEntity.SetPose(_head.State.Pose);
   }

   foreach (Joint joint in this.Joints)
   {
      if (joint != null)
        ((PhysicsJoint)joint).UpdateState();
   }
  }
  //updates non-children visual entity
  foreach (RobovieArmVisualEntity link in _links)
  {
     if (link != null)
       link.Update(update);
   }
}
```

Code 66

The update is valid when simulation is *Edit Type* state, then mesh or meshes are updated, something else only update is no-child updated. We want to remember that the no-child entities are contained into links list.

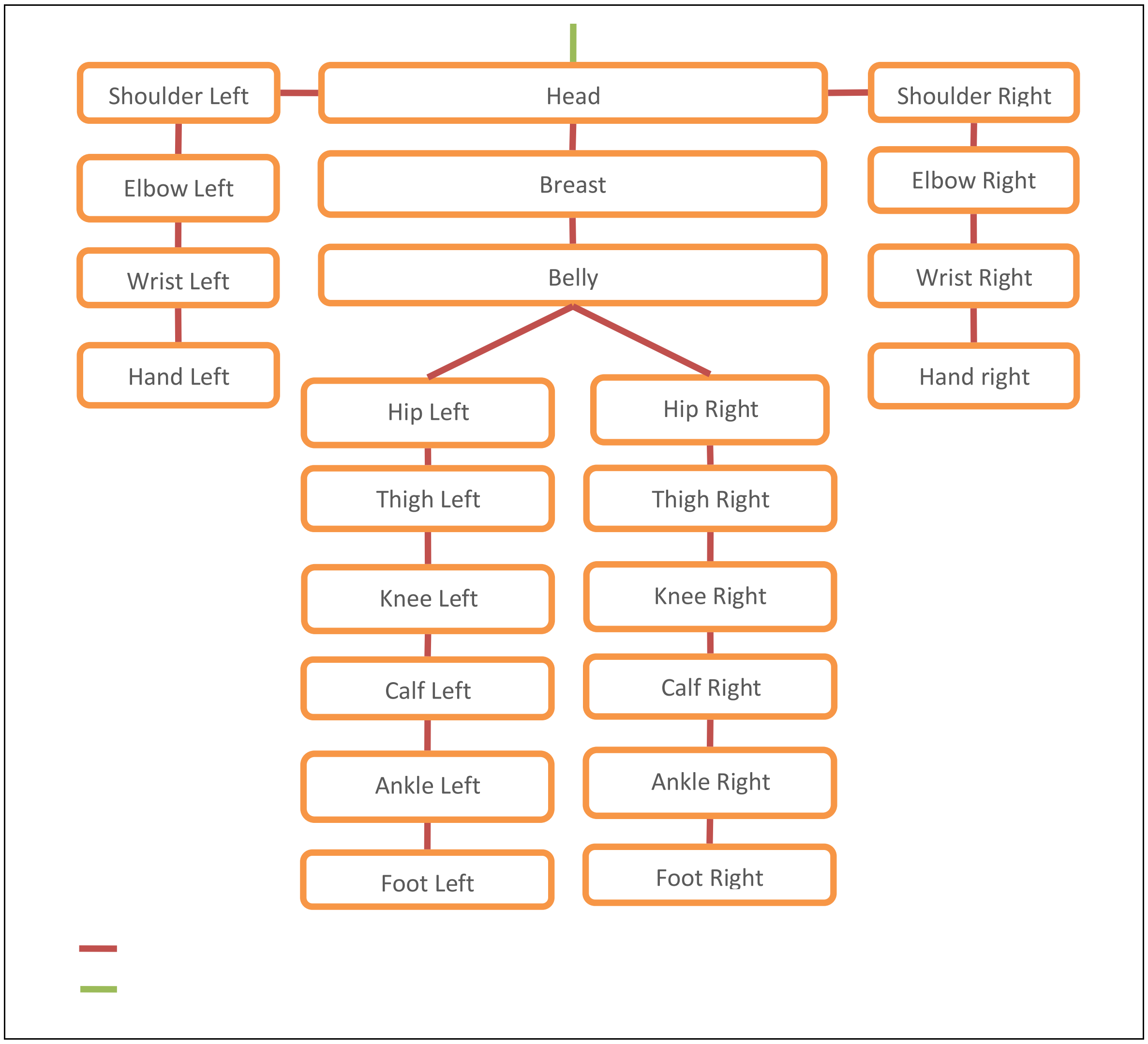


FIGURE 44

Up to now we have created a humanoid robot entity. It's static robot because all joints are locked on initialization position. If we want to move a single joint or a list of joint we must use same methods. This methods are: *SetJointTargetOrientation(...)*, *SetJointVelocity(...)* and *SetJointTargetPose(...)*. The first method is used to target joint's orientation, the second method is used to set joint's velocity and last method is used to set a joint's pose. In the following code three methods are shown:

```
/// <summary>
/// set orientation target joint
/// </summary>
/// <param name="j"></param>
/// <param name="axisAngle"></param>
public void SetJointTargetOrientation(Joint j, AxisAngle axisAngle)
{
  Task<Joint, AxisAngle> deferredTask = new Task<Joint, AxisAngle>(j, axisAngle,
SetJointTargetOrientationInternal);
  DeferredTaskQueue.Post(deferredTask);
}

/// <summary>
/// set velocity target joint
/// </summary>
/// <param name="j">joint to target Velocity</param>
/// <param name="velocity">velocity</param>
public void SetJointVelocity(Joint j, Vector3 velocity)
{
  Task<Joint, Vector3> deferredTask = new Task<Joint, Vector3>(j, velocity,
SetJointVelocityInternal);
  DeferredTaskQueue.Post(deferredTask);
}

/// <summary>
/// set pose target joint
/// </summary>
/// <param name="j">joint to target pose</param>
/// <param name="pose">position</param>
public void SetJointTargetPose(Joint j, Pose pose)
{
  Task<Joint, Pose> deferredTask = new Task<Joint, Pose>(j, pose,
SetJointTargetPoseInternal);
  DeferredTaskQueue.Post(deferredTask);
}

/// <summary>
/// Set orientation target Joint
/// </summary>
/// <param name="j">joint to Target Orientation</param>
/// <param name="axisAngle">Angle and Axis to Target Orientation joint</param>
public void SetJointTargetOrientationInternal(Joint j, AxisAngle axisAngle)
{
  if (j.State.Angular != null)
  {
    Quaternion target = TypeConversion.FromXNA(
      xna.Quaternion.CreateFromAxisAngle(
          TypeConversion.ToXNA(axisAngle.Axis), axisAngle.Angle)
    );
    ((PhysicsJoint)j).SetAngularDriveOrientation(target);
    }
}

/// <summary>
/// Set Velecity target joint
/// </summary>
/// <param name="j">joint to Target velocity</param>
/// <param name="velocity">Velocity</param>
public void SetJointVelocityInternal(Joint j, Vector3 velocity)
{
  if (j.State.Linear != null)
    j.State.Linear.DriveTargetVelocity = velocity;
  else
    j.State.Angular.DriveTargetVelocity = velocity;
}

/// <summary>
/// Set joint pose
/// </summary>
/// <param name="j">joint to target pose</param>
/// <param name="pose">position</param>
public void SetJointTargetPoseInternal(Joint j, Pose pose)
{
  if (j.State.Linear != null)
  {
    ((PhysicsJoint)j).SetLinearDrivePosition(pose.Position);
  }

  if (j.State.Angular != null)
  {
    ((PhysicsJoint)j).SetAngularDriveOrientation(pose.Orientation);
  }
}

/// <summary>
/// this method calculate AxisAngle from motor's name and degree angle...
/// </summary>
/// <param name="motor"></param>
/// <param name="degree"></param>
/// <returns></returns>
public AxisAngle CalculateAxisAngle(ServoMotorName motor, Degrees degree)
{
  AxisAngle axisAngle = new AxisAngle();
  if (ServoMotors[motor].RunMotorTypeDirection == RunMotorType.Positive)
    axisAngle =
        new AxisAngle(new Vector3(1f, 0f, 0f), degree.ToRadians.Value);
  else
    axisAngle =
        new AxisAngle(new Vector3(1f, 0f, 0f), -degree.ToRadians.Value);
  return axisAngle;
}
```

CODE 67

Once a Task is created by methods and inserted in the list of task, is then scheduled. The methods use a similar method’s name, an example, *void xxxxInternal(...),* where *xxxx* is a method’s name. The *xxxx* is a symbolic name.

The method, *void xxxxInternal(...)*, provides for command execution. *CalculateAxisAngle* is used to calculate as servomotor as running. If servomotor has run type positive, it run s clockwise if not it runs anticlockwise.

4.8. Robovie-M Useful Instance

For creating a useful instance we must extend *RobovieMBase* with a public class's constructor. The following code represents a public class:

```
/// <summary>
/// Instance of RobovieM
/// </summary>
[DataContract(Name = "RobovieM")]
public class RobovieM : RobovieMBase
{
  /// <summary>
  /// Default Constructor, set instance's name Robovie-M and set to pose (0, 0, 0)
  /// </summary>
  public RobovieM()
    : base("Robovie-M") { }

  /// <summary>
  /// Constructor set to pose (0,0,0)
  /// </summary>
  /// <param name="name">Robovie-M's name</param>
  public RobovieM(String name)
     :base(name)
  { }

  /// <summary>
  /// Constructor
  /// </summary>
  /// <param name="name">Robovie-M's name</param>
  /// <param name="initPose">Robovie-M's initial position</param>
  public RobovieM(String name, Vector3 initPose)
    : base(name, initPose)
  { }
}
```

Code 68

When we want to use an instance of *RobovieM*, we need to write the following line, that will write in to *RobovieM* Simulation service:

```
// Create an instance of our custom Robovie-M.
RobovieM _entity = new RobovieM(name, position);
```

A small important news item is: robovie's name and robovie's initial position are unique on the Microsoft Robotics Studio simulation; it isn't possible to create two similar instances on the simulation.

In the following image an entity is inserted at the initialization position, for correct the position is (0, 0, 0). We can view the approximated physic for this robot with shapes inserting in to model.

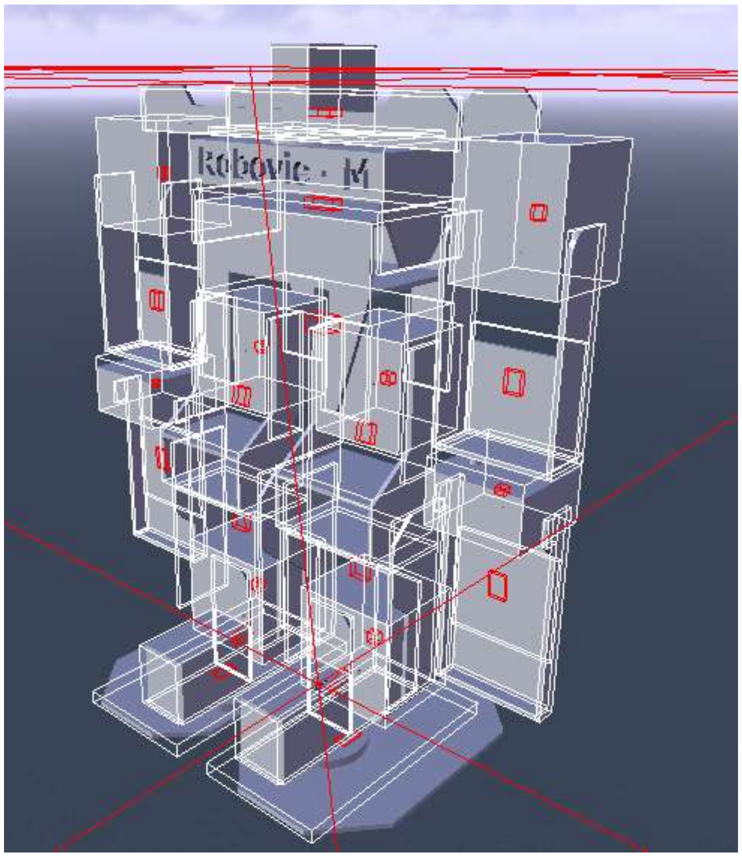


FIGURE 45

Preview image is the result of robovie-m creation. The red objects are stables points by entities that composed it, if all objects are red coloured the robovie-m is a stable position.

Chapter 5

Robovie-M Services

5.1. Introduction

Before examining implemented services in details, it is worth examining the main architecture of the system and the other design possibility that we didn't take account. Intuitively, there exist any implementations, each one with different features but all suitable for our aims.

By now, we examining two kind of architectures: a single-service architecture and a multi-service one. With respect to our implementation, we have chose the latter.

The simplest architecture, namely the single-service, where it manages all joints with a single service.

The following figure shows a graphical representation of the single-service architecture.

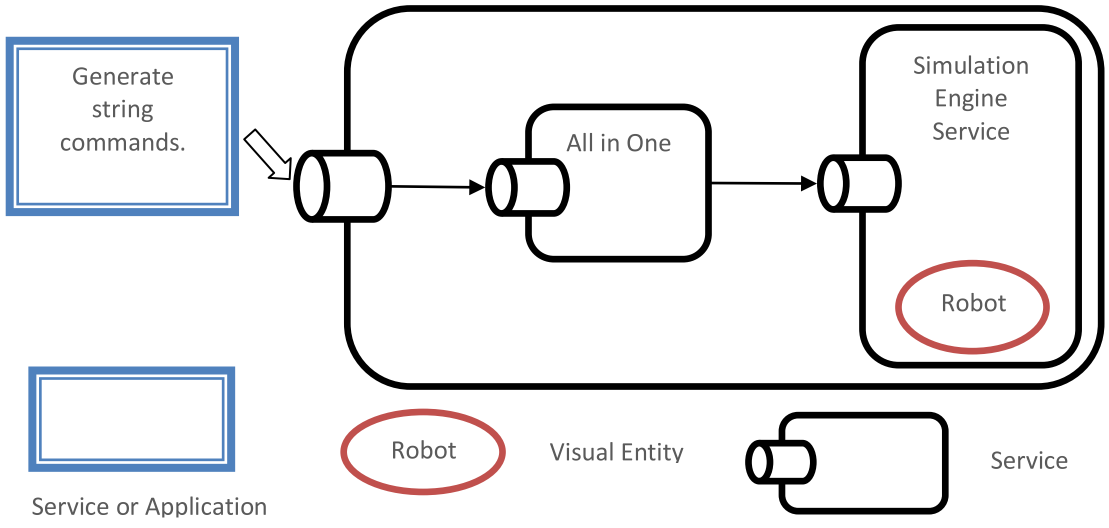


FIGURE 46

With respect to this architecture, we would have low parallelism, as each request would have make rotate all the joints using unique service only. The service, or analogously the *"Generate String Commands"* application, is used to generate well-formed command strings for the robot. The *"All in One"* service is used to manage all the dynamic behavior of the robot inside the simulation environment. Our choice, namely the multi-service architecture, models each component of the robot in a modular way; in particular, each servo motor and each arm, namely a composition of many servo motors whose make rotate one and only joint, is singularly modeled. Our model provides a high parallelism and it corresponds to a more similar model of the real nature of robots. The following figure shows:

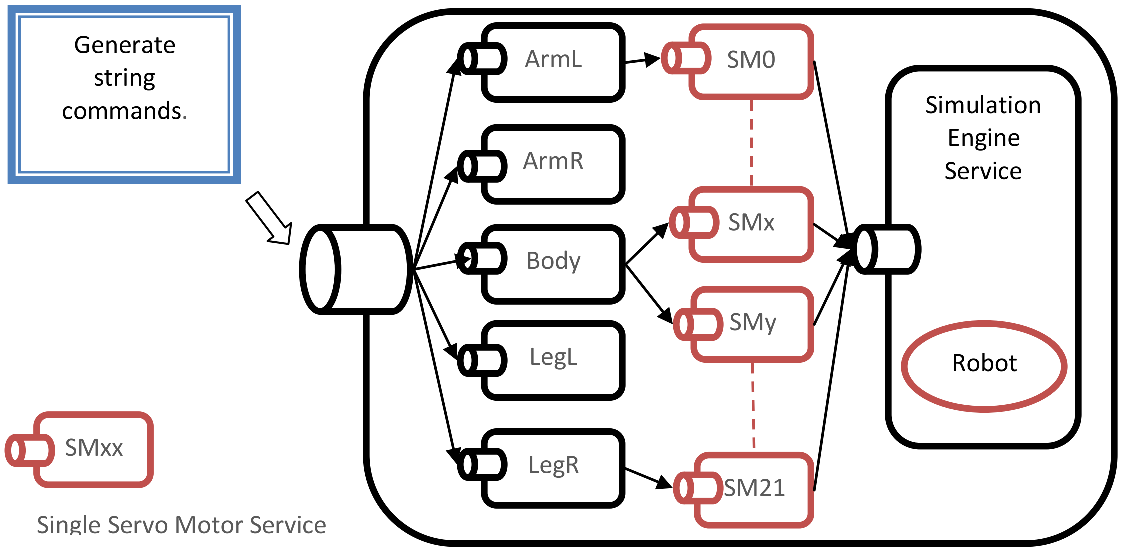

Generate string commands.
ArmL
ArmR
Body
LegL
LegR
SM0
SMx
SMy
SM21
Simulation Engine Service
Robot
SMxx
Single Servo Motor Service

FIGURE 47

As regard the generator of well-formed command strings for the robot, we used the same generator we could have used in the case of the single-service architecture.

The services are modeled by a more complex architecture. In particular, each arm is managed singularly and each component sends messages to the *Simulation Engine Service* generating the movement of the robot within the simulation environment.

Summarizing, after this simple introduction, we can examine each service, in particular we will show the architecture and the possible messages that can be sent or received by each service.

The service is part for managing a humanoid robot. The complete service list used is:

- RobovieArmLeftService
- RobovieArmRightService
- RobovieBodyService
- RobovieLegLeftService
- RobovieLegRightService
- RobovieServoMotorService

- Roboviegeneric22SMDriveService
- RobovieSimulationService

- RobovieControllerUIService

We now talk about small service's description then in a second moment we will talk about more service's details information.

RobovieArmXXXXService, RobovieBodyService and RobovieLegXXXService are used to manage a single robovie's arm, where *XXXX* can be Left or Right value. For each services has a defined number of servo motors that are necessary to move a single joint that associating it. Generally *Robovie Arm* has four servo motors, *Robovie Leg* has six servo motors and *Robovie Body* has two servo motors. In the next part we will treat same services type: two, four and six servo motors association.

RobovieGeneric22SMDriveService is a service containing all the other services to manage only type robot. When we use this service only model management is RobovieMBase, or it extend this class type. The composition of services group allows to expand structure and we have modular structure to collaborate between services. This model allows to simplicity to compose for managing other entities using Microsoft Visual Programming Language. The service receivers a single or multi string command and it provide execution received command. It can use a simulation manifest file for compose two services in to unique service.

RobovieServoMotorService provides the execution of a single joint's rotation. A servo motors service manages one and only one joint. It receives a hexadecimal command composed bytwo hex values and it rotates a joint with 180° value range. The rotation can be of two values: clockwise or anticlockwise. This value is defined by *RunType.Positive* and *RunType.Negative.*

RobovieSimulation, provides to render a robot in to Microsoft Robotics Studio simulation.

For all the services we will describe the same characteristics that compose it. The services made are grouped by the number of servo motors.

Using Services by Application Model

The application model is an abstract structure, a base to know overview application functionally. We supposed that application is divided into three fundamentals parts:

- Strings command generation
- Driver humanoid robot on the simulation
- Humanoid robot Simulation

Format string command to humanoid robot is a string's type:

```
@0fA8aB86C00D95Ec5F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
```

Where @xx represents speed execution and AxxBxx….Vxx are servo motors' generic position and the substring "+1!01" represents the end of command.

This command is a single string command. When we create a multi string commands we will need to create the following strings:

```
@0fA8aB86C00D95Ec5F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
@0fA8aB86C00D95Ec5F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
@14A8aB68C2cD95Ea4F6aGe3H60IffJ1bK81L90M7dN3fO70P89Qa2R2dSf5Tf5U80V80,+1!01
@14A8aB68C2cD95Ea4F6aGe3H60IffJ1bK81L90M7dN3fO70P89Qa2R2dSf5Tf5U80V80,+1!01
@0fA8aB86C2cD95E82F6aGe3H60IffJ1bK81L90M7dN43O70P89Qf6R2dSf5Tf5U80V80,+1!01
@0fA8aB86C2cD95E82F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
@05A97B7dC2cD85E93F6aGb5H60IffJ1bK8eL87M7dN80O82P89Qa2R0fSf5Tf5U80V80,+1!01
@0fA97B77C2cD76E93F6aGb5H60IffJ1bK8eL9eM7dN70O4dP89Qa2R0fSf5Tf5U80V80,+1!01
@0fA8aB86C00D95Ec5F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
@0fA8aB86C2cD95E82F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
@05A97B7dC2cD85E93F6aGb5H60IffJ1bK8eL87M7dN80O82P89Qa2R0fSf5Tf5U80V80,+1!01
@0fA97B77C2cD76E93F6aGb5H60IffJ1bK8eL9eM7dN70O4dP89Qa2R0fSf5Tf5U80V80,+1!01
@14A8aB68C2cD95Ea4F6aGe3H60IffJ1bK81L90M7dN3fO70P89Qa2R2dSf5Tf5U80V80,+1!01
@14A8aB68C2cD95Ea4F6aGe3H60IffJ1bK81L90M7dN3fO70P89Qa2R2dSf5Tf5U80V80,+1!01
@0fA8aB86C2cD95E82F6aGe3H60IffJ1bK81L90M7dN43O70P89Qf6R2dSf5Tf5U80V80,+1!01
@0fA8aB86C2cD95E82F6aGe3H60IffJ1bK81L90M7dN60O70P89Qa2R0fSf5Tf5U80V80,+1!01
@05A97B7dC2cD85E93F6aGb5H60IffJ1bK8eL87M7dN80O82P89Qa2R0fSf5Tf5U80V80,+1!01
```

The following scheme is an approximate data-flow about the collaboration between components.

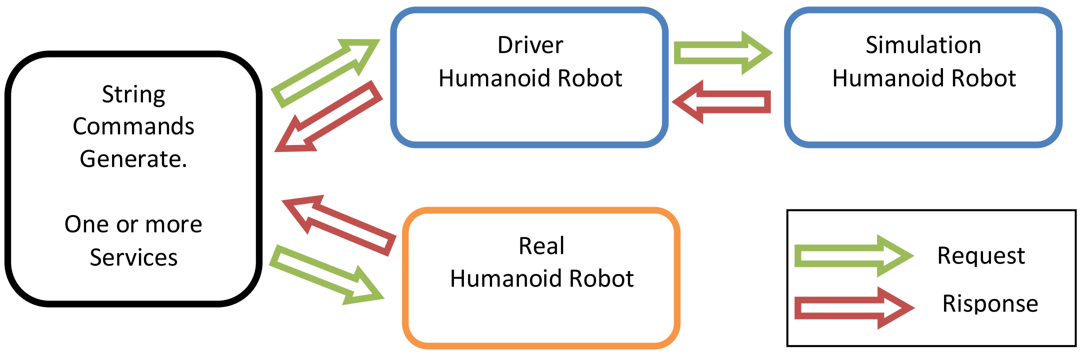


FIGURE 48

The generation string command can be of one or more services, it can be an artificial intelligence or a joystick where a person is controlling it or it can be of another service typology. This typology service makes sending a message as request, and it waits a response by execution of the request command.

The *Driver Humanoid Robot* is a collection of services that are manipulating humanoid robot on the *Simulation Humanoid Robot*. The collection sends command on the simulation by moving a robovie entity.

The *Simulation Humanoid Robot* response is a humanoid robot's state. The *Driver Humanoid Robot* receives this state computing it with state and re-sending a new command to the simulation and so on until *String Commands Generation* service sends a stop command. The *String Commands Generation* sends commands on *Real Humanoid Robot* too.

Now in the next section, we will introduce a collection of services for manipulating humanoid robot on the Microsoft robotics studio simulator. We will start talking about generic service containing same servo motors.

The most important part is when we will talk about the *Generic twenty Two Servo Motors Service* that is composed by services that the most important service in this project. In the last part we will talk about the simulation service and User Interface service.

Generic Service with Servo Motors

A generic service has the following structure. It has the same message handler used in receiving a generic message on *Main Port*, and it has a number of servo motors depending on the number of joint used for the management and the service's state.

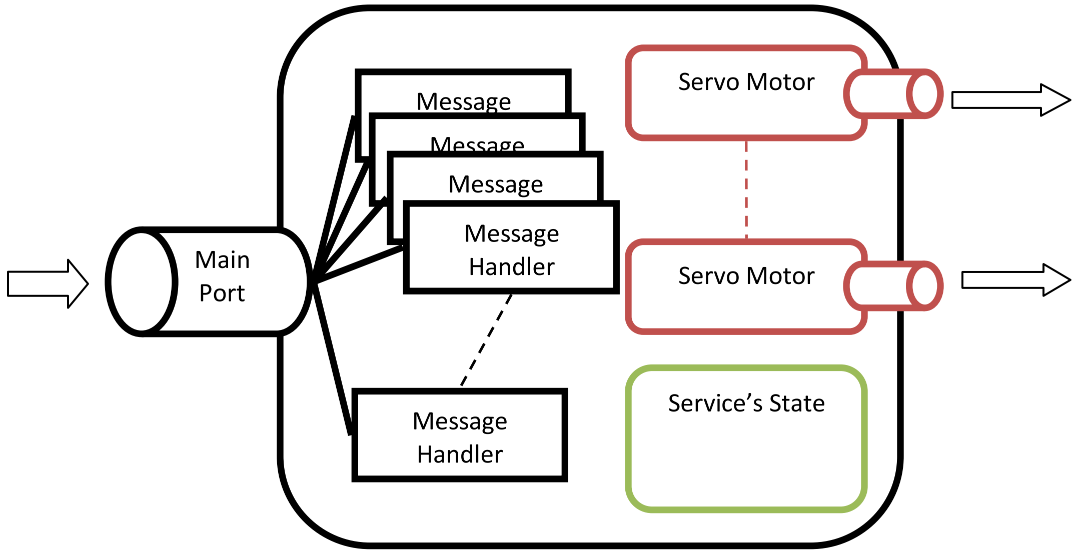


FIGURE 49

A generic servo motor service has notification message by which is possible to know servo motor's state and actually servo motor's position. A generic service used in mode type: the service receives a message type, it provides splitting message in to N sub instruction for a single servo motor and runs it.

When servo motor has reached the goal then sends a notification message to Main Port by notification that the operation has been completed and service's state updated. When all servo motors have finished running mode and sending message notification, the service sends a *CompletedTargetCommand* message notification.

The following code represents a code base for a service with same servo motor services partner:

```
[Partner("0_ServoMotor", Contract = drive.Contract.Identifier,
      CreationPolicy = PartnerCreationPolicy.CreateAlways)]
drive.RobovieServoMotorOperations _drive0Port =
  new drive.RobovieServoMotorOperations();
drive.RobovieServoMotorOperations _drive0PortNotify =
  new drive.RobovieServoMotorOperations();
           .
           .
           .
           .
[Partner("N_ServoMotor", Contract = drive.Contract.Identifier,
      CreationPolicy = PartnerCreationPolicy.CreateAlways)]
drive.RobovieServoMotorOperations _driveNPort =
  new drive.RobovieServoMotorOperations();
drive.RobovieServoMotorOperations _driveNPortNotify =
   new drive.RobovieServoMotorOperations();
```

CODE 69

Firstly the so called *start* base method we subscribe partner on this service and then we call a *base.Start()* by initialization for running service how shown in the following code :

```
protected override void Start()
{
  _drive0Port.Subscribe(_drive0PortNotify);
  .
  _driveNPort.Subscribe(_driveNPortNotify);

  base.Start();

  MainPortInterleave.CombineWith(
    new Interleave(
      new ExclusiveReceiverGroup( … ),
      new ConcurrentReceiverGroup( … )
    )
  );
}
```

CODE 70

The service must be ready to receive same messages from service partner. A servo motor during our execution sends notification of the actual joint’s position, so is sending on time a notification of the frequency changed state.

5.2. Servo Motor Service

The following drawing represents Servo Motor's logic structure:

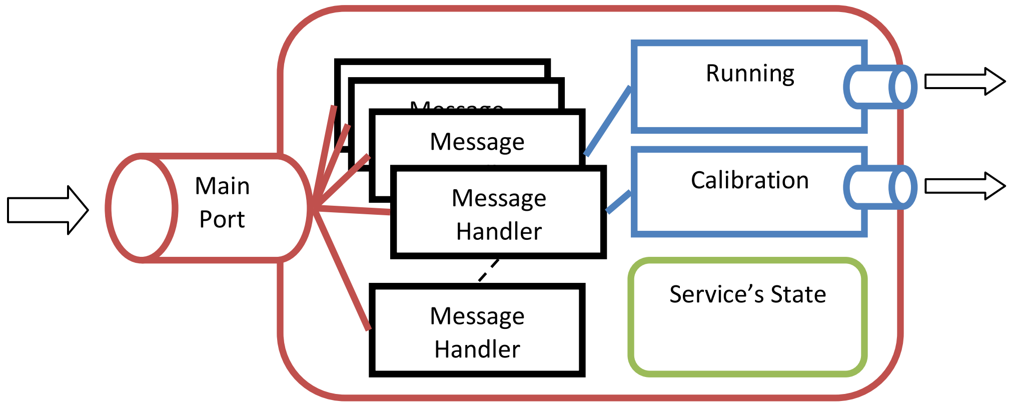


FIGURE 50

The servo motor service can receive same message types that are explicit as *PortSet<>* class. It has two methods for running and calibrating service's joint used.

The messages type definitions that service can receive are:

```
/// <summary>
/// RobovieServoMotor Main Operations Port
/// </summary>
public class RobovieServoMotorOperations : PortSet<
   DsspDefaultLookup, DsspDefaultDrop,
   Get, HttpGet,
   Replace, Subscribe,
   PowerOn, PowerOff,
   SetMotorPower, SetMotorSpeed, StopMotor,
   SetName, PowerTargetChanged, TargetCompleted, SetTargetCalibration> { … }
```

CODE 71

The messages are composed by a message request and a message response. The messages are explicit in the following code:

```
public class PowerOn :
Update<PowerOnRequest, PortSet<PowerResponse, Fault>>
{}
public class PowerOff :
Update<PowerOffRequest, PortSet<PowerResponse, Fault>>
{}
public class SetMotorPower :
Update<SetMotorPowerRequest, PortSet<DefaultUpdateResponseType, Fault>>
{}
public class SetMotorSpeed :
Update<SetMotorSpeedRequest, PortSet<DefaultUpdateResponseType, Fault>>
```

```
{}
public class StopMotor :
Update<StopMotorRequest, PortSet<DefaultUpdateResponseType, Fault>>
{}
public class SetName :
Update<SetNameRequest, PortSet<DefaultUpdateResponseType, Fault>>
{}
public class PowerTargetChanged :
Update<PowerTargetChangedRequest, PortSet<DefaultUpdateResponseType, Fault>>
{}
public class TargetCompleted :
Update<TargetCompletedRequest, PortSet<DefaultUpdateResponseType, Fault>>
{}
public class SetTargetCalibration :
Update<SetTargetCalibrationRequest, PortSet<DefaultUpdateResponseType, Fault>>
{}
```

CODE 72

PowerOn, *PowerOff* messages are use power on and power off servo motor. When a message arrives on Main Port the service schedules it with:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> PowerOnHandler(PowerOn on)
{
   if (!_state.IsOn)
   {
     _state.IsOn = true;
     on.ResponsePort.Post(new PowerResponse());
     SendNotification<PowerOn>(_submgr, on);
     yield break;
    }
    on.ResponsePort.Post(new W3C.Soap.Fault());
    yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> PowerOffHandler(PowerOff off)
{
  if (_state.IsOn)
  {
     _state.IsOn = false;
     off.ResponsePort.Post(new PowerResponse());
     SendNotification<PowerOff>(_submgr, off);
     yield break;
  }
  off.ResponsePort.Post(new W3C.Soap.Fault());
  yield break;
}
```

CODE 73

SetMotorSpeed, *SetName* messages are used to setting motor's speed and motor's name. SetMotorPower message and *PowerTargetChanged* message are used for setting a string command message that runs servo motor if the actual state is different from the requested message and *PowerTargetChanged* is a notification when servo motor's state actual position changes. When a *SetMotorPower* message arrives on Main Port the service is scheduled with the following handler code:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> SetMotorPowerHandler(SetMotorPower update)
{
   if (_state.IsOn)
   {
      if (!_state.IsRunning)
      {
         _state.IsRunning = true;
         _state.PowerTarget = ToIntDegress(update.Body.PowerTargetHex);
         _state.PowerTargetHex = update.Body.PowerTargetHex;

         SendNotification<SetMotorPower>(_submgr, update);
         Spawn<DateTime>(DateTime.Now, RunningServoMotor);

         Activate(
          Arbiter.Receive(false,
           _completePowerTargetPort, delegate(bool result)
           {
             if (result)
             {
              update.ResponsePort.Post(DefaultUpdateResponseType.Instance);
              _mainPort.Post(new TargetCompleted());
             }
             else
              update.ResponsePort.Post(new W3C.Soap.Fault());
            })
           );
           yield break;
      }
      else
      {
        update.ResponsePort.Post(new W3C.Soap.Fault());
        yield break;
      }
    }
    update.ResponsePort.Post(new W3C.Soap.Fault());
    yield break;
}
```

Code 74

Handler waiting a new message to arrives on Main Port. When the service has been scheduled we set a current request in the service' state and it throws *RunningServoMotor* method by means of:

```
Spawn<DateTime>(DateTime.Now, RunningServoMotor);
```

RunningServoMotor method provides for the joint to be updated by incrementing a current position. The method's code is shown in the following:

```
void RunningServoMotor(DateTime signal)
{
 if (_state.IsRunning)
 {
  if (_state.PowerTarget < _state.CurrentTarget)
    _state.CurrentTarget--;
  else
    _state.CurrentTarget++;

  if (_state.PowerTarget == _state.CurrentTarget)
    _state.IsRunning = false;

   _state.CurrentTargetHex = ToHex(_state.CurrentTarget);

   PowerTargetChanged changed = new PowerTargetChanged();
   changed.Body.CurrentTargetValue = _state.CurrentTarget;
   changed.Body.CurrentTargetValueHex = _state.CurrentTargetHex;

   SendNotification<PowerTargetChanged>(_submgr, changed);

   Activate(
    Arbiter.Receive(false,TimeoutPort(_state.SpeedTarget), RunningServoMotor)
   );
 }
 else
 {
   _completePowerTargetPort.Post(true);
 }
}
```

CODE 75

When string command is completed by service, it sends a new message with body current value true on *_completePowerTargetPort* that provides notification complete operation with a new message.

When message *SetTargetCalibration* arrives on Main Port, it schedules with the following code:

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask>SetTargetCalibrationHandler(SetTargetCalibration
update)
{
  Calibration(update);
  SendNotification<SetTargetCalibration>(_submgr, update);
  yield break;
}

void Calibration(SetTargetCalibration update)
{
   _state.EQUILIBRIUMTORQUE = ToIntHex(update.Body.TargetPoseZeroHex);

   float k = 180.0f / 255.0f;
   float max = (255 - _state.EQUILIBRIUMTORQUE) * k;
   float min = -(_state.EQUILIBRIUMTORQUE * k);
```

```
    _state.MAXTORQUE = (int)max;
    _state.MINTORQUE = (int)min;

    _state.CurrentTarget = ToIntDegress(update.Body.TargetPoseZeroHex);
    _state.CurrentTargetHex = update.Body.TargetPoseZeroHex;

    _state.IsCalibration = true;
}
```

Code 76

The Handler method throws calibration method. It is used to calibrate a servomotor in initial position. We show a request for execution on servomotor, that is *SetMotorPowerRequest* request. It is shown in the following code:

```
[DataContract()]
public class SetMotorPowerRequest
{
  private int _target = 0;
  private String _targethex = String.Empty;

  [DataMember()]
  public String PowerTargetHex
  {
    get { return _targethex; }
    set { _targethex = value;}
   }
}
```

Code 77

PowerTargetHex member is a servo motor request. It can be from 00 to FF value. The calibration joint is necessary to decide the initial position and maximum, minimum torque.

5.3. Robovie Body Service

The body service manages robovie's body part. It receives a string command to steer two servo motors. The first servo motor is used to move head – breast connection joint and the second servo motor are used to move breast belly connection joint. A service's scheme is showen in the following picture:

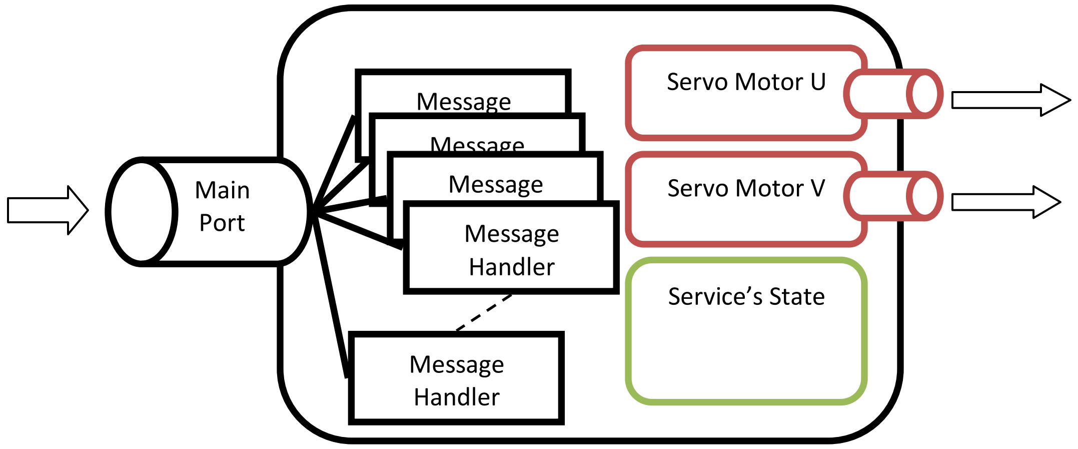


FIGURE 51

The service have two service's partner (U servo motor, V servo motor) and a main port receives same request type. The following code represents this concept:

```
[Partner("UServoMotor", Contract = drive.Contract.Identifier,
          CreationPolicy = PartnerCreationPolicy.CreateAlways)]
drive.RobovieServoMotorOperations _driveUPort =
      new drive.RobovieServoMotorOperations();
drive.RobovieServoMotorOperations _driveUPortNotify =
      new drive.RobovieServoMotorOperations();

[Partner("VServoMotor", Contract = drive.Contract.Identifier,
          CreationPolicy = PartnerCreationPolicy.CreateAlways)]
drive.RobovieServoMotorOperations _driveVPort =
      new drive.RobovieServoMotorOperations();
drive.RobovieServoMotorOperations _driveVPortNotify =
      new drive.RobovieServoMotorOperations();

[ServicePort("/roboviebody", AllowMultipleInstances=false)]
private RobovieBodyOperations _mainPort = new RobovieBodyOperations();
```

CODE 78

For all partner service there are two ports, one to create a real partner and the second, port notification, to capture the changed about it. When service starts, it subscribes service; an example is *_driveUPort* and *_driveUPortNotify*.

Firstly operation called *base.Start()* start the service must subscribe these partner services they listen to partner notification ports, it shown in the following code:

```
protected override void Start()
{
  _driveUPort.Subscribe(_driveUPortNotify);
  _driveVPort.Subscribe(_driveVPortNotify);

  base.Start();

  MainPortInterleave.CombineWith(
   new Interleave(

    new ExclusiveReceiverGroup(),

    new ConcurrentReceiverGroup(

      Arbiter.Receive<drive.PowerTargetChanged>(true,
        _driveUPortNotify,
        UPowerTargetChangedNotificationHandler),

      Arbiter.Receive<drive.PowerTargetChanged>(true,
        _driveVPortNotify,
        VPowerTargetChangedNotificationHandler),

      Arbiter.Receive<drive.TargetCompleted>(true,
        _driveUPortNotify,
        delegate(drive.TargetCompleted update)
        {
           _mainPort.Post(new TargetCompleted());
        }),

      Arbiter.Receive<drive.TargetCompleted>(true,
        _driveVPortNotify,
        delegate(drive.TargetCompleted update)
        {
            _mainPort.Post(new TargetCompleted());
        })
      )
    )
  );

  SetServoMotorInitialize();
}

void UPowerTargetChangedNotificationHandler(drive.PowerTargetChanged update)
{
   UPowerTargetChanged power = new UPowerTargetChanged();
  power.Body.CurrentTargetValue = update.Body.CurrentTargetValue;
  power.Body.CurrentTargetValueHex = update.Body.CurrentTargetValueHex;
  _mainPort.Post(power);
}

void VPowerTargetChangedNotificationHandler(drive.PowerTargetChanged update)
{
  VPowerTargetChanged power = new VPowerTargetChanged();
  power.Body.CurrentTargetValue = update.Body.CurrentTargetValue;
  power.Body.CurrentTargetValueHex = update.Body.CurrentTargetValueHex;
  _mainPort.Post(power);
}
```

CODE 79

During running partner service sends any notification that is captured, for example, by correctly *Arbiter.receiver*.

When it capturs a message it shows appropriated method. The service's message handlers received on the main port are explicit the following code:

```
public class RobovieBodyOperations : PortSet<
    DsspDefaultLookup, DsspDefaultDrop, Get, HttpGet, Replace, Subscribe,
    UPowerTargetChanged, VPowerTargetChanged,     PowerOn, PowerOff, SetMotorSpeed,
    SetMotorsPowerTargetCommand, TargetCompleted, Calibration> { … }
```

CODE 80

The important difference from Servo Motor Service is that can be received a *SetMotorsPowerTargetCommand* message, that contains a string command for two servo motors but servo motor service can received a single command for it.

The *SetMotorPowerTargetCommand* message handler is defined in the following code:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask>
SetPowerMotorsTargetCommandHandler(SetMotorsPowerTargetCommand update)
{
  drive.SetMotorPower target = new drive.SetMotorPower();

  target.Body.PowerTargetHex = update.Body.UPowerTargetHex;
  _driveUPort.Post(target);

  target.Body.PowerTargetHex = update.Body.VPowerTargetHex;
  _driveVPort.Post(target);

  yield break;
}
```

CODE 81

It provides splitting string command and it sends single command on the single servo motor port. The services we will treat in the following part are very similar at the precedent service. In fact we will not prolonging with specific details, but we will talk about essential part and the important differences between precedent service and the following service.

5.4. Robovie Arm Left and Right Services

The Arms service manage robovie's arm left and arm right part. It receives a string command to steer four servo motors. The first servo motor is used to move head-shoulder connection joint and the second servo motor is used to move shoulder-elbow connection joint, the third servo motor is used to move elbow-wrist connection joint and last servo motor is used for the wrist-hand connection joint.

A service's scheme is shown in the following picture:

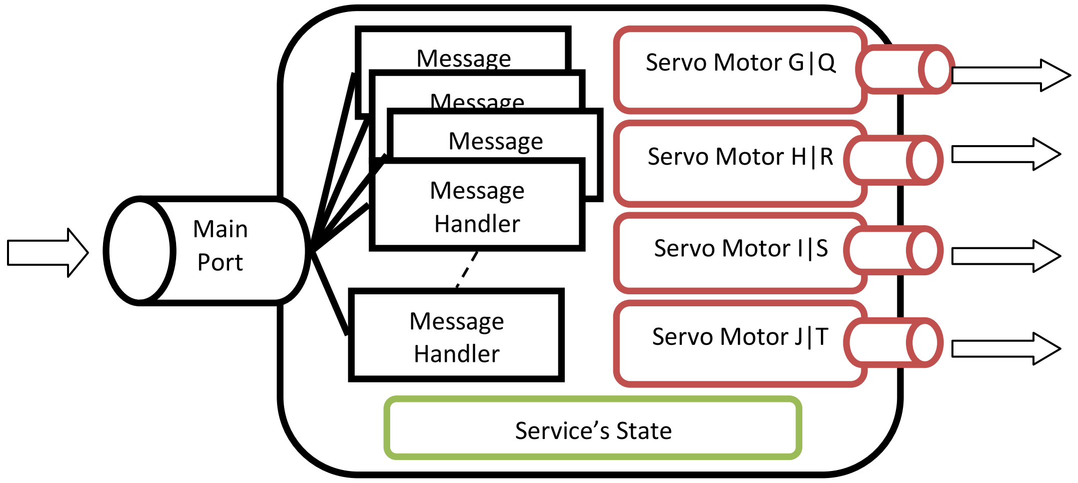


FIGURE 52

The services' message can be it received on Main Port by both services that are:

```
public class RobovieArmLeftOperations :
  PortSet<
    DsspDefaultLookup, DsspDefaultDrop,
    Get, HttpGet, Replace, Subscribe,
    GPowerTargetChanged, HPowerTargetChanged,
    IPowerTargetChanged, JPowerTargetChanged,
    Calibration,
    PowerOn, PowerOff, SetMotorSpeed,
    SetMotorsPowerTargetCommand, TargetCompleted> { … }

public class RobovieArmRightOperations :
  PortSet<
    DsspDefaultLookup, DsspDefaultDrop,
    Get, HttpGet, Replace, Subscribe,
    QPowerTargetChanged, RPowerTargetChanged,
    SPowerTargetChanged, TPowerTargetChanged,
    PowerOn, PowerOff, SetMotorSpeed,
    SetMotorsPowerTargetCommand, TargetCompleted, Calibration> { … }
```

CODE 82

The precedent code shown all the messages that arm left and arm right can receive. The *SetMotorPowerTargetCommand* message handler is defined in the following code:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask>
   SetPowerMotorsTargetCommandHandler(SetMotorsPowerTargetCommand update)
{
  drive.SetMotorPower target = new drive.SetMotorPower();
  target.Body.PowerTargetHex = update.Body.GPowerTargetHex;
  _driveGPort.Post(target);
  target.Body.PowerTargetHex = update.Body.HPowerTargetHex;
  _driveHPort.Post(target);
  target.Body.PowerTargetHex = update.Body.IPowerTargetHex;
  _driveIPort.Post(target);
  target.Body.PowerTargetHex = update.Body.JPowerTargetHex;
  _driveJPort.Post(target);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask>
   SetPowerMotorsTargetCommandHandler(SetMotorsPowerTargetCommand update)
{
   drive.SetMotorPower target = new drive.SetMotorPower();
  target.Body.PowerTargetHex = update.Body.QPowerTargetHex;
  _driveQPort.Post(target);
  target.Body.PowerTargetHex = update.Body.RPowerTargetHex;
  _driveRPort.Post(target);
  target.Body.PowerTargetHex = update.Body.SPowerTargetHex;
  _driveSPort.Post(target);
  target.Body.PowerTargetHex = update.Body.TPowerTargetHex;
  _driveTPort.Post(target);
  yield break;
}
```

CODE 83

Both services provide splitting string command and sends a single command on single servo motor port.

5.5. Robovie Leg Left and Right Services

The Legs service manages robovie's leg left and leg right part. It receives a string command to steer six servo motors. The first servo motor is used to move belly-hip connection joint and the second servo motor is used to move hip-thigh connection joint, the third servo motor is used to move thigh-knee connection joint and last three servo motors for connecting joint knee-calf, calf-ankle and ankle-foot. A service's scheme is shown in the following picture:

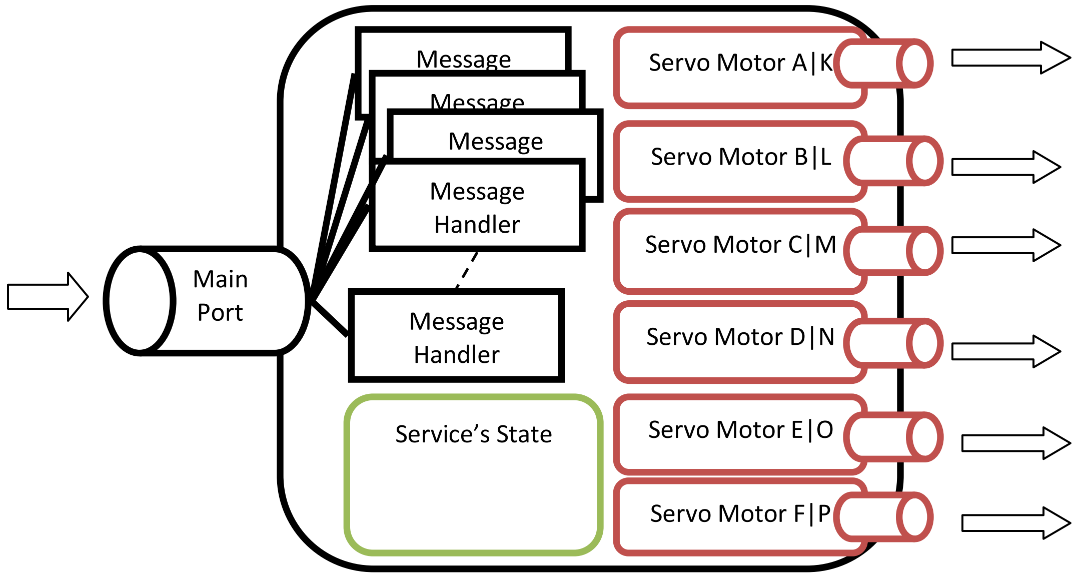


FIGURE 53

The services' message can be received on Main Port by both services that are:

```
public class RobovieLegLeftOperations
  : PortSet<
    DsspDefaultLookup, DsspDefaultDrop,
    Get, HttpGet, Replace, Subscribe,
    APowerTargetChanged, BPowerTargetChanged, CPowerTargetChanged,
    DPowerTargetChanged, EPowerTargetChanged, FPowerTargetChanged,
    PowerOn, PowerOff, SetMotorSpeed,
    SetMotorsPowerTargetCommand, TargetCompleted, Calibration>{ … }

public class RobovieLegRightOperations
: PortSet<
    DsspDefaultLookup, DsspDefaultDrop,
    Get, HttpGet, Replace, Subscribe,
    KPowerTargetChanged, LPowerTargetChanged, MPowerTargetChanged,
    NPowerTargetChanged, OPowerTargetChanged, PPowerTargetChanged,
    PowerOn, PowerOff, SetMotorSpeed,
    SetMotorsPowerTargetCommand, TargetCompleted, Calibration> { … }
```

CODE 84

The precedent code shows all messages that leg left and leg right can receive.

The *SetMotorPowerTargetCommand* message handler is defined in the following code:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask>
SetPowerMotorsTargetCommandHandler(SetMotorsPowerTargetCommand update)
{
  drive.SetMotorPower target = new drive.SetMotorPower();

  target.Body.PowerTargetHex = update.Body.APowerTargetHex;
  _driveAPort.Post(target);
  target.Body.PowerTargetHex = update.Body.BPowerTargetHex;
  _driveBPort.Post(target);
  target.Body.PowerTargetHex = update.Body.CPowerTargetHex;
  _driveCPort.Post(target);
  target.Body.PowerTargetHex = update.Body.DPowerTargetHex;
  _driveDPort.Post(target);
  target.Body.PowerTargetHex = update.Body.EPowerTargetHex;
  _driveEPort.Post(target);
  target.Body.PowerTargetHex = update.Body.FPowerTargetHex;
  _driveFPort.Post(target);

  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask>
SetPowerMotorsTargetCommandHandler(SetMotorsPowerTargetCommand update)
{
  drive.SetMotorPower target = new drive.SetMotorPower();

  target.Body.PowerTargetHex = update.Body.KPowerTargetHex;
  _driveKPort.Post(target);
  target.Body.PowerTargetHex = update.Body.LPowerTargetHex;
  _driveLPort.Post(target);
  target.Body.PowerTargetHex = update.Body.MPowerTargetHex;
  _driveMPort.Post(target);
  target.Body.PowerTargetHex = update.Body.NPowerTargetHex;
  _driveNPort.Post(target);
  target.Body.PowerTargetHex = update.Body.OPowerTargetHex;
  _driveOPort.Post(target);
  target.Body.PowerTargetHex = update.Body.PPowerTargetHex;
  _drivePPort.Post(target);

  yield break;
}
```

CODE 85

Both services provide splitting string command and send a single command on single servo motor port.

5.6. Robovie Generic Twenty Two Servo Motor Service

This service is very important because it represents unique interface by *RobovieMBase* entity. This service manages a humanoid robot full. It receives a single or multi command and it provides to splitting command in to five services; two for arm services, two for leg services and one for body service. It has a five partner services shown in the following patch code:

```
[ServicePort("/roboviegeneric22smdrive", AllowMultipleInstances = false)]
private RobovieGeneric22SMDriveOperations _mainPort =
       new RobovieGeneric22SMDriveOperations();

[Partner("ArmLeft", Contract = armleft.Contract.Identifier,
       CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
armleft.RobovieArmLeftOperations _armleftPort =
   new armleft.RobovieArmLeftOperations();
armleft.RobovieArmLeftOperations _armleftPortNotify =
   new armleft.RobovieArmLeftOperations();

[Partner("ArmRight", Contract = armright.Contract.Identifier,
       CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
armright.RobovieArmRightOperations _armrightPort =
   new armright.RobovieArmRightOperations();
armright.RobovieArmRightOperations _armrightPortNotify =
   new armright.RobovieArmRightOperations();

[Partner("Body", Contract = body.Contract.Identifier,
       CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
body.RobovieBodyOperations _bodyPort =
   new body.RobovieBodyOperations();
body.RobovieBodyOperations _bodyPortNotify =
   new body.RobovieBodyOperations();

[Partner("LegLeft", Contract = legleft.Contract.Identifier,
       CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
legleft.RobovieLegLeftOperations _legleftPort =
   new legleft.RobovieLegLeftOperations();
legleft.RobovieLegLeftOperations _legleftPortNotify =
   new legleft.RobovieLegLeftOperations();

[Partner("LegRight", Contract = legright.Contract.Identifier,
       CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
legright.RobovieLegRightOperations _legrightPort =
  new legright.RobovieLegRightOperations();
legright.RobovieLegRightOperations _legrightPortNotify =
   new egright.RobovieLegRightOperations();
```

CODE 86

The service's scheme is shown in the following picture, where *SetRobovieCommandHandler* , *CalibrationHandler* are handler when the service receives a new message type request and Body, Arm Left, Arm Right, Leg Left and Leg Right are other services as specified in the precedent part. We considere only two types message handler because they are more important to the other handler.

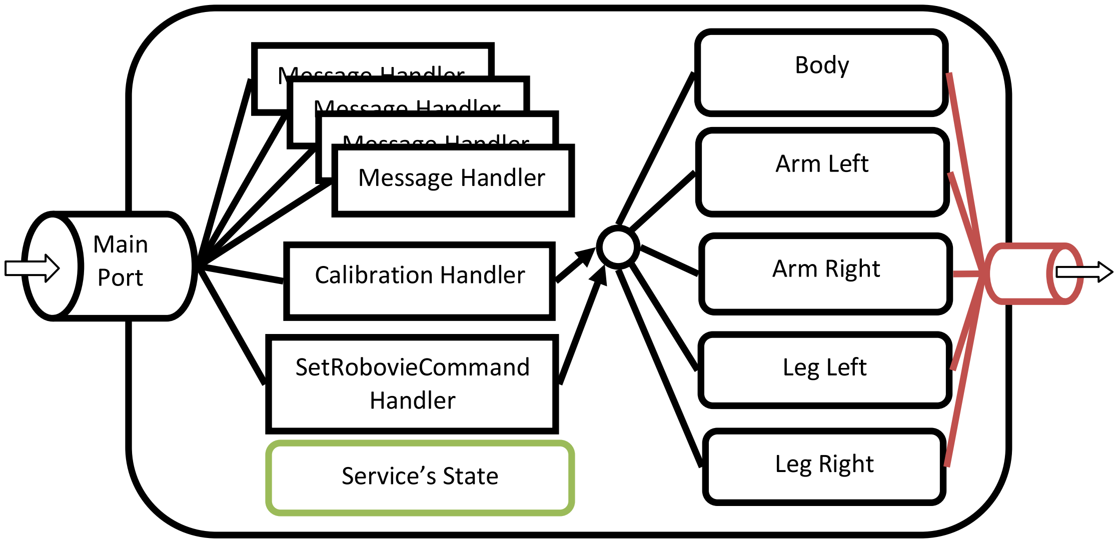


FIGURE 54

The Main Port can receive a list of message type. The complete *PortSet<>* is:

```
[ServicePort()]
public class RobovieGeneric22SMDriveOperations
  : PortSet<
    DsspDefaultLookup, DsspDefaultDrop,
    Get, HttpGet, Replace, Subscribe,
    SetJointTargetVelocity, SetJointTargetPose,
    SetJointTargetOrientation, SetJointTargetDegreesOrientation,
    SetRobovieCommand, RobovieSingleCommandCompleted, SetCalibration> { }
```

CODE 87

The not default messages specification is shown in the following code:

```
public class SetJointTargetVelocity
  : Update<SetJointTargetVelocityRequest, PortSet<DefaultUpdateResponseType, Fault>>
{ }
public class SetJointTargetPose
  : Update<SetJointTargetPoseRequest, PortSet<DefaultUpdateResponseType, Fault>>
{ }
public class SetJointTargetOrientation
  : Update<SetJointTargetOrientationRequest, PortSet<DefaultUpdateResponseType,
Fault>>
{ }
public class SetJointTargetDegreesOrientation
  : Update<SetJointTargetDegreesOrientationRequest, PortSet<DefaultUpdateResponseType,
Fault>>
{ }
public class SetRobovieCommand
  : Update<SetRobovieCommandRequest, PortSet<DefaultUpdateResponseType, Fault>>
{ }
public class RobovieSingleCommandCompleted
  : Update<RobovieSingleCommandCompletedRequest, PortSet<DefaultUpdateResponseType,
Fault>>
{ }
public class SetCalibration
  : Update<SetCalibrationRequest, PortSet<DefaultUpdateResponseType, Fault>>
{ }
```

Code 88

SetJointTargetVelocity used to set a joint's rotation velocity, *SetJointTargetPose* and *SetJointTargetOrientation* are used to set a joint's target pose and joint's target orientation.

SetRobovieCommand is used by service to receive a new command from the other services, it is not important how service type, but it must generate a valid string command for the humanoid robot. *SetCalibration* is used to calibrate servo motors in to equilibrium pose for each servo motors list. When one command is running the service waiting notifications by service's child then and just in this case the service sends a notification *RobovieSingleCommandCompleted* on main port to notify that current command has been run now has been completed; then the service can load next string command by *eprom* data structure.

The service has a private variable that represent robovie-m entity. This variable is used by manipulating entity in the Microsoft robotics studio simulator by this service. Its definition is the following:

```
RobovieMBase _entity;
```

When the service is running with simulation's manifest file, the _entity variable is not null value, but it is assigned with the following code:

```
void InsertEntityNotificationHandler(simengine.InsertSimulationEntity ins)
  {
    if (ins == null)
      return;

    _entity = (simengine.RobovieM.RobovieM)ins.Body;
    _entity.ServiceContract = Contract.Identifier;

    if(_entity!=null)
      SetServoMotorInitializeWithService();
}
void DeleteEntityNotificationHandler(simengine.DeleteSimulationEntity del)
{
  _entity = null;
  _jointLookup = null;
  _servoLookup = null;
}
```

Code 89

With *_entity* variable we manipulate Robovie entity into simulation. For removing the actual entity we call the following method code and we set _entity value to null:

```
void DeleteEntityNotificationHandler(simengine.DeleteSimulationEntity del)
{
  _entity = null;
  _jointLookup = null;
  _servoLookup = null;
}
```

Code 90

When the service is starting calls a *Start* method for initializing itself. The *Start* method is explicit in the following code:

```
protected override void Start()
{
  _notificationTarget = new simengine.SimulationEnginePort();

  // PartnerType.Service is the entity instance name.
  SimulationEngine.GlobalInstancePort.Subscribe(
    ServiceInfo.PartnerList,
    _notificationTarget
  );

  _armleftPort.Subscribe(_armleftPortNotify);
  _armrightPort.Subscribe(_armrightPortNotify);
  _bodyPort.Subscribe(_bodyPortNotify);
  _legleftPort.Subscribe(_legleftPortNotify);
  _legrightPort.Subscribe(_legrightPortNotify);
  …………
}
```

Code 91

First step is used for subscribing services, in fact we subscribe partner service with same services for its explicit notification. In the start method same *Activate.Arbier* are set to listen on the specific port some new messages. When we insert a new entity, *RobovieMBase* type, in the first time the service throws the following method:

```
void InsertEntityNotificationHandlerFirstTime(simengine.InsertSimulationEntity ins)
{
  InsertEntityNotificationHandler(ins);

  base.Start();

  MainPortInterleave.CombineWith(
   new Interleave(
    new TeardownReceiverGroup(

    ),
    new ExclusiveReceiverGroup(
     // Arbiter.Receive list
    ),
    new ConcurrentReceiverGroup(
      // Arbiter.Receive list
     )
   )
  );
}
```

Code 92

Where the list of *Arbiter.Receve* is a list of ports where it captures message notification from subscribed service. The complete list is shown in the following code:

```
Arbiter.Receive<simengine.InsertSimulationEntity>(true,
    _notificationTarget, InsertEntityNotificationHandler),
Arbiter.Receive<simengine.DeleteSimulationEntity>(true,
    _notificationTarget, DeleteEntityNotificationHandler),
Arbiter.Receive<SetJointTargetOrientation>(true,
    _mainPort, SetJointTargetOrientationNotificationHandler),
Arbiter.Receive<SetJointTargetDegreesOrientation>(true,
    _mainPort, SetJointTargetDegreesOrientationNotificationHandler),
Arbiter.Receive<SetJointTargetPose>(true,
    _mainPort, SetJointTargetPoseNotificationHandler),
Arbiter.Receive<SetJointTargetVelocity>(true,
    _mainPort, SetJointTargetVelocityNotificationHandler),
Arbiter.Receive<SetRobovieCommand>(true,
    _mainPort, SetRobovieSingleCommandNotificationHandler),
Arbiter.Receive<RobovieSingleCommandCompleted>(true,
    _mainPort, SetRobovieSingleCommandCompletedNotificationHandler),
Arbiter.Receive<SetCalibration>(true,
    _mainPort, CalibrationNotificationHandler)
```

Code 93

The precedent receive group is inserted into *ExclusiveReceiveGroup(...)* part of code. The concurrent group *ConcurrentReceiveGroup(...)* is explicit in the following code:

```
Arbiter.Receive<armleft.GPowerTargetChanged>(true,
    _armleftPortNotify, SetGPowerTargetChangedNotificationHandler),
Arbiter.Receive<armleft.HPowerTargetChanged>(true,
    _armleftPortNotify, SetHPowerTargetChangedNotificationHandler),
Arbiter.Receive<armleft.IPowerTargetChanged>(true,
    _armleftPortNotify, SetIPowerTargetChangedNotificationHandler),
Arbiter.Receive<armleft.JPowerTargetChanged>(true,
    _armleftPortNotify, SetJPowerTargetChangedNotificationHandler),

Arbiter.Receive<armright.QPowerTargetChanged>(true,
    _armrightPortNotify, SetQPowerTargetChangedNotificationHandler),
Arbiter.Receive<armright.RPowerTargetChanged>(true,
    _armrightPortNotify, SetRPowerTargetChangedNotificationHandler),
Arbiter.Receive<armright.SPowerTargetChanged>(true,
    _armrightPortNotify, SetSPowerTargetChangedNotificationHandler),
Arbiter.Receive<armright.TPowerTargetChanged>(true,
    _armrightPortNotify, SetTPowerTargetChangedNotificationHandler),

Arbiter.Receive<body.UPowerTargetChanged>(true,
    _bodyPortNotify, SetUPowerTargetChangedNotificationHandler),
Arbiter.Receive<body.VPowerTargetChanged>(true,
    _bodyPortNotify, SetVPowerTargetChangedNotificationHandler),

Arbiter.Receive<legleft.APowerTargetChanged>(true,
    _legleftPortNotify, SetAPowerTargetChangedNotificationHandler),
Arbiter.Receive<legleft.BPowerTargetChanged>(true,
    _legleftPortNotify, SetBPowerTargetChangedNotificationHandler),
Arbiter.Receive<legleft.CPowerTargetChanged>(true,
    _legleftPortNotify, SetCPowerTargetChangedNotificationHandler),
Arbiter.Receive<legleft.DPowerTargetChanged>(true,
    _legleftPortNotify, SetDPowerTargetChangedNotificationHandler),
Arbiter.Receive<legleft.EPowerTargetChanged>(true,
    _legleftPortNotify, SetEPowerTargetChangedNotificationHandler),
Arbiter.Receive<legleft.FPowerTargetChanged>(true,
    _legleftPortNotify, SetFPowerTargetChangedNotificationHandler),

Arbiter.Receive<legright.KPowerTargetChanged>(true,
    _legrightPortNotify, SetKPowerTargetChangedNotificationHandler),
Arbiter.Receive<legright.LPowerTargetChanged>(true,
    _legrightPortNotify, SetLPowerTargetChangedNotificationHandler),
Arbiter.Receive<legright.MPowerTargetChanged>(true,
    _legrightPortNotify, SetMPowerTargetChangedNotificationHandler),
Arbiter.Receive<legright.NPowerTargetChanged>(true,
    _legrightPortNotify, SetNPowerTargetChangedNotificationHandler),
Arbiter.Receive<legright.OPowerTargetChanged>(true,
    _legrightPortNotify, SetOPowerTargetChangedNotificationHandler),
Arbiter.Receive<legright.PPowerTargetChanged>(true,
    _legrightPortNotify, SetPPowerTargetChangedNotificationHandler),

Arbiter.Receive<armleft.TargetCompleted>(true, _armleftPortNotify,
  delegate(armleft.TargetCompleted update)
  {
    notification++;
    _mainPort.Post(new RobovieSingleCommandCompleted());
  }),
Arbiter.Receive<armright.TargetCompleted>(true, _armrightPortNotify,
  delegate(armright.TargetCompleted update)
  {
    notification++;
    _mainPort.Post(new RobovieSingleCommandCompleted());
  }),
Arbiter.Receive<body.TargetCompleted>(true, _bodyPortNotify,
  delegate(body.TargetCompleted update)
  {
    notification++;
    _mainPort.Post(new RobovieSingleCommandCompleted());
  }),

Arbiter.Receive<legleft.TargetCompleted>(true, _legleftPortNotify,
  delegate(legleft.TargetCompleted update)
  {
    notification++;
    _mainPort.Post(new RobovieSingleCommandCompleted());
  }),
Arbiter.Receive<legright.TargetCompleted>(true, _legrightPortNotify,
  delegate(legright.TargetCompleted update)
  {
    notification++;
    _mainPort.Post(new RobovieSingleCommandCompleted());
  })
```

Code 94

For each servo motor defined into service it can send servomotor's state notify. Last five *Arbiter.Receive* are used to notify when a single string command by a service is completed. When all partner services are completed, the principal service sends on main port a message notification to load new command by *eprom* data structure.

When service is listen on notify port, it needs a delegate for the execution code. Generally when we write:

```
Arbiter.Receive<legright.$PowerTargetChanged>(true,
    _portNotify, Set$PowerTargetChangedNotificationHandler),
```

Set$PowerTargetChangedNotificationHandler is a method that it has a generic code:

```
void Set$PowerTargetChangedNotificationHandler(using.$PowerTargetChanged update)
{
  if (_entity != null)
  {
    mainPort.Post(
       CreateTargetDegreesOrientation(
          ServoMotorName.SERVOMOTOR_$,
          update.Body.CurrentTargetValue));
  }
}
```

Code 95

Where $ letter's value can have from A to V value range. This concept is valid for all notification handlers existing in service's code.

The *eprom* data structure is a list of single command type and it is instanced when the service is instanced. The *eprom* is defined:

```
List<RobovieSingleStringCommand> _eprom = new List<RobovieSingleStringCommand>();
```

When the service starts the *eprom* is empty. When a new string command arrives on main port, service schedules it and it splits a single string command into list of single command, and it loads *eprom* with list string commands, and then it sends small command to dedicate partner services. The current command is scheduled with an index. The code representing this concept is shown in the following code:

```
void SetRobovieSingleCommandNotificationHandler(SetRobovieCommand update)
{
  _armleftPort.Post(new armleft.PowerOff());
  _armrightPort.Post(new armright.PowerOff());
  _bodyPort.Post(new body.PowerOff());
  _legleftPort.Post(new legleft.PowerOff());
  _legrightPort.Post(new legright.PowerOff());

  _eprom.Clear();

  if (update.Body.TargetCommand == String.Empty)
    return;

RobovieMultiStringCommands multi =
    new RobovieMultiStringCommands(update.Body.TargetCommand);

  foreach (String c in multi.ListCommands)
    _eprom.Add(new RobovieSingleStringCommand(c));

  RunSingleCommand(_eprom[_currentcommand]);
}
```

Code 96

When service has splitted string command it calls *RunSingleCommand(...)* method with current command parameter, usually this command is at the first position on the list. When a new multi command arrives on main port the first operation is to send for all partner services a *PowerOff* message for stopping all servo motors.

The *RunSingleCommand* method takes *RobovieSingleStringCommand* parameter and provides to get command information for a single servo motors and it creates a single *SetMotorPowerTargetCommand* for partner services that it manages robovie's single part.

```
void RunSingleCommand(RobovieSingleStringCommand command)
{
  armleft.SetMotorsPowerTargetCommand armL =
    new armleft.SetMotorsPowerTargetCommand();
  armL.Body.GPowerTargetHex = command.ServoMotorGHex;
  armL.Body.HPowerTargetHex = command.ServoMotorHHex;
  armL.Body.IPowerTargetHex = command.ServoMotorIHex;
  armL.Body.JPowerTargetHex = command.ServoMotorJHex;

  armright.SetMotorsPowerTargetCommand armR =
    new armright.SetMotorsPowerTargetCommand();
  armR.Body.QPowerTargetHex = command.ServoMotorQHex;
  armR.Body.RPowerTargetHex = command.ServoMotorRHex;
  armR.Body.SPowerTargetHex = command.ServoMotorSHex;
  armR.Body.TPowerTargetHex = command.ServoMotorTHex;

  body.SetMotorsPowerTargetCommand bodyC =
    new body.SetMotorsPowerTargetCommand();
  bodyC.Body.UPowerTargetHex = command.ServoMotorUHex;
   bodyC.Body.VPowerTargetHex = command.ServoMotorVHex;

  legleft.SetMotorsPowerTargetCommand legL =
    new legleft.SetMotorsPowerTargetCommand();
  legL.Body.APowerTargetHex = command.ServoMotorAHex;
  legL.Body.BPowerTargetHex = command.ServoMotorBHex;
  legL.Body.CPowerTargetHex = command.ServoMotorCHex;
  legL.Body.DPowerTargetHex = command.ServoMotorDHex;
  legL.Body.EPowerTargetHex = command.ServoMotorEHex;
  legL.Body.FPowerTargetHex = command.ServoMotorFHex;

  legright.SetMotorsPowerTargetCommand legR =
    new legright.SetMotorsPowerTargetCommand();
  legR.Body.KPowerTargetHex = command.ServoMotorKHex;
  legR.Body.LPowerTargetHex = command.ServoMotorLHex;
  legR.Body.MPowerTargetHex = command.ServoMotorMHex;
  legR.Body.NPowerTargetHex = command.ServoMotorNHex;
  legR.Body.OPowerTargetHex = command.ServoMotorOHex;
  legR.Body.PPowerTargetHex = command.ServoMotorPHex;

  _armleftPort.Post(new armleft.PowerOn());
  _armrightPort.Post(new armright.PowerOn());
  _bodyPort.Post(new body.PowerOn());
  _legleftPort.Post(new legleft.PowerOn());
  _legrightPort.Post(new legright.PowerOn());

  _bodyPort.Post(bodyC);
  _armleftPort.Post(armL);
  _armrightPort.Post(armR);
  _legleftPort.Post(legL);
  _legrightPort.Post(legR);
}
```

CODE 97

After having provided to send *PowerOn* message for power on single service and sends a single parts command to partner services.

When each partner services have completed internal operations and the services have sent message notification at the *Generic Twenty Two Servo Motor Service* must load a new command from *eprom*. It must control that all partner sent a completed message and then it executes the following code for loading a new command and runs it:

```
void SetRobovieSingleCommandCompletedNotificationHandler(RobovieSingleCommandCompleted
update)
{
  if (notification == 5)
  {
     base.SendNotification<RobovieSingleCommandCompleted>(_submgrPort, update);
     notification = 0;

     if (_eprom.Count == 0 | _eprom.Count == 1)
     {
       _currentcommand = 0;
       return;
     }

     _currentcommand++;
     int nextcommand = (_currentcommand % _eprom.Count);

      if (_currentSingleCommand != _eprom[nextcommand])
      {
        _currentcommand = nextcommand;
        _currentSingleCommand = _eprom[_currentcommand];
        RunSingleCommand(_currentSingleCommand);
      }
    }
}
```

CODE 98

The precedent method controls the number of notification and if it equals at five then sends completed message notification and resets notification number and it increments current command by one. The new current command is the next command in to *eprom*. This number must be modular of number string single command on *eprom* because humanoid robot must be executing until the external service sends a stop type message. The external service can be any type.

The most important thing is that the service generates a valid string command for the *Generic twenty Two Servo Motor service*. This service can receive on main port the same type of messages. Same examples are shown in the following code with methods definition. The methods are executed when on main port a message type has been sent. The following code explicit the precedent concept:

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask>
SetRobovieSingleCommandHandler(SetRobovieCommand update)
{
  SetRobovieSingleCommandNotificationHandler(update);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask>
RobovieSingleCommandCompletedHandler(RobovieSingleCommandCompleted update)
{
  SetRobovieSingleCommandCompletedNotificationHandler(update);
   ield break;
}

[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> CalibrationHandler(SetCalibration update)
{
  CalibrationNotificationHandler(update);
  yield break;
}
```

CODE 99

There are default methods handler contained in all the other services and that are expressed in the following code:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
ublic IEnumerator<ITask> GetHandler(Get get)
```

```
{
  get.ResponsePort.Post(_state);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> ReplaceHandler(Replace replace)
{
  replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> HttpGetHandler(HttpGet httpget)
{
  httpget.ResponsePort.Post(new HttpResponseType(_state));
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> SubscribeHandler(Subscribe subscribe)
{
  Activate(Arbiter.Choice(
   SubscribeHelper(_submgrPort, subscribe.Body, subscribe.ResponsePort),
    delegate(SuccessResult success)
    {
     _submgrPort.Post(new submgr.Submit(
      subscribe.Body.Subscriber, DsspActions.UpdateRequest, _state, null));
    },
    delegate(Exception ex) { LogError(ex); }
  ));
  yield break;
}
```

Code 100

The handlers used for: *Get* and *HttpGet* by get browser requests, *Replace* by replace request, *Subscribe* by to subscribing with another service.

5.7. Robovie Simulation Service

Robovie simulation service is a particular service used for rending and managing humanoid robot on the Microsoft Robotics Studio simulator. Same important information is not to have own type messages, but it useds *Robovie Twenty Two Servo Motor Drive Service*'s type messages.

This service uses following namespace in addition to the standard namespaces for the simulation:

```
using motor = Robotics.RobovieGeneric22SMDrive.Proxy;
```

A service's scheme is showed into following image:

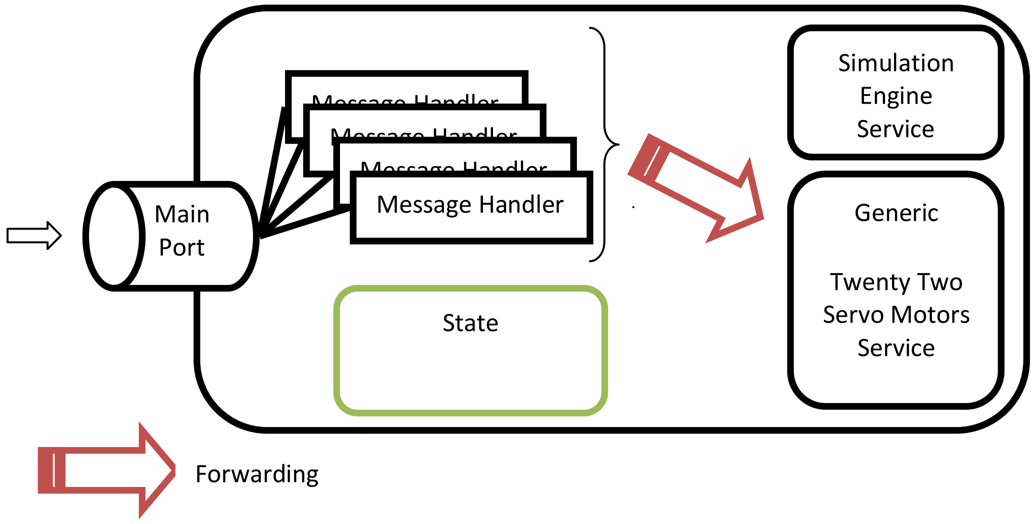


FIGURE 55

The service has a partner *SimulationEnginePort* that it is fundamental for the correct simulation execution and a main port that receives same messages and it makes the forwarding messages to *Generic Twenty Two Servo Motors Service*. The following code shown this concept:

```
[Partner("Engine", Contract = engineproxy.Contract.Identifier,
                    CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
private SimulationEnginePort _engineStub = new SimulationEnginePort();

[ServicePort("/roboviesimulation", AllowMultipleInstances = false)]
private RobovieSimulationOperations _mainPort = new RobovieSimulationOperations();

private motors.RobovieGeneric22SMDriveOperations _servomotorPort;
```

CODE 101

The *_servomotorPort* will use to make the forwarding on its own port. A message the service can receive on main port.

```
[ServicePort]
public class RobovieSimulationOperations
: PortSet<
  DsspDefaultLookup, DsspDefaultDrop, Get, Replace,
  motor.SetJointTargetDegreesOrientation,
  motor.SetRobovieCommand,
  motor.SetCalibration>
{

}
```

Code 102

The messages type that start with *motor* are messages type defined into *Robovie Generic Twenty Two Servo Motor Service*. When service starts it's running Start method. In the following code shown the Start method:

```
protected override void Start()
{
  base.Start();
  SetupCamera();
  PopulateWorld();
}
```

Code 103

The method calls some other methods. The *base.Start()* is used to initialize base class, *SetupCamera* is used to set a camera on the simulation and in the following code is shown the method's code:

```
void SetupCamera()
{
  CameraView view = new CameraView();
  view.EyePosition = new Vector3(-55.60f, 14.83f, -0.3f);
  view.LookAtPoint = new Vector3(-6.28f, 14.58f, -0.19f);
  SimulationEngine.GlobalInstancePort.Update(view);
}
```

Code 104

The *PopulateWorld* method is used to insert the entities on the simulation. It's called *AddSky* method the one used to insert a sky texture and *AddGround* to insert ground texture on the simulation. In the following code the method is shown:

```
void PopulateWorld()
{
  AddSky();
  AddGround();
  SpawnIterator<String, Vector3>("RobovieM", new Vector3(0, 0, 0), AddRobovieM);
}
```

Code 105

An important method is *SpawnIterator* that is used insert entity builds on the simulation. *SpawnIterator* takes how arguments: an entity's name, an entity's initial position and a delegate to the execution of the code and initialize it. The code method is shown below:

```
IEnumerator<ITask> AddRobovieM(String name, Vector3 position)
{
  // Create an instance of our custom Robovie-M.
  RobovieM _entity = new RobovieM(name, position);

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

  // create simulated arm service
  DsspResponsePort<CreateResponse> resultPort = CreateService(
    motors.Contract.Identifier,
    Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
      "http://localhost/" + _entity.State.Name));

  // asynchronously handle service creation result.
  yield return Arbiter.Choice(resultPort,
   delegate(CreateResponse rsp)
   {
     _servomotorPort = ServiceForwarder<motors.RobovieGeneric22SMDriveOperations>(
       rsp.Service
     );
      _servomotorPort.Subscribe(_mainPort);
   },
   delegate(Fault fault)
    {
      LogError(fault);
    }
  );

  if (_servomotorPort == null)
    yield break;

  do
  {
    yield return Arbiter.Receive(false,
      TimeoutPort(1000),
      delegate(DateTime signal) { });

     yield return Arbiter.Choice(
       _servomotorPort.Get(new GetRequestType()),
       delegate(motors.RobovieGeneric22SMDriveState state)
       {
         _cachedDriveState = state;
       },
       delegate(Fault f)
       {
         LogError(f);
       });
```

```
        // exit on error
        if (_cachedDriveState == null)
          yield break;
   } while (_cachedDriveState.Joints == null);
}
```

Code 106

In the initial code entity's type *RobovieM* is instanced, than it's insert in the simulation and creates service for the simulation. The code listens to the result port service, when service is created it assigns at *_servomotorPort* an instance of service. All requests on the main port are the forwarding on this port. The service is listening on main port until *_servomotorPort* has a valid value. Service handler is very simple because it posts messages on the servo motor port.

The simple code of this concept is shown in the following code:

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> SetJointTargetDegreesOrientationHandler(
    motors.SetJointTargetDegreesOrientation update)
{
  _servomotorPort.Post(update);
  yield break;
}

ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> SetRobovieSingleCommandHandler(
    motors.SetRobovieCommand update)
{
  _servomotorPort.Post(update);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public virtual IEnumerator<ITask> SetCalibrationHandler(
    motors.SetCalibration update)
{
  _servomotorPort.Post(update);
  yield break;
}
```

Code 107

For each message types an explicit service handler is defined, so that the service can satisfy every type of requests.

Now we show how the generic methods for adding sky and adding ground are on the project's code. The methods are very similar for the other simulation of the service and they represents a standard code implementation. The code is written by Microsoft Robotics Studio group.

```
/// <summary>+
/// adds sky into simulation engine
/// </summary>
void AddSky()
{
  SkyEntity sky = new SkyEntity("sky.dds", "sky_diff.dds");
  SimulationEngine.GlobalInstancePort.Insert(sky);

  LightSourceEntity sun = new LightSourceEntity();
  sun.State.Name = "Sun";
  sun.Type = LightSourceEntityType.Directional;
  sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
  sun.Direction = new Vector3(0.5f, -.75f, 0.5f);
  SimulationEngine.GlobalInstancePort.Insert(sun);
}

/// <summary>
/// adds ground into simulation engine
/// </summary>
void AddGround()
{
  HeightFieldEntity ground = new HeightFieldEntity(
    "simple ground", // name
    "03RamieSc.dds", // texture image
    new MaterialProperties("ground",
      0.2f, // restitution
      0.5f, // dynamic friction
      0.5f) // static friction
    );
  SimulationEngine.GlobalInstancePort.Insert(ground);
}
```

Code 108

The method *SetupCamera* has been written by Microsoft Robotics Studio group; in this thesis it has been modified accordingly to our requirements. The camera works dependently on the position of the humanoid robot position and of the scale factor of the simulation.

5.8. RobovieController User Interface Service

This *RobovieController User Interface Service* is a particular service. It uses a graphic user interface for the manipulation of a list of joints position and communicates with humanoid robot on the simulation service. The service is not important because it belongs to a group of generic services that generate a valid string commands for humanoid robot, real robot or simulated robot. A scheme for this service is shown in the following picture with the same connection between user interface and the real service.

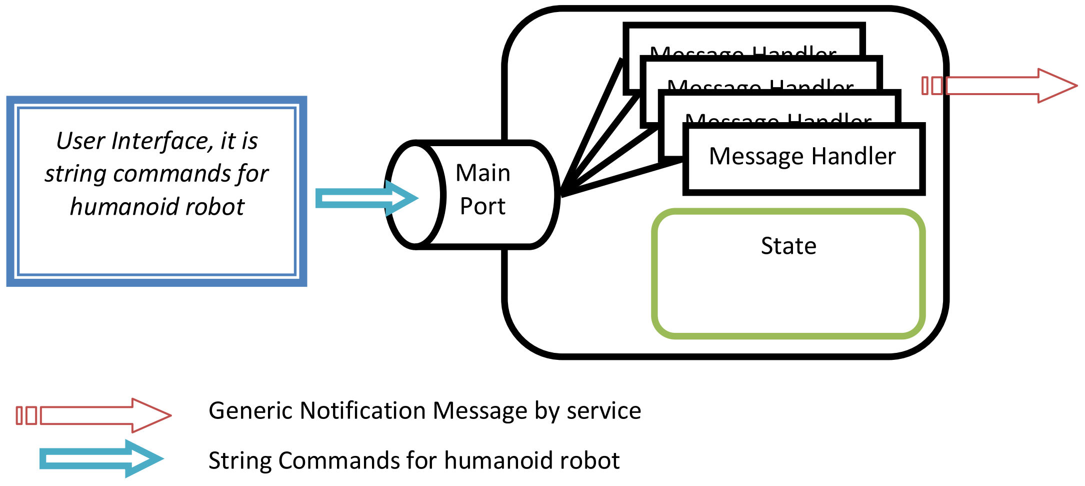

User Interface, it is string commands for humanoid robot
Main Port
Message Handler
Message Handler
Message Handler
Message Handler
State
Generic Notification Message by service
String Commands for humanoid robot

FIGURE 56

The service Main Port can receive a list of message types, the list is shown in the following patch code:

```
public class RobovieControllerUIOperations : PortSet<
    DsspDefaultLookup, DsspDefaultDrop,
    Get, HttpGet, Subscribe, Replace,
    RobovieTargetCommand,
    PowerOnServoMotor, PowerOffServoMotor,
    TargetCalibration>
{

}
```

CODE 109

Where *DesspDefaultLookup*, *DsspDefaultDrop*, *Get*, *HttpGet*, Subscribe and *Replace* are standard messages type discussed in the recent part of this thesis; and *RobovieTargetCommand* is a valid string command for humanoid robot, *PowerOnMotor*, *PowerOffMotor* are used to send a power on and power off command request. *TargetCalibration* is used for the calibration of the servomotor list on the simulation request.

In the following there are message types but there aren't message types defaulting.

```
public class RobovieTargetCommand
   : Update<RobovieTargetCommandRequest, PortSet<RobovieControllerUIState, Fault>> { }

public class PowerOnServoMotor
   : Update<PowerOnServoMotorRequest, PortSet<RobovieControllerUIState, Fault>> { }

public class PowerOffServoMotor
   : Update<PowerOffServoMotorRequest, PortSet<RobovieControllerUIState, Fault>> { }

public class TargetCalibration
   : Update<TargetCalibrationRequest, PortSet<RobovieControllerUIState, Fault>> { }
```

Code 110

When a valid message type is received on the main port, it is schedules correctly with an execution method:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> SetRobovieTargetCommandHandler(
      RobovieTargetCommand command
){
  SendNotification<RobovieTargetCommand>(_submgrPort, command);
  command.ResponsePort.Post(_state);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> PowerOnServoMotorHandler(PowerOnServoMotor update)
{
  SendNotification <PowerOnServoMotor>(_submgrPort, update);
  update.ResponsePort.Post(_state);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> PowerOffServoMotorHandler(PowerOffServoMotor update)
{
  SendNotification<PowerOffServoMotor>(_submgrPort, update);
  update.ResponsePort.Post(_state);
  yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> TargetCalibrationHandler(TargetCalibration update)
{
  SendNotification<TargetCalibration>(_submgrPort, update);
  update.ResponsePort.Post(_state);
  yield break;
}
```

Code 111

The method sends a notification when a message type arrives correctly on to main port so that begin to schedule.

An important method is *Start* method. The method throws a *MainWindows* to interact with service. When service opens a *WindowsForm*, it must be specified as in the following code:

```
MainWindows _mainwindows = null;
protected override void Start()
{
  base.Start();

  //set specificy Rendering mode
  Application.SetCompatibleTextRenderingDefault(false);
  Application.EnableVisualStyles();

  // Add service specific initialization here.
  WinFormsServicePort.Post(
    new Microsoft.Ccr.Adapters.WinForms.RunForm(
      delegate()
      {
        _mainwindows = new MainWindows(
          ServiceForwarder<RobovieControllerUIOperations>(ServiceInfo.Service),
          _state
        );
        return _mainwindows;
      }
    )
  );
}
```

Code 112

The method, in this case, is running graphics application and is sending messages type for to interact with this service.

MainWindows Application

The graphic application, *MainWindows*, has been written interactive with *Generic Twenty Two Servo Motor Drive* service.

Application has twenty two control manipulator twenty two servo motors and a control for sets execution speed. It subdivisions consist in five parts: left arm control, right arm control, body control, left leg control and right leg control. For each part there are a number of defined controls for the manipulation of the number servo motors. Each control contains a track bar that can possibly rotate a single servo on clockwise or anticlockwise.

The rotation typology is defined in to servo motor specification. The control execution speed is used to set a speed value that can 00 represent minimum value and FF represent maximum value. The default value is 0F.

The control has text and a numeric bar, first is used viewing a power target request, and the second sub control we can modify step to step the servo motor request. All servo motor's positions are definite for humanoid robot poses, we can load command on the *eprom* and run it.

After a small introduction on *MainWindows* application, now we will describe application's structure. Firstly we talk about *MainWindows*'s public constructor:

```
public MainWindows(
    RobovieControllerUIOperations mainport,
    RobovieControllerUIState state)
{
  this._mainport = mainport;
  this._state = state;
  InitializeComponent();
  Init();
}
```

Code 113

MainWindows class takes two parameters, *mainport* is service's *Main Port*, this parameter is used for the communication of the service and the state parameter is service's state.

When user interface throws an event, it uses service communication, it sends a message type on *mainport*, so that the service can receive a new message and it satisfies request

The following scheme represents structure:

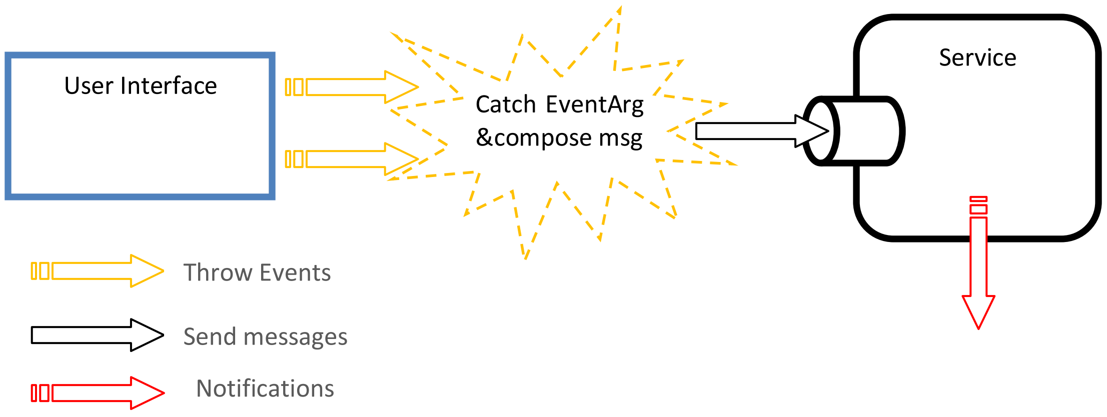


Figure 57

The most important part in the combination between application and service is when the application runs string command.

The current string command is defined by control value representing joint rotation position. The following code shows a single run target command request on:

```
void _singleTargetCommand_toolStripButton_Click(object sender, EventArgs e)
{
  if (MainWindowsTypeMode == TypeMode.RunMode)
    CreateNewSingleCommad();
}

private void CreateNewSingleCommad()
{
  RobovieTargetCommand target = new RobovieTargetCommand();
  target.Body.TargetCommand = GetCommandString();
  _mainport.Post(target);
}

public String GetCommandString()
{
  String command = String.Format(
   "@{0}" +
   "A{1}B{2}C{3}D{4}E{5}F{6}" +
   "G{7}H{8}I{9}J{10}" +
   "K{11}L{12}M{13}N{14}O{15}P{16}" +
   "Q{17}R{18}S{19}T{20}" +
   "U{21}V{22}" +
   ",+1!01",

  _speedControl.CustomCommandHex,

  _smcA.CustomCommandHex, _smcB.CustomCommandHex, _smcC.CustomCommandHex,
  _smcD.CustomCommandHex, _smcE.CustomCommandHex, _smcF.CustomCommandHex,

  _smcG.CustomCommandHex, _smcH.CustomCommandHex,
  _smcI.CustomCommandHex, _smcJ.CustomCommandHex,

  _smcK.CustomCommandHex, _smcL.CustomCommandHex, _smcM.CustomCommandHex,
   _smcN.CustomCommandHex, _smcO.CustomCommandHex, _smcP.CustomCommandHex,

  _smcQ.CustomCommandHex, _smcR.CustomCommandHex,
  _smcS.CustomCommandHex, _smcT.CustomCommandHex,

  _smcU.CustomCommandHex, _smcV.CustomCommandHex);

  return command;
}
```

CODE 114

The application can possibly send a calibration command to calibrate all servo motors on the simulation.

This process uses a file, where it is defining a calibrate information and it's reading file and sending information on the service's main port. The following code represents this concept:

```
void _calibrationMotortoolStripButton1_Click(object sender, EventArgs e)
{
  if (_mode == TypeMode.RunMode)
  {
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Filter = "Robovie Command (*.calib) | *.calib";

    if (ofd.ShowDialog() == DialogResult.OK)
    {
      StreamReader reader = new StreamReader(ofd.FileName);

      TargetCalibration calibration = new TargetCalibration();
      calibration.Body.TargetCalibration = new List<String>(22);

      for (int i = 0; i < 22; i++)
      {
         calibration.Body.TargetCalibration.Add("");
         calibration.Body.TargetMax.Add("");
         calibration.Body.TargetMin.Add("");
      }

      String line = null;
      while ((line = reader.ReadLine()) != null)
      {

        if (line.StartsWith(@"#") |
            line.StartsWith(@"//") |
            line == String.Empty)
          continue;

      try
      {
        String[] linesplit = line.Split(new String[] { " " },
StringSplitOptions.None);

        int zero = int.Parse(linesplit[0],
System.Globalization.NumberStyles.HexNumber);
        int min = int.Parse(linesplit[1],
System.Globalization.NumberStyles.HexNumber);
        int max = int.Parse(linesplit[2],
System.Globalization.NumberStyles.HexNumber);
         int idx = int.Parse(line.Split(new String[] { " " },
StringSplitOptions.None)[3]);

         _motors[idx].ScrollBar.Value = zero;
         _motors[idx].NumericBar.Value = zero;
         _motors[idx].TextBoxHex.Text = linesplit[0].ToUpper();

         calibration.Body.TargetCalibration[idx] = linesplit[0].ToUpper();
         calibration.Body.TargetMin[idx] = linesplit[1].ToUpper();
         calibration.Body.TargetMax[idx] = linesplit[2].ToUpper();
        }
        catch (Exception ex)
```

```
            {
              MessageBox.Show(ex.Message, ex.Source, MessageBoxButtons.OK);
            }
          }

          _iscalibration = true;
          reader.Close();

          _mainport.Post(calibration);
        }
        else
          return;
      }
    }
  }
```

CODE 115

The method reads same information by file, how is the equilibrium position, maximum torque and minimum torque. It creates a *TargetCalibration* message and sends it on service's main port.

The application has two running modes: *Edit Mode* and *Run Mode*.

- ***Run Mode*** is used to generate string command at application runtime. In fact we move some graphics control and send current command on the service's main port so the humanoid robot on the simulation changed its position or its state.
- ***Edit Mode*** is used to create a multi string command and save it on file. The file's name is specified by user. When we create a multi command, we can run it. To run a multi command is very simple, we can load on the *eprom* a multi string command using an user interface by the selection of a file.

An example to graphics control for a single servo motor execution is shown in the following image:

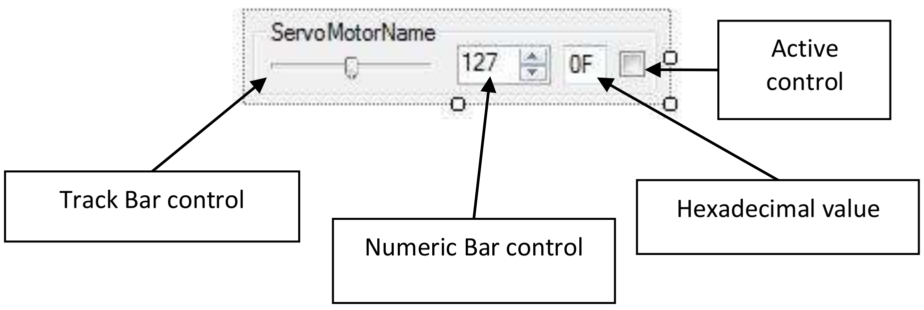


FIGURE 58

Active control uses activate control, so it sets a position and runs servo motor dedicated to it.

A collection control to manipulate arms or legs or body is shown in the following picture:

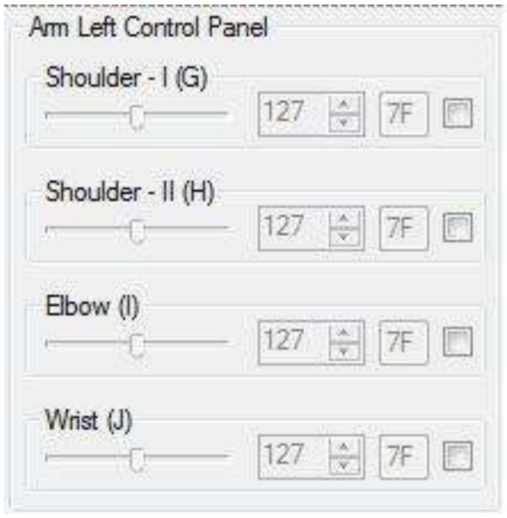
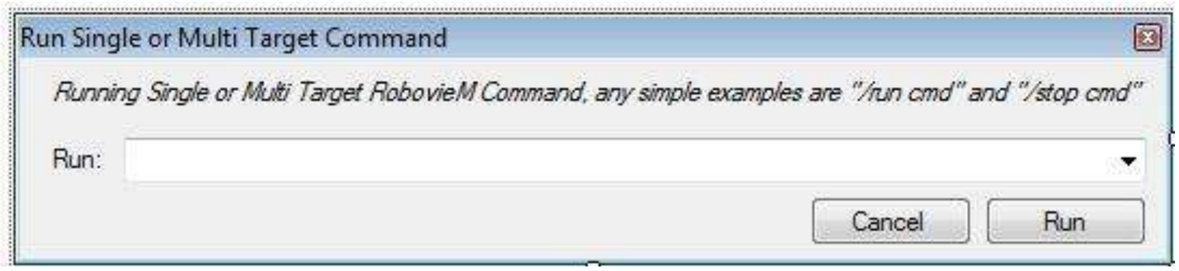


FIGURE 59

A single control manages a single servo motor on the simulation. In this case a control collection specified manages a humanoid robot's arm left. The application can be possible creating a multi command with apposite user interface, and save it on the file. It's running with a specific windows dialog, as shown here:



Running Single or Multi Target RobovieM Command, any simple examples are "/run cmd" and "/stop cmd"

FIGURE 60

A valid command has a valid syntax. The command must be so customed: */run cmd1 cmd2 cmd3 ....cmdn;* where /run is a specific command, in this case, we require a run list command, cmd1, cmd2 and etc. when we have a list commands, the application composes it on single command but its execution is sequential. Firstly runs a cmd1 then cmd2 until the last command.

There is a specification *cmdx* must be valid file's name. When we create a new command, it saves with this name type: *robovie.cmdx.rcmd*. When we want to run it, we require *cmdx* only. We can stopping a string command with /stop command and command's name. When we want to create a custom command, we set on the *edit mode* the own application.

The Edit Mode selection, it opens a dialog that can capture actually joints position for composing a new command and save it. A window dialog is shown here:

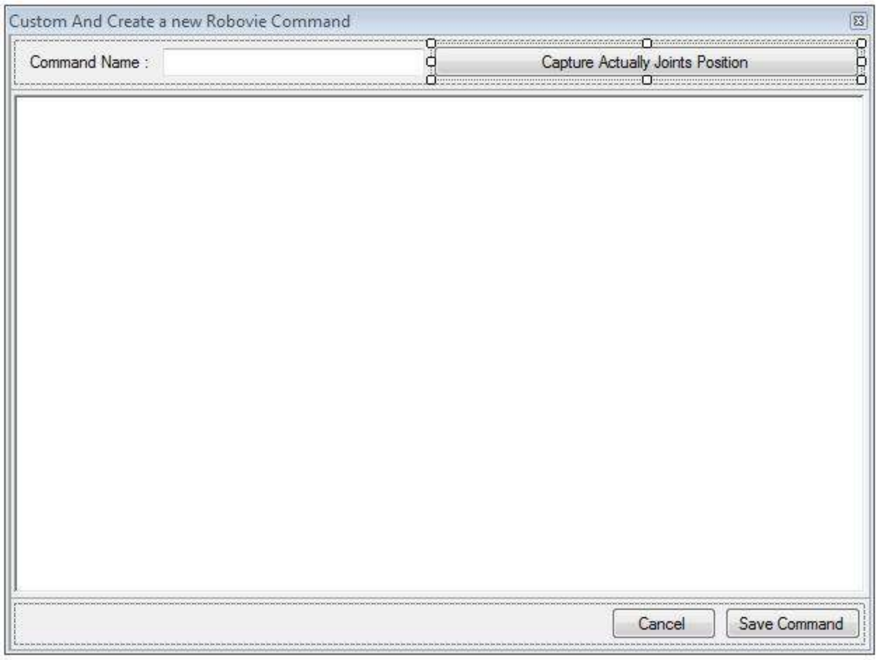

Custom And Create a new Robovie Command
Command Name :
Capture Actually Joints Position
Cancel
Save Command

FIGURE 61

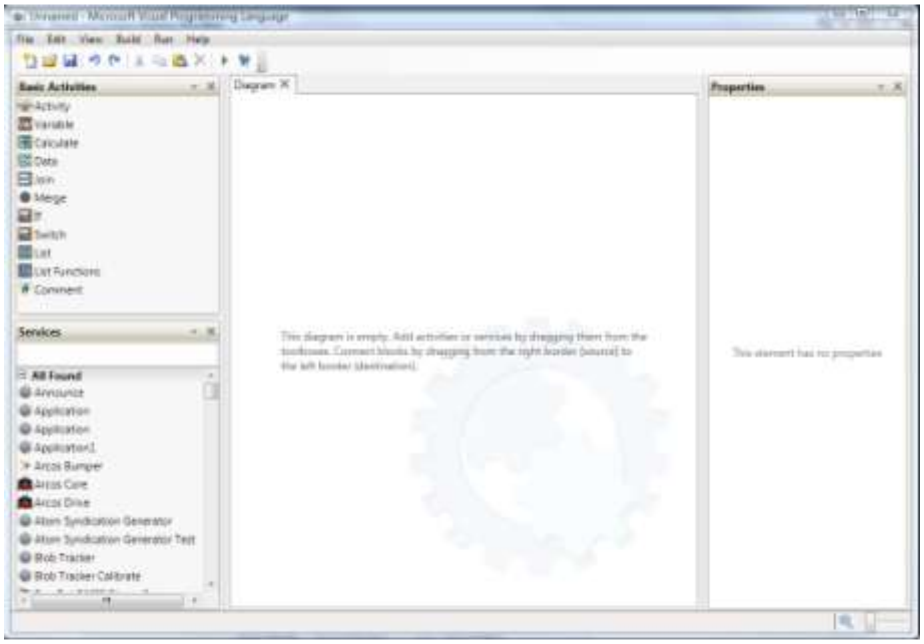
Where: command's name is a simple name as *cmdx*. *The Capture Actually Joints Position* button uses to capture the actually joints position on the control graphics. The number of single position is not arbitrary, we can create a new command with 10, 100 and etc custom joints position and save it. *Save Command* button uses to save command created.

Chapter 6

Robovie-M Visual Programming Language Model

6.1. Introduction

In this chapter we will treat how the Visual Programming Language application is used. Creating and composing a robotic application is very simple. In the previous chapter all services have been described. When a service is compiled, it inserts in the visual programming language's list of services section. In the following picture we will show the VPL user interface. The application has three important sections: *Basic Activities*, *Services* and *Diagram*. The *Basic Activities* contains a component base list, same examples are: Variable, Calculate, Data, Join, Merge and etc. *The Services* contains a service list. These services are inserted in the *bin/* Microsoft Robotics Studio's directory. A service compiled, but its assembly is not inserted in this specific directory we don’t view it on the list of services. The *Diagram* is where we can design our application.



This diagram is empty. Add activities or services by dragging them from the toolboxes. Connect blocks by dragging from the right border (source) to the left border (destination).

This element has no properties

FIGURE 62

The *Services* section allows the search of a specific service in the list of services. We want to seek all services that have a substring *“robovie”* on the service’s name. In the following picture results of the searched are shown.

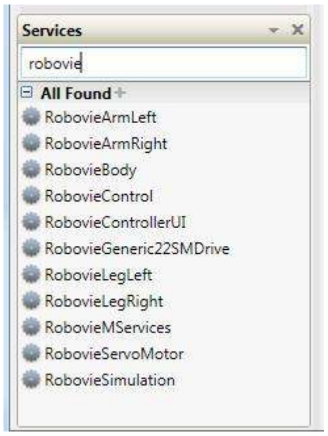


FIGURE 63

To use a service is very simply, we must select a service and drag in Diagram section. We can connect two or more services simply dragged from the specific service’s connection point to destination service’s connection point. This operation allows to subscribe two or more services. After we have dragged the *RobovieGeneric22SMDrive* service on the diagram, it views how shown in the following picture:

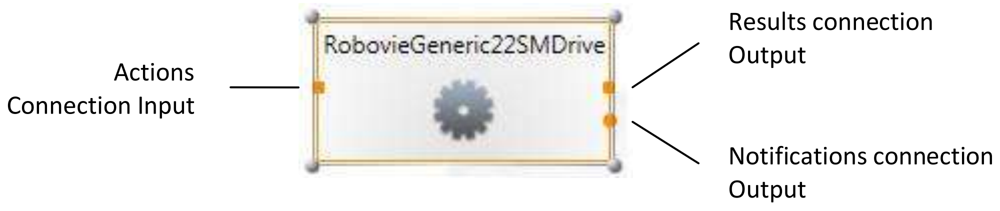
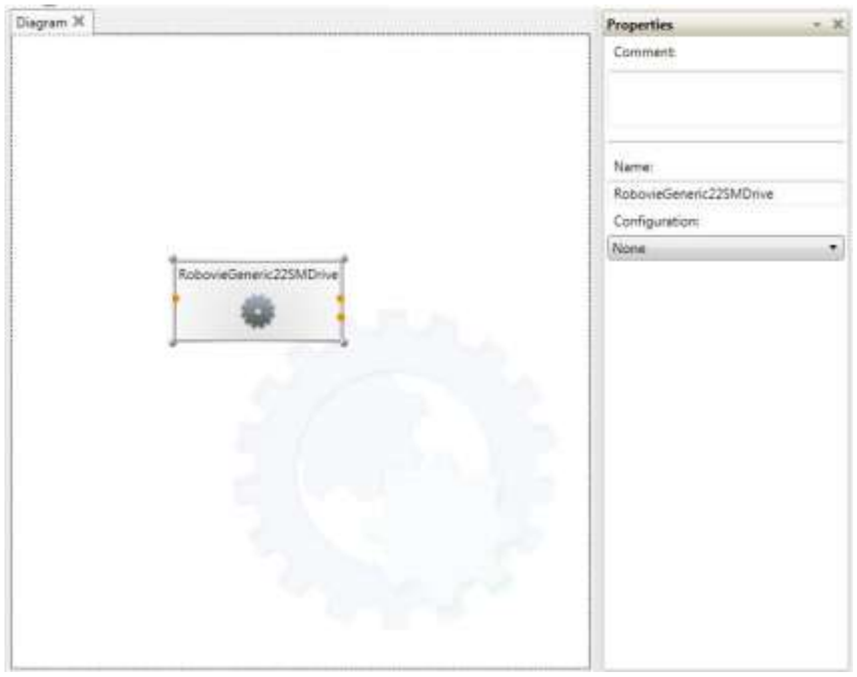


FIGURE 64

6.2. Robovie-M Application Model

Now we will describe an example of the application that uses the service compiled in the previous section. Now, we must use two services typology, *Robovie Generic Twenty Two Servo Motor Drive Service* and *Robovie Controller User Interface Service*. The second service is used to generate a single or multi string command and sends notification message on the Robovie *Generic Twenty Two Servo Motor Drive Service*'s main port by which *SetCommandTarget* message typology. The first service provides to consume message received and runs the string command in the simulation. The first service uses *Robovie Simulation Service*'s manifest by which the service and the simulation service are composed in a single service. In the following picture is shown that the *Robovie Generic Twenty Two Servo Motor Drive Service* uses the simulation's manifest.



Comment:

Name: RobovieGeneric22SMDrive

Configuration: None

FIGURE 65

After we have dragged *Robovie Generic 22 SM Drive service* on the *Diagram*, we double click on the service with mouse's left button and set configuration on *Use a manifest*, as shown in the following picture:

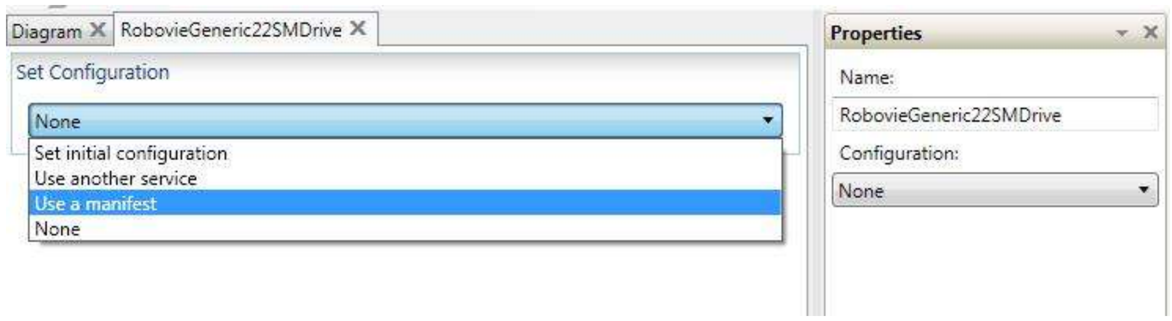
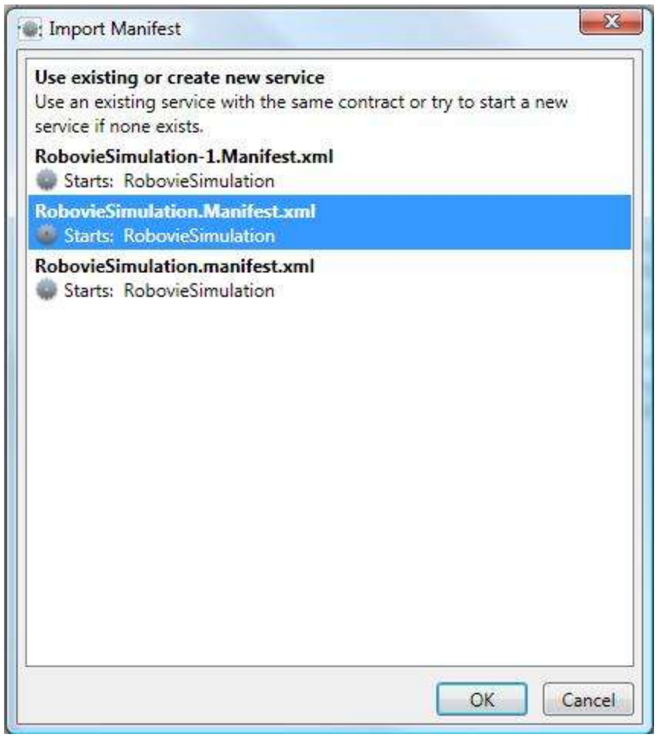


FIGURE 66

Now we import manifest file with clicks on "Import Manifest", and select manifest from available manifest shown in the list of manifest. In the following picture is shown a list of manifest available:



Use existing or create new service
Use an existing service with the same contract or try to start a new service if none exists.

RobovieSimulation-1.Manifest.xml
Starts: RobovieSimulation

RobovieSimulation.Manifest.xml
Starts: RobovieSimulation

RobovieSimulation.manifest.xml
Starts: RobovieSimulation

FIGURE 67

A manifest selects and presses OK button to confirm our choice. A manifest contains a list of services that are started together.

Now we drag and inset a *Robovie Controller User Interface Service* on Diagram section for connecting these entities with a notification message. When we connect two entities, with Visual Programming Language, we want that the destination service connection point is subscribed for notifications message type from the source service connection point.

In the following picture is shown this concept:

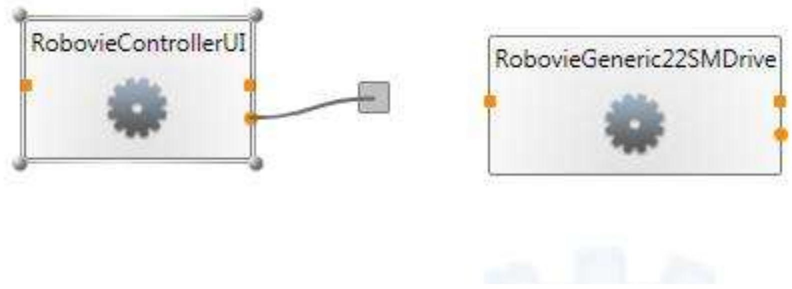


FIGURE 68

Doing like this, we want to send a notification message, when it exists and it is generated, from *Robovie Controller UI* to *Robovie Generic 22 SM Drive*. In this case, *Robovie Generic 22 SM Drive* is subscribed at *Robovie Controller UI*. Now we must choices a connection typology selecting source and destination message type. We are doing this with the following dialog:

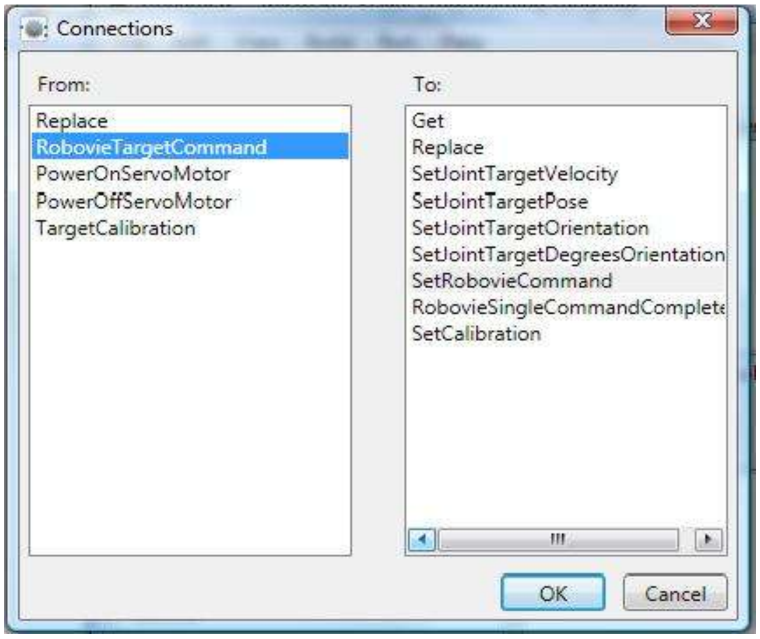


FIGURE 69

RobovieTargetCommand and *SetRobovieCommand* are described in the previous chapter when we have talked about Main Port message type that can be received. When a connection is confirmed a *Data Connections* value must be selected. The following picture represents this concept with a window dialog:

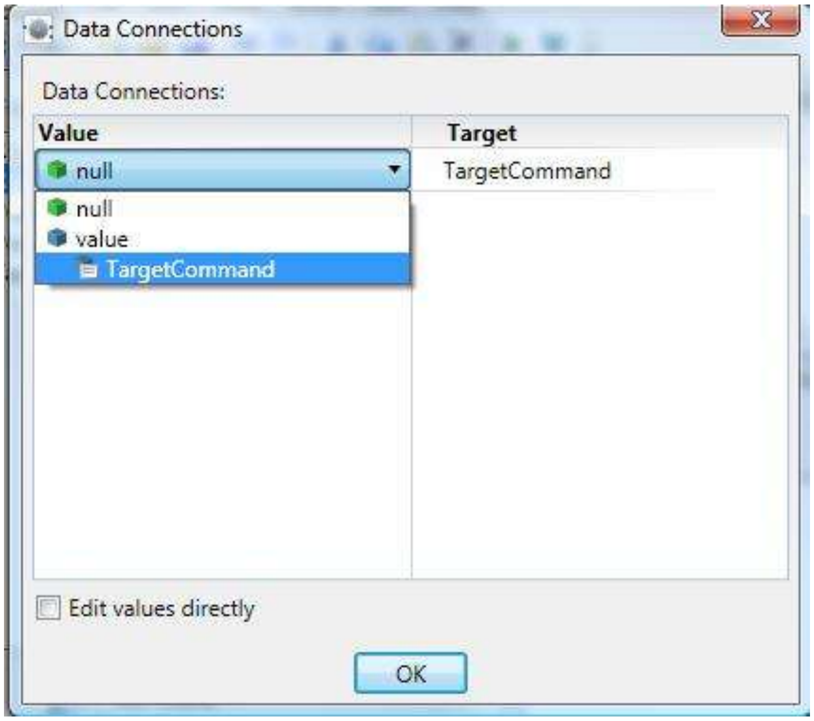


FIGURE 70

We can edit value directly too. The final result is shown in the following picture. Service has been connected and when *Robovie Control UI* generates a valid string command notifies with a message to *Robovie Generic 22 SM Drive*.

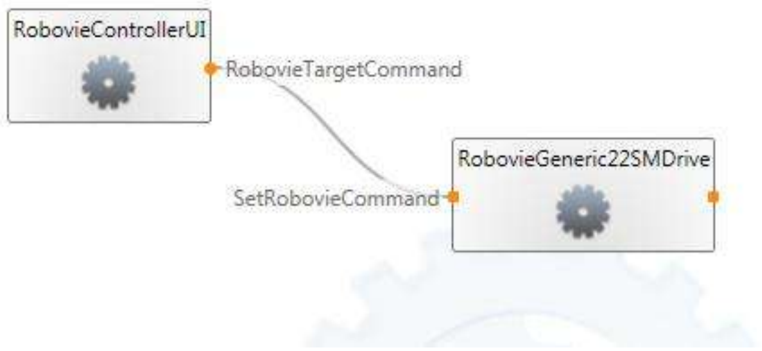


FIGURE 71

Now we can run the application with clicks on *Run* in the VPL's main menu and select *Start* or *Start Debug.* Doing this when the application is started three dialogs or three main windows will be shown: *Prompt*, *Controller User Interface* and *Simulation Engine*.

6.3. The Application Model Custom

When we talk of the application model, we think that these are modular with the other applications. In fact, we build an application that is not used just in this case. Robovie Generic 22 SM Drive is a particular case of service, because it uses to manipulate a specific humanoid robot entity only. With service created in the previous section we can builds a modular application.

In the following part we can describe same examples that demonstrate this concept. Now we want to manage a robot with two servo motors only. This concept is shown in the following picture:

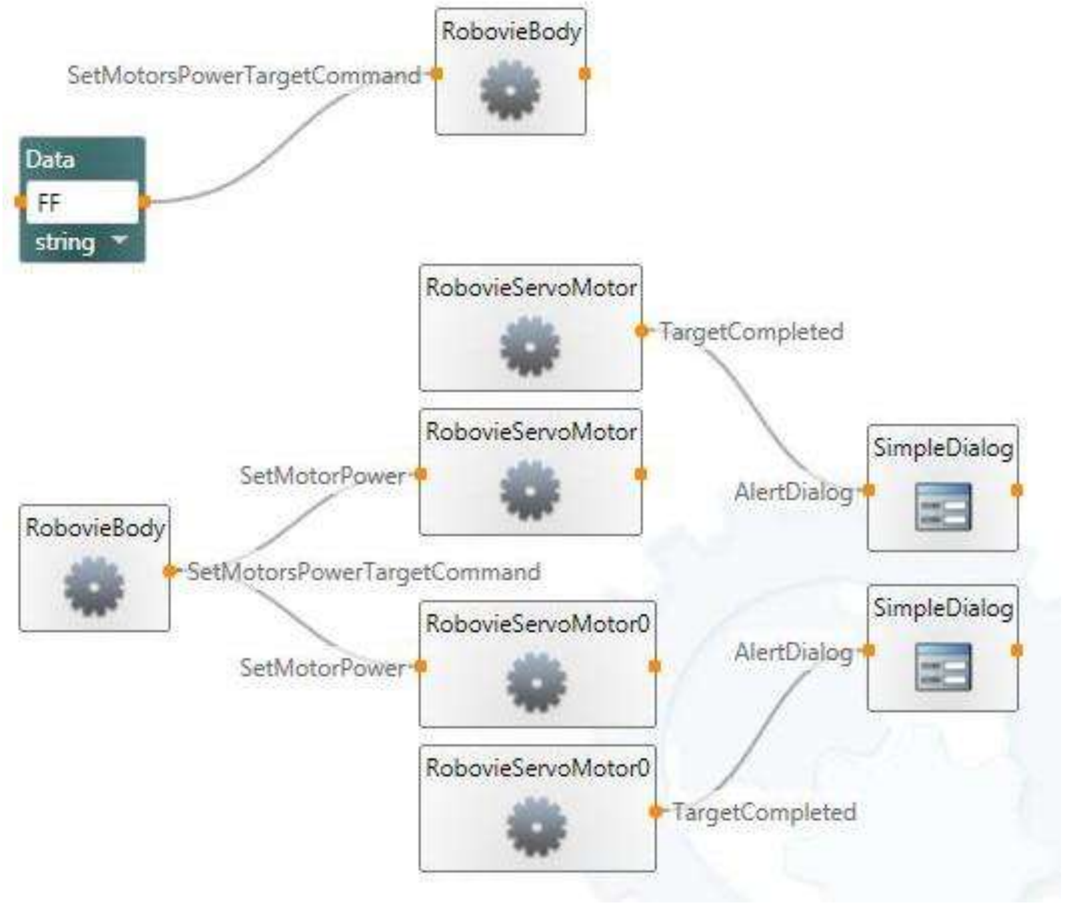


FIGURE 72

The application defined allows to management two servo motors. Data is a data value that sends its value on *Robovie Body*’s main port. When *Robovie Body* receives message, message’s value is “FF”, sends the command on mains ports of *RobovieServoMotor* and *RobovieServoMotor0*. When the servo motors are completed executing command they send a notification to *SimpleDialog* services, they show a window dialog message, containing the result of operation request.

We can compose a new application the way we want. Generally we needs to respect the service specifications created as in this case.

Chapter 7

DEMO

7.1. Introduction

In this chapter we will talk about a current plain execution of the humanoid robot model as real humanoid robot and visual entity humanoid robot.

The sending of some messages on the real humanoid robot and visual entity humanoid robot belongs to our goal.

We will describe command specification with a code part and we will show a flow diagram of our sent message type on both humanoid robot.

Then we will show Visual Programming Language application and we will small describe it minutely. In the last part we will show some screenshots that represents the command execution.

7.2. Flow Diagram

The following diagram represents the concept of flow diagram that under application:

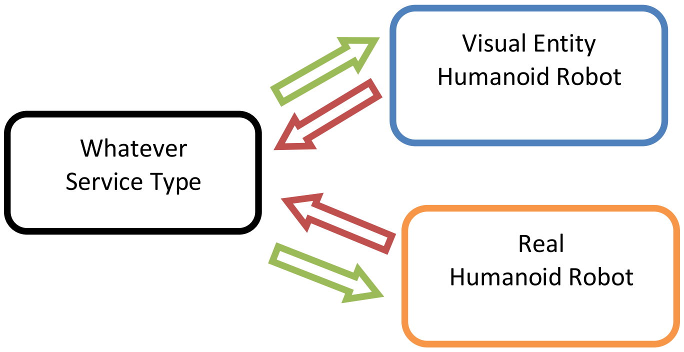


FIGURE 73

We will send a unique command on both humanoid robots, as it's represented in the following part of code:

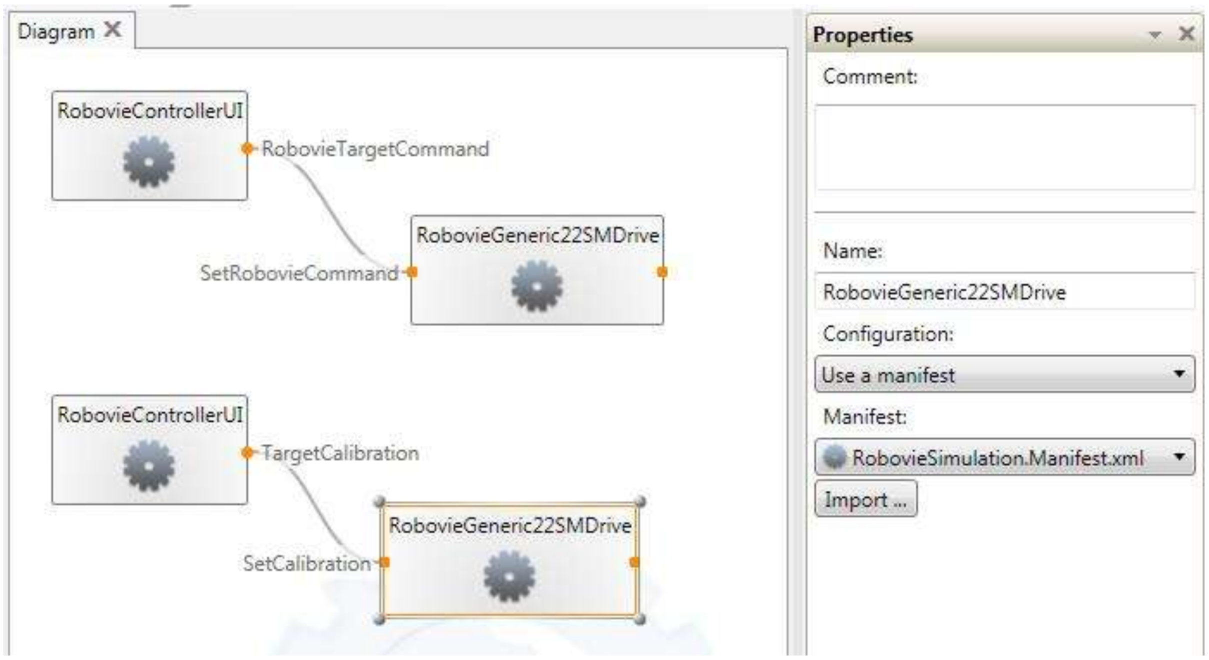
```
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ67R7FS00T7FUA0V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJFFK7FL7FM7FN7FO7FP7FQ67R7FS00TFFUD6V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ67R7FS00T7FUB4V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG78H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ78R7FS00T7FU7FV7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG78H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ78R7FS00T7FU7FV7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ67R7FS00T7FUA0V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJFFK7FL7FM7FN7FO7FP7FQ67R7FS00TFFUD6V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ67R7FS00T7FUB4V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG78H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ78R7FS00T7FU7FV7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG78H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ78R7FS00T7FU7FV7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG67H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ67R7FS00T7FUB4V7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG78H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ78R7FS00T7FU7FV7F,+1!01
@7FA7FB7FC7FD7FE7FF7FG78H7FIFFJ7FK7FL7FM7FN7FO7FP7FQ78R7FS00T7FU7FV7F,+1!01
```

CODE 116

7.3. Visual Programming Language Application

As we have described in the previous chapter, we use a Visual Programming Language application for running it.

Our visual programming language diagram is shown in the following picture:



Name: RobovieGeneric22SMDrive

Configuration: Use a manifest

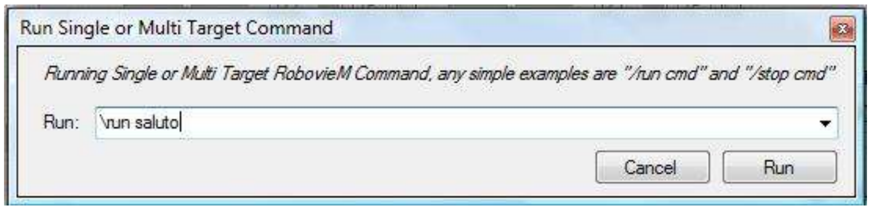
Manifest: RobovieSimulation.Manifest.xml

FIGURE 74

Where:

- *TargetCalibration* message is used to send a calibration message for setting servomotor.
- *RobovieTargetCommand* is the message type by we can send command on both humanoid robots.

When the application is running we can run the command using the following window dialog:



Running Single or Multi Target RobovieM Command, any simple examples are "/run cmd" and "/stop cmd"

Run: \run saluto

FIGURE 75

The command line is *"/run saluto"* by which both humanoid robots go running. Firstly we must send a calibration command by servo motors calibrated.

7.4. Screenshots

In this paragraph we will show a same screenshots by humanoid robot execution.

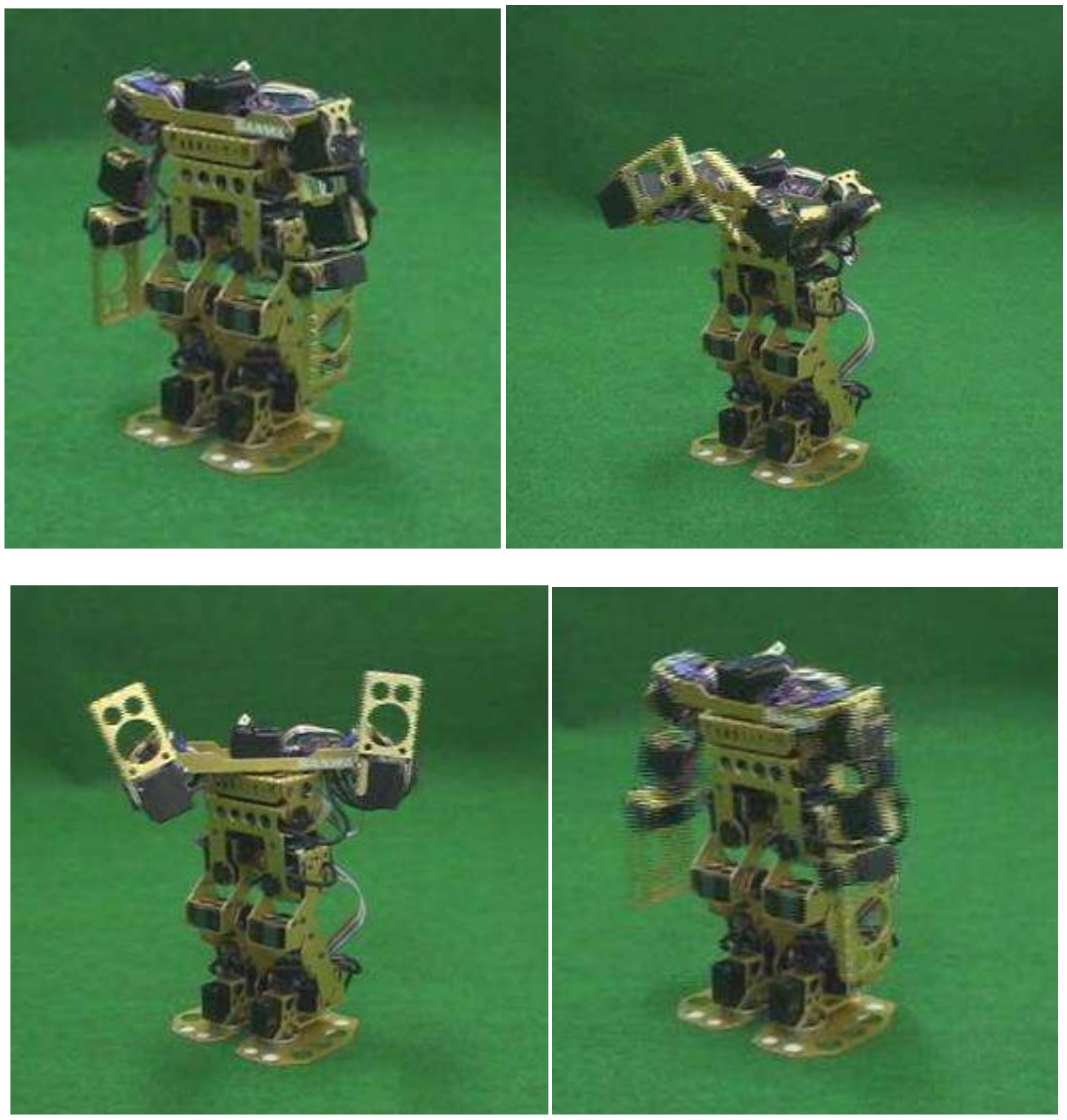


FIGURE 76: REAL ROBOVIE-M

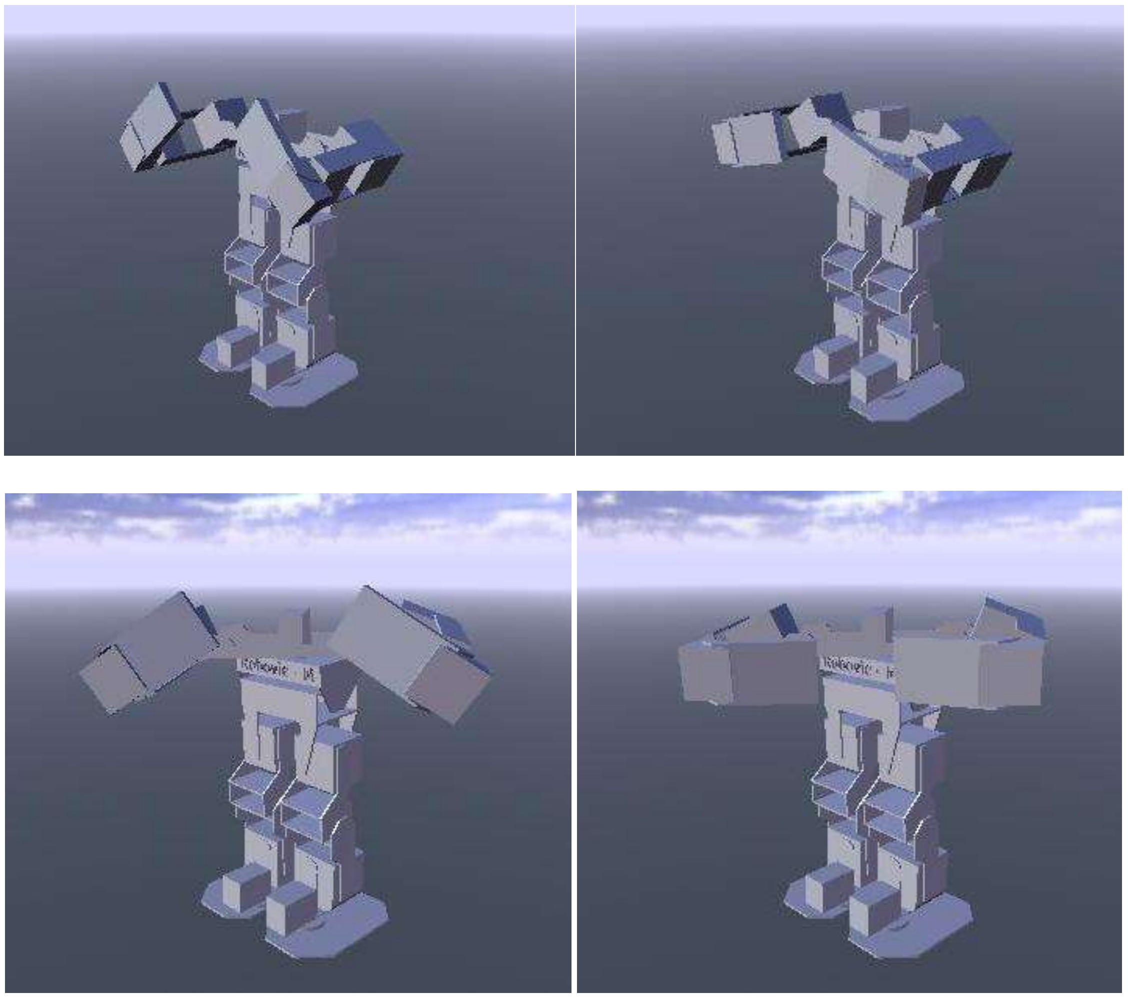


Figure 77: Robovie-M on the Microsoft Robotics Studio Simulation

Conclusions

The development of this dissertation has played a fundamental role in my global training received by the specialized degree in Information Technologies. The purposes identified at the beginning of the work have been largely realized and they will be partly taken into consideration in future jobs.

Anyway, I'm going to discuss the matter of my degree dissertation.

In the first part we have described the Microsoft Robotics studio framework specifying its main features as for example: CCR, DSS and the REST model;

We have described the way to program the framework using VPL and finally we have talked about simulation and its architectural features in a general way.

In the second part the pattern of the humanoid robot has been created as an entity to be inserted inside the Microsoft Robotics Studio simulator; we have talked about the services necessary for the robot motion and then the different types of services and their structure becoming precise and detailed especially on the services considered necessary for a right implementation; after this, we have utilized the Visual Programming Language for using the created services and for giving birth to our humanoid pattern.

Finally, a DEMO of the robot has been shown inside the simulator.

Now we can clearly affirm that the mock behavior is nearly the same as the real one, so the modeling and the insertion of the whole robot in the simulation is reasonable.

The problem that nowadays remains in the software is that actually the correspondence 1 to 1 between a command execution on the real robot and the same execution on the mock doesn’t exist.

The problem is due to a misalignment between real servomotors and the ones simulated with the maximum and minimum break-even point still in undergoing trial, marked in a notable way for not having commanded the real robot by which we could realize some attempts to verify the real operation and the real servomotors disposition in the robot internal structure.

Concerning the future jobs this inconvenient about servomotors has definitely to be solved and updating to have to be brought to make the robot more modular and sectional.

The future aim will be that of making real and proper soccer matches inside the simulator totally; a sort of RoboCup, but called in a different way, for example: Ms Virtual RoboCup.

Bibliography

[1] MSDN, Microsoft Developer Network, *http://msdn.microsoft.com/*

[2] Microsoft Robotics Studio Developer Center, *http://msdn.microsoft.com/robotics*

[3] Vstone Corporation, http://www.vstone.co.jp/e/etop.html

[4] Vstone Robovie-M, http://www.vstone.co.jp/top/products/robot/Robovie-M.html

[5] IRC, Intelligent Robotics and Communication Laboratories, http://www.irc.atr.jp/productRobovie/robovie-m-e.html

[6] Microsoft Communities, http://channel9.msdn.com/

[7] PhysX by AGEIA Company, http://www.ageia.com/

[8] Gini, Caglioti; "ROBOTICA"; ISBN:8808079635; Zanichelli; 2003

[9] Nicola Greggio, Giovanni Silvestri, Stefano Antonello, Emanuele Menegatti, Enrico Pagello; "A 3D Model of Humanoid for USARSin Simulator".

[10] O'Reilly Xml.com, http://webservices.xml.com/