uc3m | Universidad Carlos III de Madrid

Bachelor's Degree in Industrial Technologies Engineering
2016/2017

Bachelor's Thesis

Methodology for Task Development in Humanoid Robots Using ArmarX Framework

Álvaro Martínez Robledo

Tutors
Juan Miguel Gracía Haro
Santiago Martínez de la Casa Díaz

October 2017

A Carmen...

Acknowledgements

The thesis that you are reading is the conclusion of the work many people was involved in. I cannot detail in only a page all the help and support provided during all this time, but I will try.

First thanks to my tutor Juan Miguel García Haro and his trust in me completing this thesis. This work could not have been possible without your help and encouragement. If I end up development my teaching career, I know what kind of tutor I would like to be.

Special thanks to Nikolaus Vahrenkamp and Mirko Wächter for their help. Thanks for finding time to solve my questions. Also, thanks for developing such an amazing tool like ArmarX.

Thanks to Santiago Martinez de la Casa Diaz for your help in key points of this project.

Thanks to Carlos Balaguer for offering me the opportunity to continue my formation in the University Carlos III.

Thanks to all the people in TEO's laboratory for making me feel like home and not hesitating when I asked for help: Raúl F., Raúl, Elisabeth, Olaya, Aitor and Javier.

Thanks to all my friends, specially those who have made me finding joy with the most dumb things: Adri, Ferni, Kimi, Rubén, Santi, and the Chocu. Thanks to all of my friends who have shared their life with me and taught me so much things about other persons and myself.

Thanks to my family, for being there always. I could never pay back all the things you did for me.

Finally, thanks to Carmen for teaching me what happiness means.

Contents

List of Figures

List of Tables

Listings

Abstract

In this bachelor thesis, a guide on how to take the first steps in the ArmarX framework. This software provides a complete robot development environment. In particular, how to use the humanoid robot TEO of the investigation group RoboticsLab from University Carlos III in this software.

In order to complete this objective, the several stages were followed. First, a initial research about the software ArmarX was performed, learning its main features and how to use this tool. Then, the 3D model of TEO was adapted to be used in this environment. When it was complete, a task was created and adapted to the robot. Finally, it was exported to the real-world TEO and the results were compared.

During this process, an extensive documentation was performed, in order to pursue the objective previously mentioned, the formulation a guide for future researchers interested in ArmarX.

1 Introduction

1.1 Motivations

The motivations for this thesis can be found in several aspects. First, I already knew that working with robots will be something I would like to do. That is why I specialized at the end of my bachelor in electronics and automatics. Therefore, I took the opportunity of working with such an amazing robot as TEO.

Second, the idea of providing future students with information and a guide to work with. Creating a manual of a new tool, skipping them the tedious task of learning the most basic things of a program, so they can focus on more intricate projects with this software. Also, it is an opportunity to show our humanoid robot and its teamwork.

The importance of this thesis and work is immense. ArmarX has such potential, from being able to generate basic control programs, to support the creation of tasks with visual servoing; passing through learning and grasping with human-like hands. This range of abilities is the reason why several investigation centres are interested in ArmarX, to exploit to the maximum the capabilities of their own humanoid robots. From the Italian Institute of Technology (IIT), to the University of Tokyo. Their intention to collaborate is creating bridges that will allow them release all the potential of their robots, iCub and HRP-2 respectively, and their software platforms: YARP and OpenHRP.

Finally, the opportunity to challenge myself and learn from zero such amazing (and complex) tool like ArmarX. Combining things I like: robots and programming. Even thought sometimes it was frustrating, little steps were a huge reward. Watching the results of my own work is a fulfilling emotion.

1.2 Objectives

This project aims to the main goal of introducing the use of **ArmarX** in the University Carlos III. To do so, it is necessary to create a useful manual to initiate an environment that allows us to simulate a robot. So the next one who will work with it can be develop more complex tasks with the framework.

With that in mind, we will go step by step, with intermediate goals:

- Familiarize with the framework ArmarX, learn its way of work and some of the possibilities it offers.
- Prepare TEO to make its model compatible with ArmarX, learning the syntax used in the XML file that describes it. Also knowing things we can add in order to create a more accurate simulation model.
- Learning how to create out tasks in ArmarX: how to make Statecharts.

- Adding all necessary parameters and options in ArmarX to deploy a simulation.
- Compare the results of the simulation with a demonstration in the real robot. Check if there are differences and investigate their origin.

1.3 Document Organization

This project is divided in two main sections:

- The first part contains the development of the subject, in it we will find:
  - A state of the art about Robot Development Environments (RDEs), with a special mention of ArmarX and the humanoid robots that are using this software. Some of them developed their own bridges to exploit the capabilities of ArmarX.
  - Introduction to **TEO** (Task Enviroment Operator), our university's humanoid robot. Presentation of his soft- hardware architectures, capabilities and devices.
  - Presentation of the ArmarX framework, its design principles, main features and how it works. Also a special mention to its StateCharts.
  - How a robot is defined for the ArmarX environment. Possible enhances to our model.
  - Explanation of the process of running a simulation in ArmarX, using TEO as subject. Running a simulation in order to obtain the results.
  - Deploying the operation in the real TEO, compare results to highlight the differences between the simulation and the real model.
- The second part includes several annexes describing step by step the process followed in the thesis. This section comprehends a notable extension of this work. That is due to the main idea of this work to become a guide to teach new developers in the use of the software ArmarX. Also it is divided in several sub-annexes:
  - Annex A explains the steps followed to build a XML file that contains our robot compatible with ArmarX environment, in addition with some features we can add to it.
  - Annex B is intended as a introducing guide to ArmarX and will implement a counter.
  - Annex C will expand the work done in Annex B adding the robot model and the task chosen for TEO.

2 State of the Art

This section will provide a general information about actual robot-development software. Their idea and main purposes of them. Also a brief introduction to some of the most used. Then, ArmarX is presented. A brief presentation of some robots that use this framework. With that, the intention is to show the importance of this tool, and the advantages it will bring to our university.

2.1 Robot Development Environment

In contrast with other software components, robots work with real-time actuators, sensors and effectors with physical parameters that must be monitored [1]. In order to ease the research in humanoid robots, **Robot Development Environments (RDEs)** were created. This programs join together all different elements of the robot. In addition, they should make the work and training needed for the developer as easy as possible.

Currently, several RDEs are available because they have been developed in the past 10 years. A few examples of them are:

- **OpenRTM-Aist:** Stands for Robot Technology (RT) Middleware [2]. Developed by the National Institute of Advanced Industrial Science and Technology of Japan. Is a distributed control architecture intended not only for industrial field but also to nonindustrial field such as human daily life support systems. Several Japanese robot platforms build their robot frameworks over OpenRTM, like the one for the HRP family, OpenHRP [3].

Figure 2.1: OpenRTM-aist software.

- **ROS:** The Robot Operating System (ROS) was designed to develop large-scale service robots. This framework relies on a communication based on *peer-to-peer* [4]. It is also language neutral and can support different languages: C++, Python Octave and LISP are supported. ROS created a base to communicate using a *peer-to-peer* method, and in addition, includes modules to work with it, some of them for simulation.

Figure 2.2: Lunar Loggerhead is the 11th official ROS release.

- **OROCOS:** Open RObot COntrol Software (OROCOS), is a project that follows a similar Open Source development than Linux or LaTeX. Consists in researchers and engineers that will contribute code, documentation and feedback to the project. Relies on COBRA standards and its real time extensions [5].

Figure 2.3: The Orocos project.

- **YARP:** Stands for “Yet Another Robot Platform”. This middleware is not intended to be a complete operating system of our robots, but to be used alongside other programs and communicate them. The goal of YARP is to minimize the effort devoted to infrastructure-level software development by facilitating code reuse, modularity and so maximize research-level development and collaboration [6].

  This software is already used in the Robotics department. TEO’s current control software is constructed over it. All access to TEO’s hardware is run through YARP applications.

Figure 2.4: Yet Another Robot Platform.

- **OpenRAVE:** Another Open Source software that provides a visualization environment for simulation. OpenRAVE has been already used with several TEO applications like its main simulator and a collision checker plug-in.

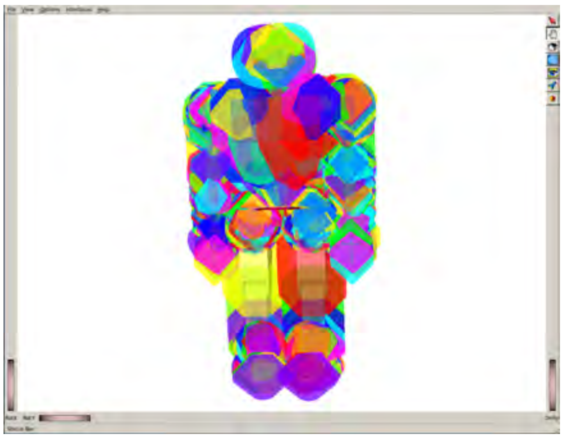


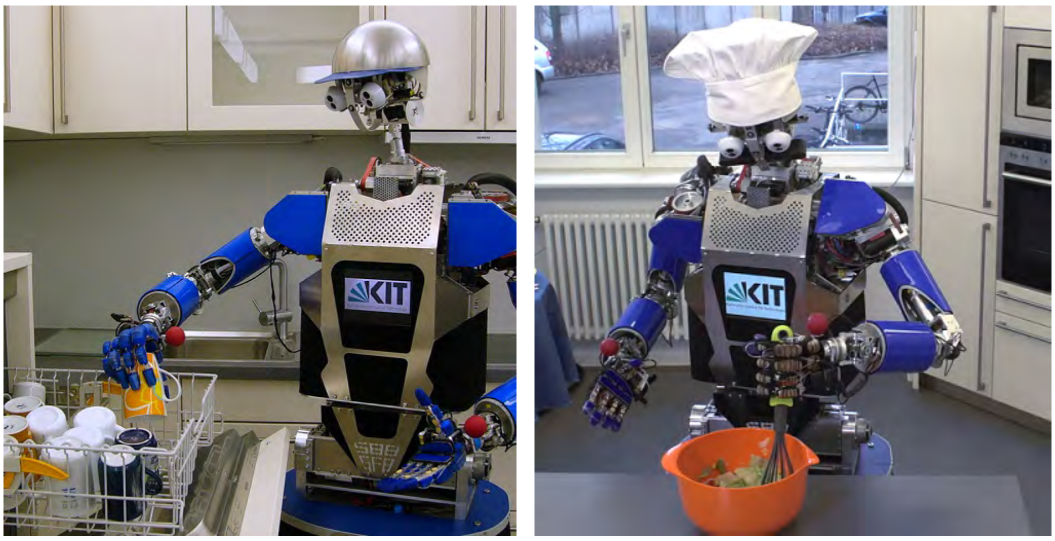
Figure 2.5: OpenRAVE collision module implemented in TEO.

2.2 ArmarX Relevance

ArmarX is a robot programming environment that has been developed by the Karlsruhe Institute of Technology (KIT). In order to ease the realization of higher level capabilities needed by complex robotic systems such as humanoid robots. It is built over the idea of a constant disclosure of the system state. This feature will allow programmers to easily access to all data [7].

The ArmarX framework provides several tools, like a Graphical User Interface (GUI) or statechart editors. With a high modularity and reusability, ArmarX can be used to easily program complex tasks and revise all its information.

This simulation software is mainly used by its developers in with the Armar series, actually **Armar III** [8] and **Armar IV** [9]. But ArmarX framework is not only used by its developers: ArmarX is used in several international projects of cooperation.



(a) ArmarIII doing the dishes. (b) ArmarIII preparing a salad.

Figure 2.6: ArmarIII household environments assistant.

One example is the **iCub**, the humanoid robot built by the Italian Institute of Technology (IIT) and founded by the European Union. The common project had as objective trespassing the grasping skills from Armar III to iCub using the ArmarX framework [10]. Both teams IIT and KIT succeed in creating a bridge between its middlewares: whereas ArmarIII used ArmarX, the iCub works with YARP [11].

Figure 2.7: iCub from the Istituto Italiano di Tecnologia (Italian Institute of Technology).

Figure 2.8: HRP-2 humanoid robot.

ArmarX is also used by the Japanese HRP-2 humanoid robot. The aim is a bridge between OpenHRP and ArmarX. This humanoid robot has 30 DOF (Degrees Of Freedom) and is able to stand up from lying down in the floor. This ability was already achieved in smaller robots, but not in one this size, thus HRP-2 is 1.54 meters high and weights 58 kilograms.

3 TEO

TEO (Task Environment Operator) is a humanoid robot developed by the robotics group *RoboticsLab*, at the University Carlos III of Madrid. It was meant as an improved version of its predecessors RH-0 and RH-1 [12].

3.1 Structure Overview

This anthropomorphic robot was designed to act and behave like a human. It is 1.70 meters high and weights 70 kilograms. With 28 DOF (Degrees Of Freedom) can perform elaborated manipulation tasks.

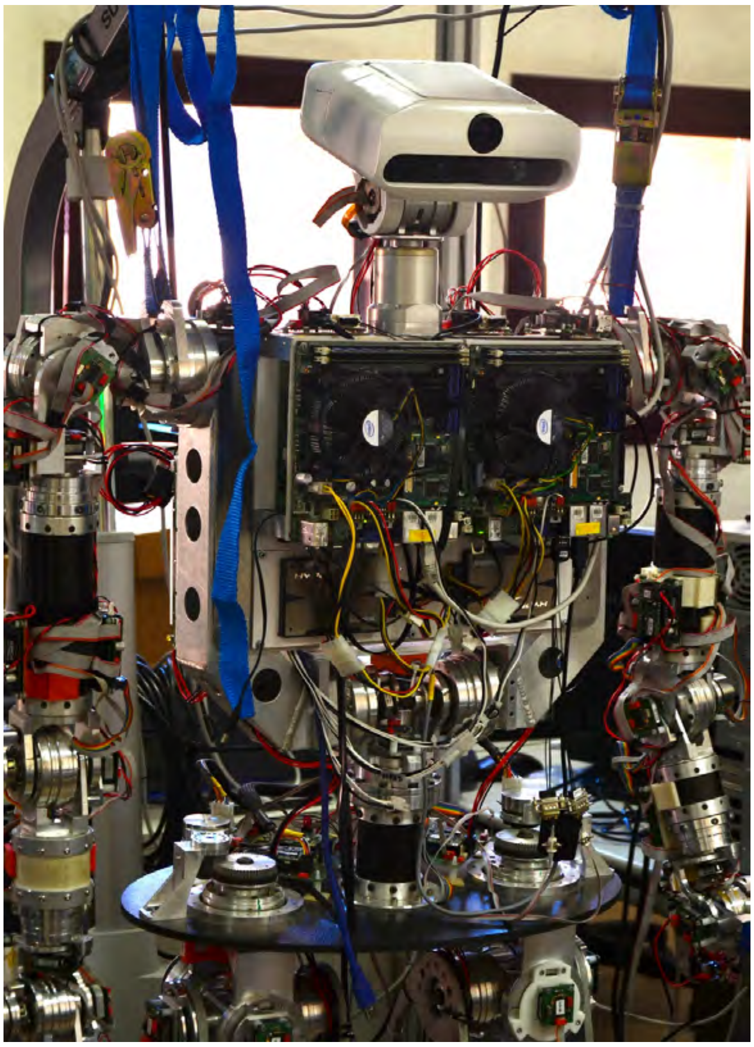


Figure 3.1: Task Environment Operator, TEO.

Its body consists in a structure of aluminium and over it the electronic is mounted. TEO has several force/torque sensors that allows him to stand and control its posture [13]. The robot has 4 microprocessors: *locomotion*, *manipulation*, *artificial vision* and finally, the main processor which manages the others. The locomotion processor, that controls the legs and the torso.

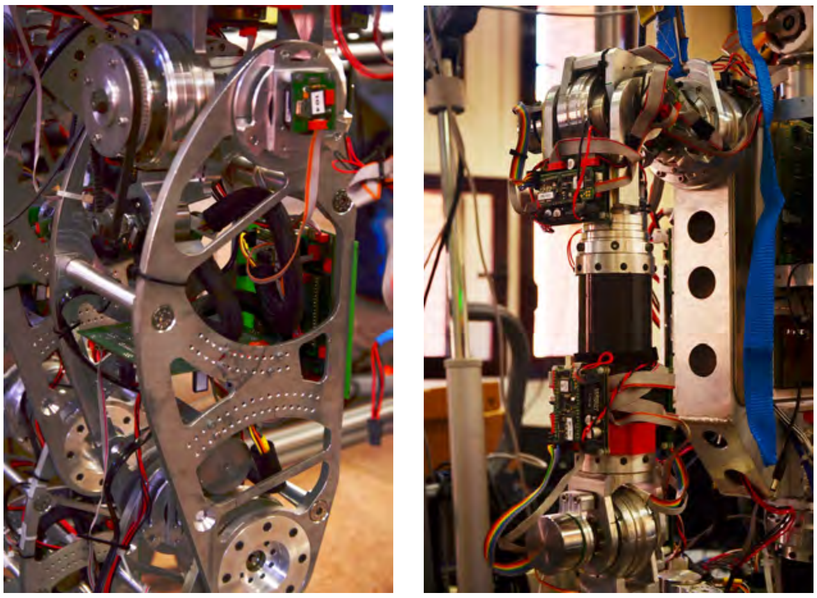

(a) TEO's right leg. (b) TEO's right arm.

Figure 3.2: Closer insights of TEO's structure.

The manipulation processor controls the movement of the arms and the head. The processor responsible fort he computer vision uses a camera with a infra-red sensor, RGB and depth capabilities provided by ASUS. The camera is located in the head, as seen in Figure 3.3.

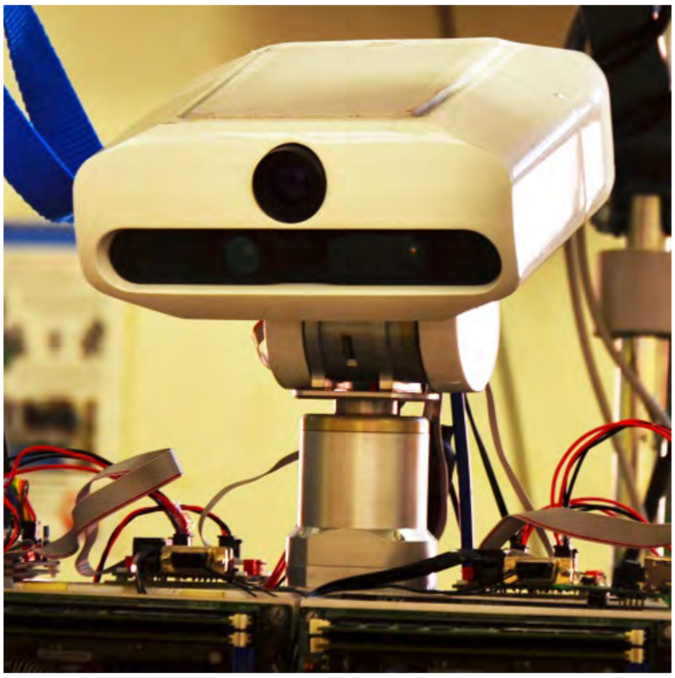
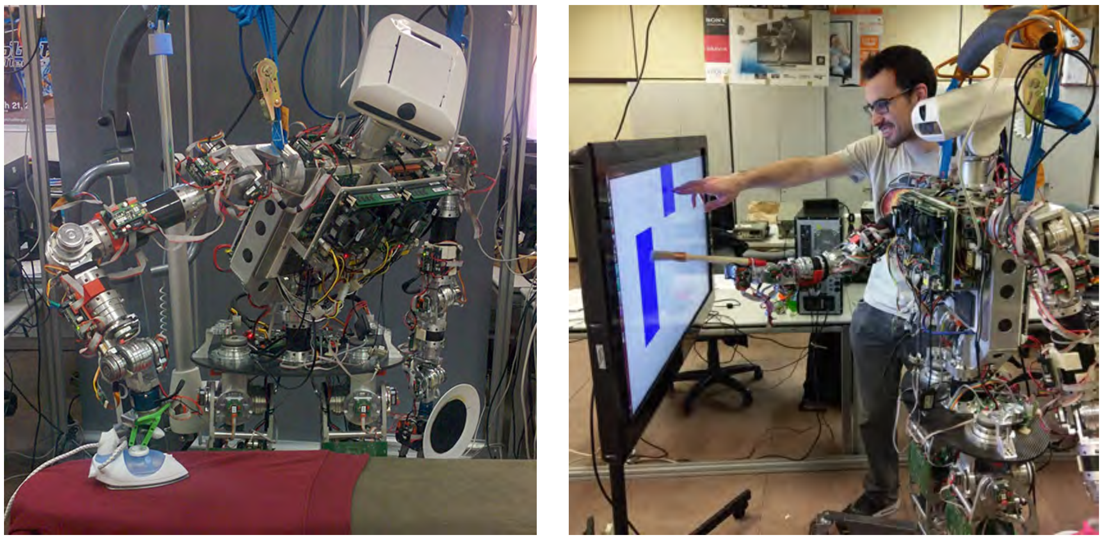


Figure 3.3: TEO head with the camera

3.2 Functional Capabilities

In addition to its mechanical abilities, TEO can also learn and interact with its environment [14]. This high level capabilities have been used to program knowledge-acquiring skills. Recently, developers in the RoboticsLab have been able to create applications that allowed TEO to reproduce human exercises like ironing or painting (Figure 3.4).


(a) TEO ironing the wrinkles of a shirt [15].

(b) TEO learning how to paint from a human.

Figure 3.4: TEO completing human tasks.

TEO can successfully perform duties in human environments, after all, it is the final purpose of all humanoid robots. Researchers in the university have developed a high end application for TEO to act as a waiter, delivering bottles and caps in a tray while advancing. It uses all of its capacities to this task:

- The use of his vision facility allows him to detect unstable bottles in the plate located in its arm. In addition, using this information, it can relocate the arm to make it stable again [16].
- In addition, it can obtain direct feedback of the object's stability with its inertial, position and force/torque sensors. This information is used to recalculate the new position so the object in balance dues not fall [17].

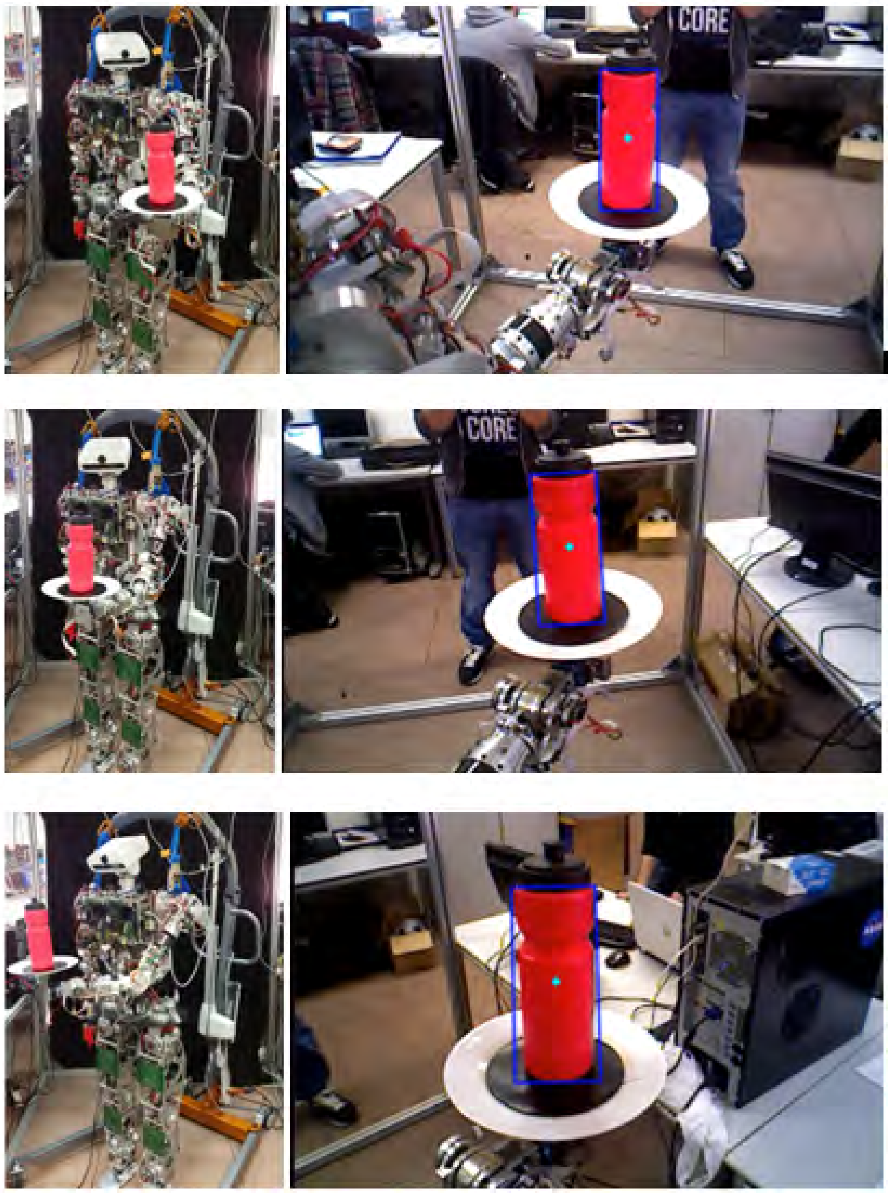


Figure 3.5: Head movement to maintain the bottle centered in the image.

4 ArmarX

The Robot Development Environment **ArmarX** is a software framework created by the High Performance Humanoid Technologies Lab (H^2T) of the Karlsruhe Institute of Technology (KIT).

4.1 Introduction

The ArmarX platform allows programmers to design robot tasks in a easy, visual and intuitive way; via **statecharts**. Here we will describe how our robot will behave, each state and its parameters. Later ArmarX will execute this simulation, and with its extensive log information, we will be able to obtain the results and visualize the program loaded in our robot [7].

ArmarX is composed by numerous components: 3D visualizers, visual perception recorders, collision checkers and several observers in order to collect the information. This components communicate themselves using ethernet ports.

Recapitulating, this is a very powerful software with a immense range of possibilities that will allow us to simulate each and every aspect of a robot. More importantly, we can push the knowledge of our robot, programming new tasks and inspect every result obtained. With the constant feedback od ArmarX and its intuitive way-of-work, we can develop complicated tasks for our robots far more easily than just writing thousands of code lines.

4.2 Design Principles

This section will provide a general information about ArmarX and the objectives its developers wanted to accomplish with this tool:

- **Distributed processing:** Typical architecture of actual robots consists in several PCs, each controlling one or various subsystems. They can be in charge of controlling hardware elements such as motors, sensors, cameras, etc. All this computers must be able to communicate properly and in a clear way. With this purpose, ArmarX allows communication using a shared memory or Ethernet, so the robot programs reach all required hardware.

- **Interoperability:** Nowadays, both Hardware and Software used in robot applications are far from being standardized. Therefore is compulsory that every RDE allow us to work with an extensive range of different hardware platforms and operating systems. To fulfil this requirement, ArmarX can be used in Windows, Mac OS X and Linux. In addition, ArmarX uses an interface definition language (IDL) supporting a variety of platforms and programming languages (C++, Java, C#, Objective-C, Python, Ruby, PHP, and ActionScript). Also the hardware coverage can be easily extended.

- **Open source:** ArmarX is available open source under the GPL license. Providing RDEs under an open source license is essential in order to achieve the most impact on robotics by allowing researchers and developers to achieve a deep insight in the underlying mechanisms.

- **Graphical editing of Control- and Dataflow:** ArmarX makes a special emphasis in allowing us to manage the control of our robot using a graphical interface, like with the StateChart editor; but also giving us all the data that is being manipulated in a clear way.

- **Disclosure of the system state:** Means that the current state and all its parameters can be inspected at runtime and logged for future behavior adaptation via a network interface. It allows programmers to access the data of many parts of the system required for debugging, monitoring and profiling purposes. Since the amount of available data increases with the size of the developed system an abstraction of the data flow into easy to grasp visualizations is required.

4.3 Software Architecture

From a sofware point of view, ArmarX is formed by three layers: the *Middleware Layer*, the *Robot Framework Layer*, and the *Application Layer*.

The Middleware Layer abstracts communication mechanisms, provides basic building blocks for distributed applications, and defines entry points for visualization and debugging.

The Robot Framework Layer includes several different and specific projects but under the final goal of adding capabilities closer to our real robot. This includes access to the sensor and motors on a higher level. The mentioned projects are the ones that include memory (**MemoryX**), robot and world model, perception (**VisionX**), and execution modules. Robot custom APIs can be e implemented by extending and further specializing the generic robot API building blocks.

The Application Layer is where the final robot program is implemented. We can use generic and specific APIs in our robot.

Also ArmarX provides us tools to facilitate the development of robot applications. Here is integrated the graphic user interface (**ArmarX GUI**), the **StateChart Editor**, inspection and simulating environments.

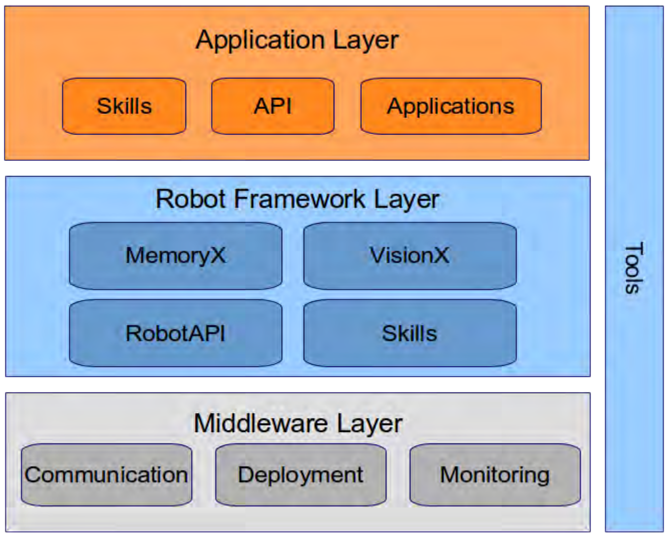


Figure 4.1: ArmarX software structure. Components in grey are provided by ArmarX, in blue, configurable/extensible and in orange, custom implementation.

4.3.1 Middleware Layer

The Middleware Layer is built over the **ZeroC Internet Communication Engine** (**Ice**) [18], a computing platform that provides basic components for developing robot architectures. Ice is a platform that implements a distributed computing processes using **Ethernet**. In ArmarX, this network based-communication mechanism is expanded using a shared memory block in order to transfer large data chunks efficiently. **ArmarXCore** belongs to this part of the framework and is vital for the foundations and integration between the lower level elements and custom APIs with high level programming.

Additionally, the Middleware Layer provides a number of essential tools such as transparent shared-memory, Ethernet transfer of data, and thread pool based threading facilities.

The lowest level is the *Sensor-Actor Unit* serving as abstraction of robot components. *Observers* monitor sensory data streams and generate application specific events which trigger *Operations* to issue control commands to the robot.

- **Sensor-Actor Units**: Provide abstractions of robot components such as kinematic structures or cameras. In ArmarX the sensor data is accessible by a publisher-subscriber mechanism. In the other hand, the control is realized using remote invocation.

- **Observers**: API events originate from desired patterns in sensory data such as high motor temperatures or reaching a target pose. Observers generate these events by evaluating application defined conditions on the sensory data stream. The API offers a set of basic checks and provides interfaces to support implementing more advanced and application specific checks easily.

- **StateCharts**: One fundamental elements of ArmarX , already mentioned in Section 4.2, is the representation of robot operations using as state-transition networks, called statecharts. Each state in the network is defined by a set of input/output parameters and can issue control commands or start asynchronous calculations. State transitions define the data flow between states by mapping output values of one state into the input values of the next state.

4.3.2 Robot Framework Layer

The Robot Framework Layer makes possible the integration of robot definitions with sensors and memory use. Here the APIs can be customized in order to be used in a specific robot.

A robot definition for ArmarX environment will consist in a kinematic structure, with optional dynamic parameters, such as masses or inertia matrices; and sensors like position or cameras. This robot definition will be written in a XML file. Further detail is explained in Section 5. This part of ArmarX will be in charge of this dynamic and cinematic aspects of our robot.

The Robot Framework Layer contains important modules of the ArmarX architecture [19]:

- **MemoryX**: Responsible of all memory related components in ArmarX. Holds basic building blocks for memory structures. This can be integrated in the system's database to form permanent information [20]. It contains several applications: **Working Memory, Long-Term Memory** and **Prior Knowledge.**
  - **Prior Knowledge**: Holds information which is already known to the robot and which has been predefined by the user. We can add information about 3D objects or features for object detection.
  - **Working Memory**: Contains robot's actual state. It can be updatedby the perception of our robot or by *Prior Knowledge* using an updater interface.
  - **Long-Term Memory**: Provides permanent storage capabilities. Offers the potential of learning . It also allows creating snapshots of the current *Working Memory* state to be used for later re-loading.

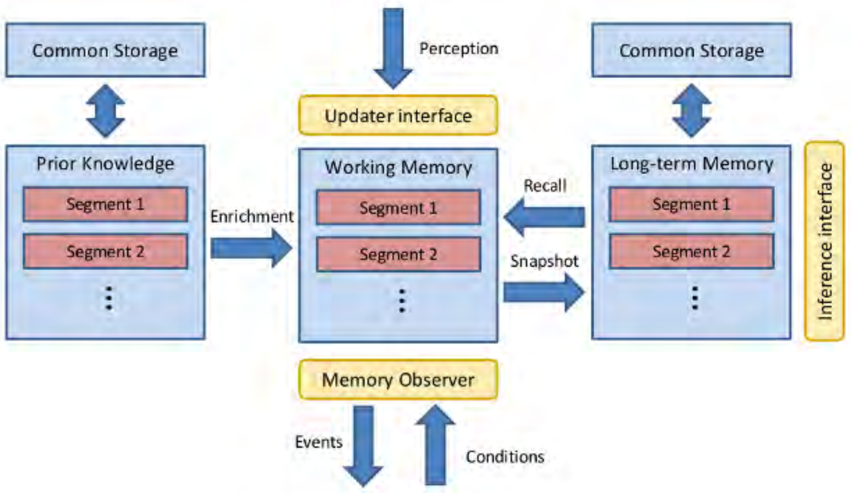


Figure 4.2: MemoryX structure.

- **VisionX**: Includes perception components of ArmarX based on image processing from cameras. It can provide images and distribute the data through Ethernet or shared memory. It can also integrate object recognition. The results are written in the *Working Memory* of *MemoryX*.

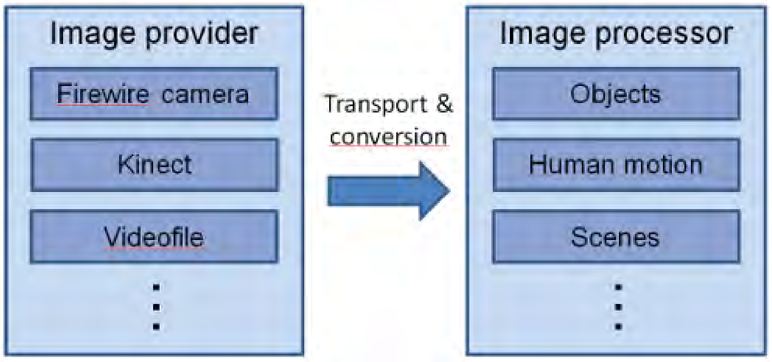


Figure 4.3: VisionX structure.

- **ArmarX Simulator**: All RDEs must include a simulator in order to revise the robot programs. Even though they cannot provide decisive information about components such as sensors or physical interaction, we can observe the general behaviour of our robot program. The framework provides the ArmarX Simulator component, which can carry the computing of sensors, motors and dynamics of our robot.

- **Robot Program**: All applications for ArmarX will be organized in Statecharts. All program logic is represented as one comprehensive statechart consisting of many operations which might be executed on different hosts due to the distributed nature of ArmarX. **Añadir imagenes de estos tres apartados**

4.4 StateChart Concept

4.4.1 Introduction to the Statecharts

The statechart language is an evolution of conventional formalism of state machines and state diagrams, created by D. Harel [21]. This type of diagram, widely used in computer science, offers information about the behaviour of a system.

The use of statecharts in ArmarX is deeply integrated in the framework itself, thus, every application will consist in a statechart, with hardware abstraction of a robot. Most of components in the software are aimed to complement themselves with its use.

4.4.2 Design Overview

The key principles of the ArmarX statecharts are modularity, reusability, runtime-reconfigurability, decentralization, and state disclosure.

- **Modularity:** Comes naturally in ArmarX statecharts. Each individual state can have its own input and output parameters.
- **Reusability:** Each state can be introduced in other sub-state and define its specific interaction. Smaller processes defined in a statechart can be joined together in order to create bigger and more complex robot behaviour.
- **Runtime-reconfigurability:** Means that a statechart can be defined in configuration files, and that the statechart structure can be changed completely at runtime.
- **Decentralization:** For increased robustness, the statecharts don not consist in one process, they can be spread over several ones. Errors in one state component will not crash the whole process, but to inform other stances that has failed.
- **State disclosure:** Already mentioned in Section 4.2, it is very important to access and log all actual states and parameters. That ArmarX Framework and the statecharts share this common goal is one of several hints about how deeply the use of statecharts is related with the software itself.

4.4.3 ArmarX Statecharts

The statecharts of ArmarX are based on Harel's, but with some differences. They are simplified versions of them, some features are omitted. Also some additions are made, do not forget that Harel created the system in 1987, data flow specification and control during transitions are added in ArmarX. Also each state can include asynchronous code, so different states can run simultaneously.

Every statechart in ArmarX is itself a state of the robot. Also, we can add ongoing transitions to a statechart. This transitions link the states between them. The states can be programmed to consider several possible events. Figure 4.4 shows a minimal example. It is possible to include states inside others, creating substates. This is the main structure of statecharts in ArmarX: a main state that holds several substates related with outgoing events.

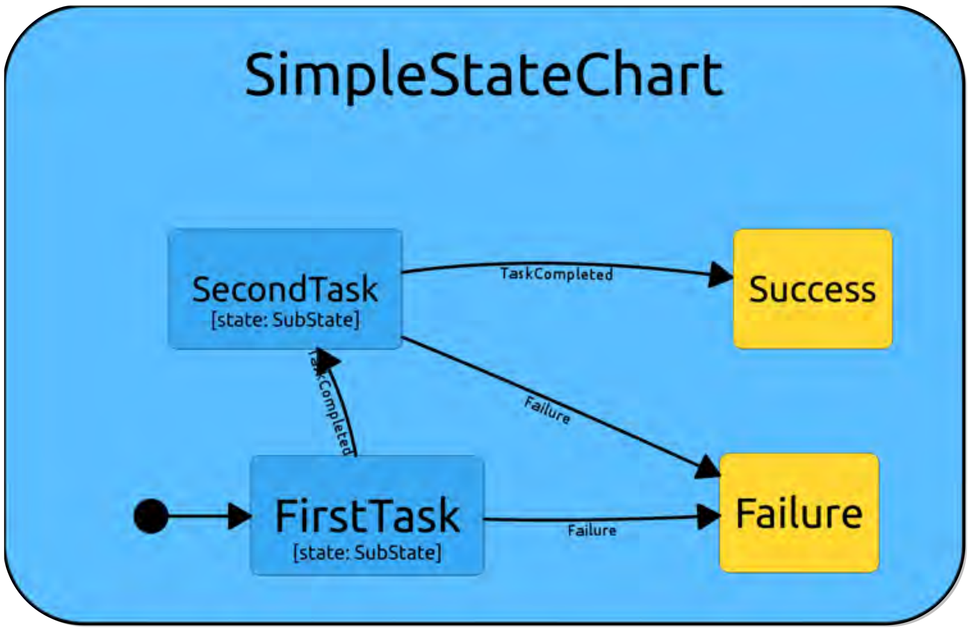
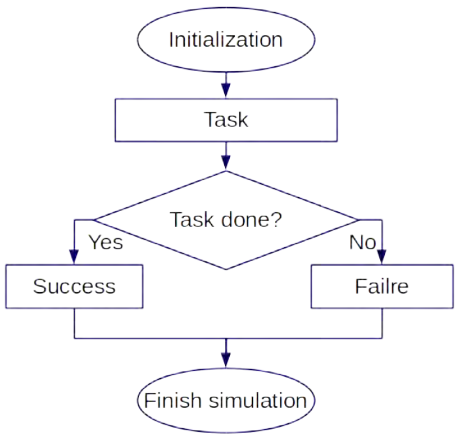
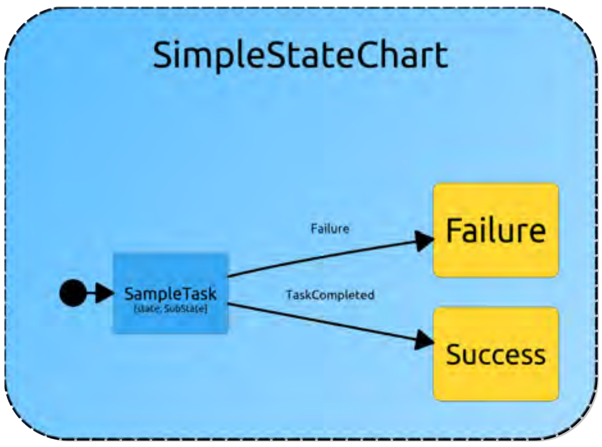


Figure 4.4: A basic statechart in ArmarX.

They contain the functions: *onEnter(), onBreak() and onExit()*, (synchronous) and *run()* (asynchronous) to provide functionality. The synchronous functions will be executed each time the state is entered, stopped or normally exited, respectively. The *run()* function will hold the continuous use of the desired state [22]. This functions can be programmed (in C++) create parameters, use actuators or make decisions. Hence, this capability can compared to the flowcharts diagrams: being each state together both action and decision. Figure 4.5 shows two diagrams, one a flowchart and one statechart, identical.



(a) Flowchart diagram of a simple behaviour.



(b) Statechart diagram of a simple behaviour.

Figure 4.5: Equivalent diagrams in different representations.

All states can hold parameters, in fact, is where the main data is hold. A state can specify three types of parameters:

- **Input parameters:** Already defined by the programmer or previous states before the state is entered. They are read-only and cannot be changed by this state. Generally, they specify general parametrization for the program like fixed parameters or constants.
- **Local parameters:** Computed within the state. Can be used for temporary memory storage or to be passed to following states.
- **Output parameters:** Created in the developer code. They can be passed to the next state.

5 TEO Models

This section explains the process followed to create a compatible model of TEO for ArmarX, describing the tool we will use: **Simox**, and the possibilities and requisites to build a robot for this environment.

5.1 Prerequisites

Before installing and compiling ArmarX in our computer we first need the 3D models of the robot we want to work with. Each program usually uses a different and specific format to load the full robot, not the 3D format: .iv, .wrl, etc. But the way all its parts are joined together: what hierarchy follows, joints parameters, actors units and so on. Usually, **XML** (eXtensible Markup Language) is used. It contains the information previously mentioned in a text-based class structure. ArmarX is no exception and it bases the loading process in the tool **Simox**.

Simox is a software platform also developed by H^2T that allows to set up 3D robot environments. The robotics department already had XML files compatibles with OpenRAVE, but unfortunately, they are not suitable for Simox-based tools.

5.2 Simox

Simox is another proyect from H^2T, its a package C++ tools that provides algorithms for 3D simulation. Is composed of three libraries: *VirtualRobot, Saba and Grasp Studio*; their objective is create scenarios and calculate functionalities for robots [23].

- **VirtualRobot:** This library is used for defining environments with one or several robots with many degrees of freedom. This will be the part of Simox we will use for our project.
- **Saba:** Works with data incorporated from VirtualRobot to calculate free-collision trajectories.
- **Grasp Studio:** Library completely dedicated to plan optimal *grasping*, from simple clamps to complex humanoid hands with tens of joints.

5.3 Robot Components in ArmarX

Once we have a look at the options ArmarX give us we soon realize the huge range of possibilities we have in front of ourselves. We can include in our robot not just the basic 3D models and their position, we can add directly: sensors, cameras, electrical information. Not only that, is possible to fulfil, a detailed and extensive information about all aspects and elements in the robot.

In the next section, complete information about defining a robot is given. Each part contains a description, diagram o the class and, if necessary, a table for specific fields of the class.

Figure 5.1 describes how a robot is structured: a origin point (*RootNode*) and a series of child nodes which contains the robot's parts. Also you can add custom lists of nodes called *RobotNodeSet*.

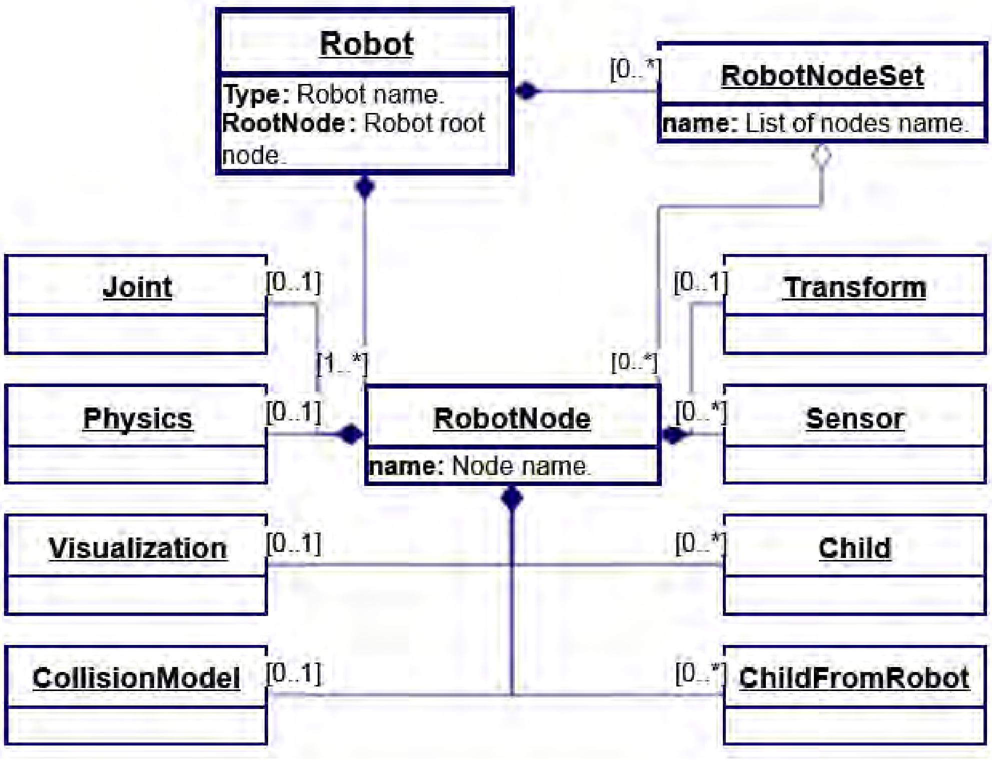


Figure 5.1: Class diagram of a robot in Simox.

5.3.1 RobotNode

Simox defines a robot like a succession of objects, so called *RobotNodes*. They are points in the space that also contain information. Each node corresponds to a piece of the robot: the waist, the forearm, etc; and stores all information about it. The way a node is usually defined follow this process:

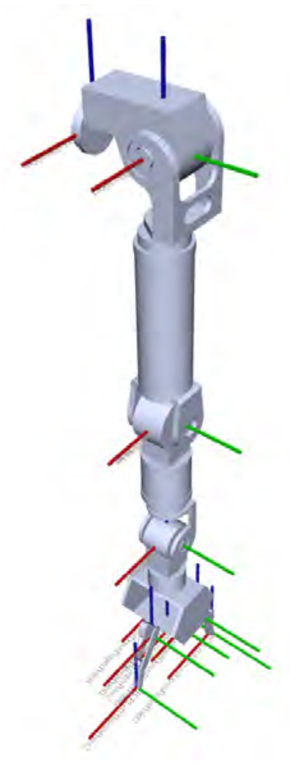
- First name it, this exact same name will be used if it is a child of another node.
- Apply a transformation if the coordinate system does not match with its parent's.
- Add the visualization and collision 3D models. Also calculate and introduce the physics parameters.
- Designate a child (or more than one) to keep with this same procedure. Also yo can import a child from another previously defined XML file.

5.3.2 Joint

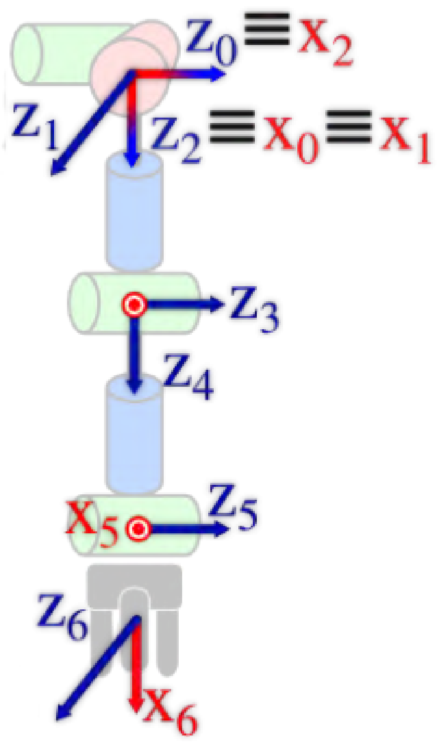
Each node can contain a joint, but is optional, for example, when defining a part that does not begin where the previous ends, so we need a transformation to keep cohesion in the robot's structure.

It is possible to define two kinds of articulations: *Prismatic* and *Revolute*; they can also be set to *Fixed*, but not defining a joint already implies that will not allow relative movement.

Once we set the type of our new joint, we mark in which axis should move or rotate. Here we have a difference with other TEO's models: Simox does not change the axis of subordinate coordinate systems (See Figure 5.2), so the rotation axis and the direction are changed to match previous diagrams.



(a) TEO's left arm coordinate sub-systems in Simox (RobotViewer).



(b) TEO's left arm coordinate sub-systems diagram in OpenRAVE.

Figure 5.2: Coordinate system differences between Simox and OpenRAVE.

As shown in Figure 5.3 we can add more information about the joints like its limits or maximum velocity/acceleration/torque.

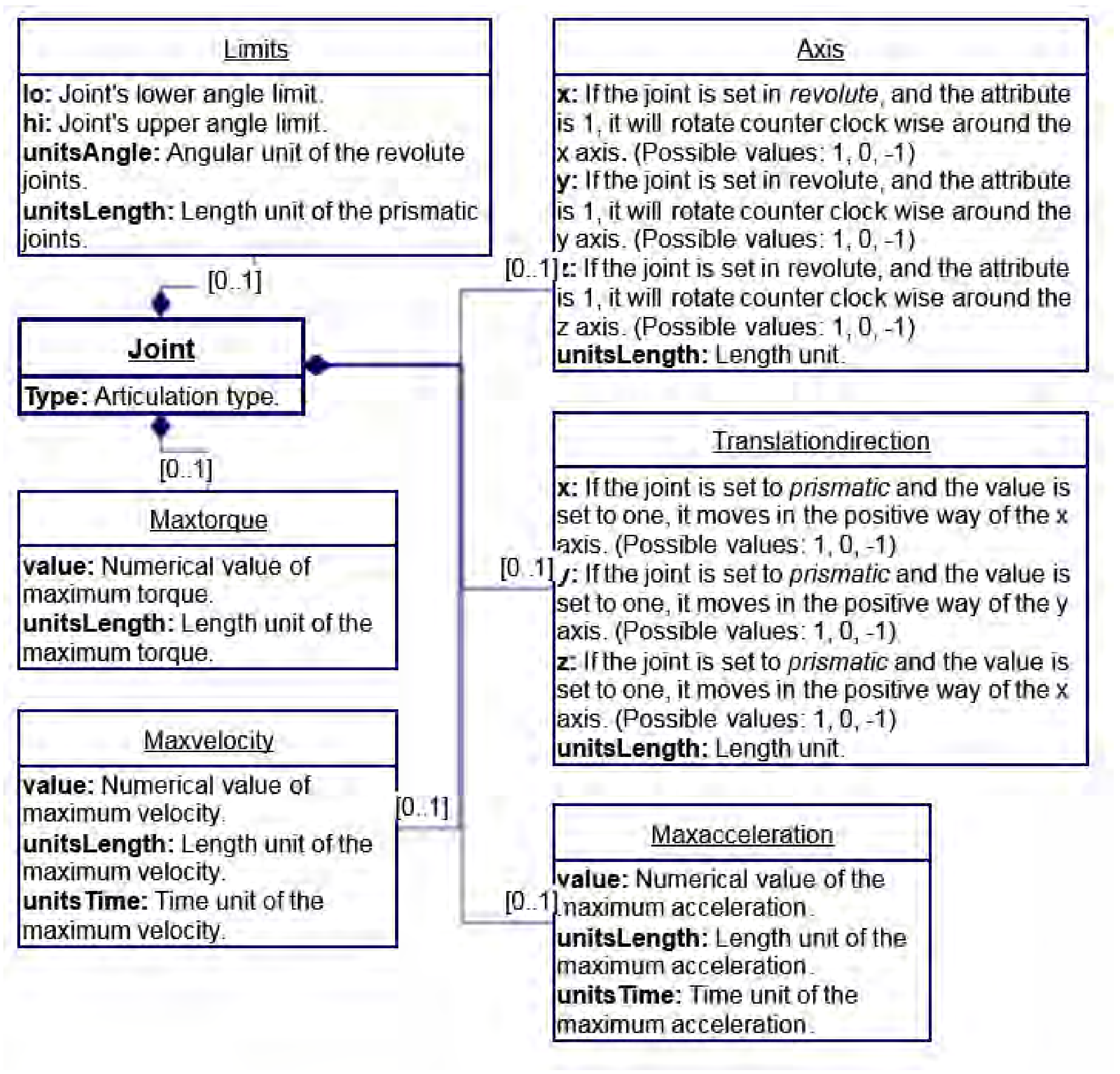

Limits
lo: Joint's lower angle limit.
hi: Joint's upper angle limit.
unitsAngle: Angular unit of the revolute joints.
unitsLength: Length unit of the prismatic joints.
[0..1]
Joint
Type: Articulation type.
[0..1]
Maxtorque
value: Numerical value of maximum torque.
unitsLength: Length unit of the maximum torque.
Maxvelocity
value: Numerical value of maximum velocity.
unitsLength: Length unit of the maximum velocity.
unitsTime: Time unit of the maximum velocity.
[0..1]
Axis
x: If the joint is set in revolute, and the attribute is 1, it will rotate counter clock wise around the x axis. (Possible values: 1, 0, -1)
y: If the joint is set in revolute, and the attribute is 1, it will rotate counter clock wise around the y axis. (Possible values: 1, 0, -1)
[0..1]
z: If the joint is set in revolute, and the attribute is 1, it will rotate counter clock wise around the z axis. (Possible values: 1, 0, -1)
unitsLength: Length unit.
Translationdirection
x: If the joint is set to prismatic and the value is set to one, it moves in the positive way of the x axis. (Possible values: 1, 0, -1)
[0..1]
y: If the joint is set to prismatic and the value is set to one, it moves in the positive way of the y axis. (Possible values: 1, 0, -1)
z: If the joint is set to prismatic and the value is set to one, it moves in the positive way of the x axis. (Possible values: 1, 0, -1)
unitsLength: Length unit.
Maxacceleration
value: Numerical value of the maximum acceleration.
[0..1]
unitsLength: Length unit of the maximum acceleration.
unitsTime: Time unit of the maximum acceleration.

Figure 5.3: *Joint* class diagram in Simox.

Parameter	Available options
unitsAngle	"rad", "radian", "deg" or "degree"
unitLength	"m", "meter", "mm" or "millimetre"
unitTime	"sec", "second", "min", "minute", "h" or "hour"

Table 5.1: Additional information for the class *Joint*.

5.3.3 Physics

It is possible to store parameters to have, in addition to the kinematic simulation, a dynamic one. We can add the mass, the inertia matrix and the center of mass so we have an accurate dynamic description of our robot. More detail is given in the Figure 5.4 and in the Table 5.2.

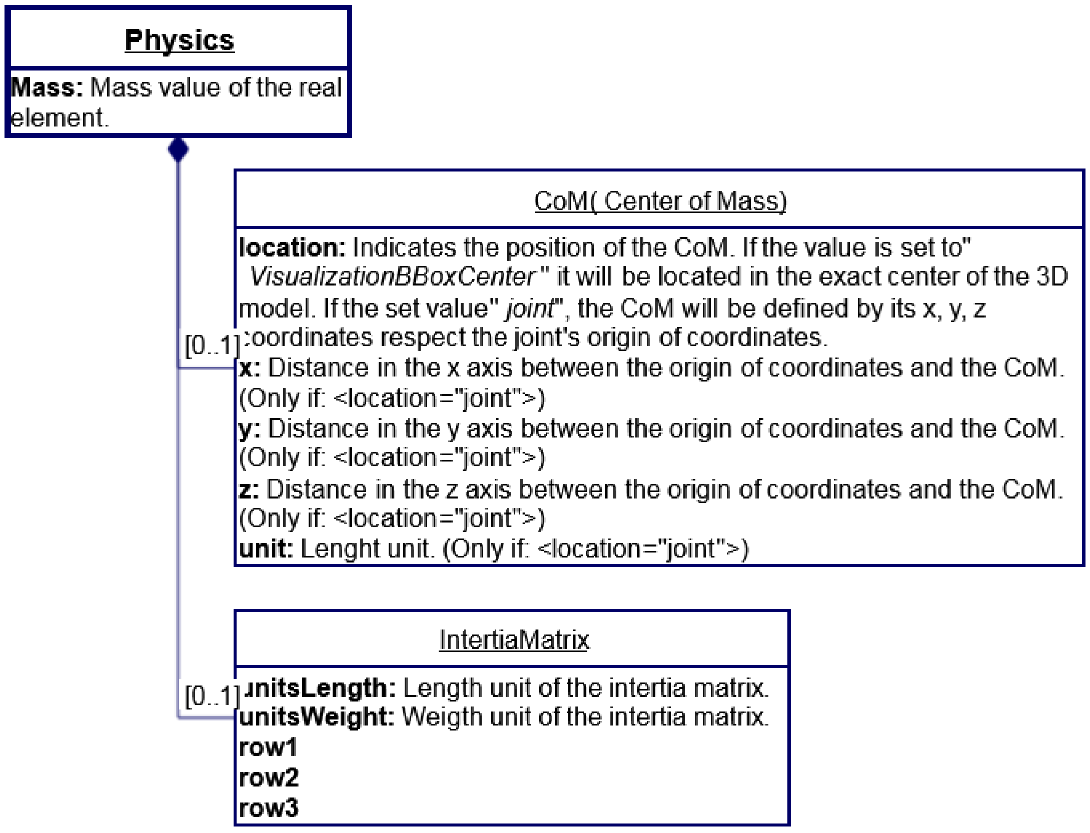

Physics
Mass: Mass value of the real element.
[0..1]
CoM(Center of Mass)
location: Indicates the position of the CoM. If the value is set to" VisualizationBBoxCenter" it will be located in the exact center of the 3D model. If the set value" joint", the CoM will be defined by its x, y, z coordinates respect the joint's origin of coordinates.
x: Distance in the x axis between the origin of coordinates and the CoM. (Only if: <location="joint">)
y: Distance in the y axis between the origin of coordinates and the CoM. (Only if: <location="joint">)
z: Distance in the z axis between the origin of coordinates and the CoM. (Only if: <location="joint">)
unit: Lenght unit. (Only if: <location="joint">)
[0..1]
IntertiaMatrix
unitsLength: Length unit of the intertia matrix.
unitsWeight: Weigth unit of the intertia matrix.
row1
row2
row3

Figure 5.4: *Physics* class diagram in Simox.

Parameter	**Available options**
unitLength, unit	”m”, ”meter”, ”mm” or ”millimetre”
unitsWeight	”g”, ”gram”, ”kg”, ”kilogram”, ”t” or ”ton”

Table 5.2: Additional information fo the class *Physics*.

5.3.4 Visualization

In most of the cases we will like to view the 3D model of our robot, to have a more accurate picture of how is it working in a simulation. Simox offers us the possibility to add a 3D model to our robot. Seems compulsory, but it is not: for example we only want to calculate paths and collisions, we can skip this part and just add the collision model and go straight to compute the process.

Simox supports **.wrl** (Virtual Reality Modeling Language), **.iv** (Inventor) and **.stl** (STereoLithography) files. TEO’s models are .wrl and will be the only file type used.

One interesting option is to use the main 3D model as out collision file, but is not recommended: collision models are simplified sketches, while main files are a fully-detailed files. If we use them as collision models the PC will have to load a big 3D file twice, and most certainly slowing the computing process. Another reason to use a custom expanded model is the difference between the 3D model and the real one: in the simulated one, cables and small boards attached to TEO are not taken in consideration, if we run a simulation with the exact same model, we can damage components that are not considered in the normal model.

The code in Listing 5.1 gives an example of how to refer to a file outside this XML.

```
<Visualization>
  <File type="Inventor">models/2.0^brazo_izquierdo_links.wrl</File>
</Visualization>
```

Listing 5.1: File importation example

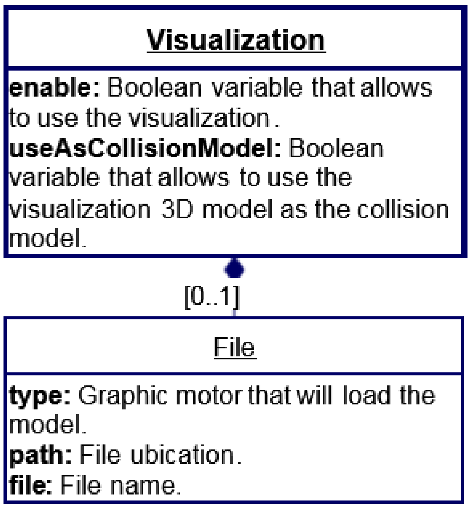

Visualization
enable: Boolean variable that allows to use the visualization.
useAsCollisionModel: Boolean variable that allows to use the visualization 3D model as the collision model.
[0..1]
File
type: Graphic motor that will load the model.
path: File ubication.
file: File name.

Figure 5.5: *Visualization* class diagram in Simox.

File format	*type*
.iv and .wrl	"Inventor"
stl	"osg"

Table 5.3: Additional information of the class *Visualization.*

5.3.5 Collision

One of the most interesting parts about Simox's robots definitions is the possibility to add a 3D collision model of our robot, instead of creating a whole application that calculates a expanded model in real time like the one developed in OpenRAVE in this department, we can just add another 3D file. Like said in the *Visualization* section is recommended creating another expanded model for this part of the simulation. Is also advisable simplify this model, just a 3D sketch of the space that TEO occupies.

We can refer to a file for the collision model in the exact same way as in the *Visualization* class, explained in Listing 5.1.

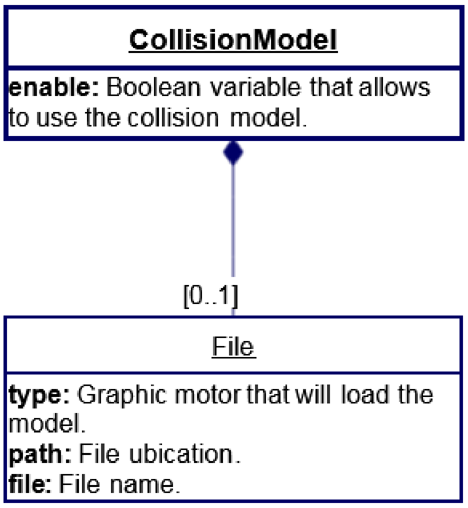

CollisionModel
enable: Boolean variable that allows to use the collision model.
[0..1]
File
type: Graphic motor that will load the model.
path: File ubication.
file: File name.

Figure 5.6: *Collision* class diagram in Simox.

5.3.6 Transform

The class *Transform* is indispensable for the construction of a robot: allows us to refer the position of the next *RobotNode* to the previous one. As shown in Figure 5.7, we have a wide range of possible transformations: a **4 by 4 matrix** or a rotation like a **quaternion**, a **3 by 3 matrix** or **RollPitchYaw**. Also a **Denavit-Hartenberg** transformation can be used. For TEO, the former method was used because all the parameters were already calculated and documented.

One thing to note here is that the coordinate systems differ from the main diagram in TEO. For Simox, coordinate axis were left unchanged, meanwhile in the other, there are changed in each articulation.

```xml
<Transform>
  <matrix4x4>
    <row1 c4="0" c3="0" c2="1" c1="0"/>
    <row2 c4="-146" c3="0" c2="0" c1="-1"/>
    <row3 c4="0" c3="1" c2="0" c1="0"/>
    <row4 c4="1" c3="0" c2="0" c1="0"/>
  </matrix4x4>
  <DH units="degree" theta="90" alpha="0" d="0" a="0"/>
</Transform>
```

Listing 5.2: Transformation example.

Parameter	Available options
unitsAngle, units	"rad", "radian", "deg" or "degree"

Table 5.4: Additional information fo the class *Transform*.

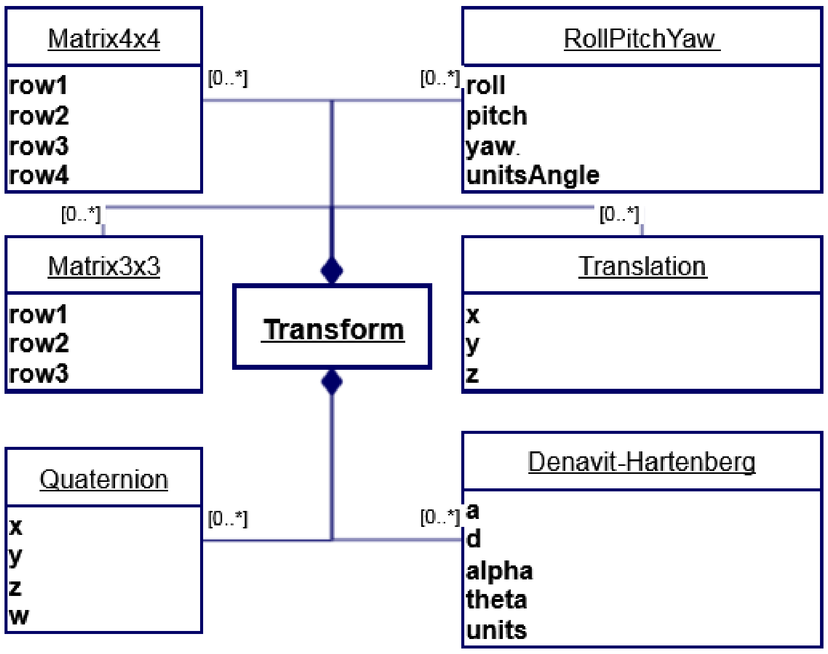


Figure 5.7: *Transform* class diagram in Simox.

5.3.7 Sensor

It is possible to add sensors in the definition of our robot, so we can use them in ArmarX directly. Actually there is only two types supported: *Position* and *Camera*.

This class can have a *Transformation* class on its own, when the sensor is not exactly located in the node.

5.3.8 Child

Robots are usually defined in a tree structure, with a starting part or root is set as initial and branches expand from it, creating the limbs. Simox follows the same rules, and the way to define them are with the *Child* class. A example can be seen in Listing 5.3, where two joints are related together.

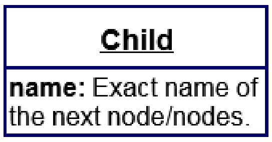


Figure 5.8: *Child* class diagram in Simox.

```xml
<RobotNode name="LeftAxialShoulder">
<Joint type="revolute">
<Axis z="-1" y="0" x="0"/>
<Limits units="degree" hi="55" lo="-50"/>
</Joint>
<Transform>
<DH units="degree" alpha="0" a="0" d="-329.01" theta="0"/>
</Transform>
<Visualization>
<File type="Inventor">models/2.2^brazo_izquierdo_links.wrl</File>
</Visualization>
<Child name="LeftFrontalElbow"/>
</RobotNode>

<RobotNode name="LeftFrontalElbow">
<Joint type="revolute">
<Axis z="0" y="1" x="0"/>
<Limits units="degree" hi="10" lo="-100"/>
</Joint>
<Visualization>
<File type="Inventor">models/2.3^brazo_izquierdo_links.wrl</File>
</Visualization>
<Child name="LeftAxialWrist"/>
</RobotNode>
```

Listing 5.3: Code fragment showing two consecutive *RobotNodes*

5.3.9 ChildFromRobot

As said before, a robot is defined as a tree, but sometimes we don't want to define the whole robot in a single file: if it has a considerable size, making changes can be a tedious task. For this reason, is recommended to split the robot in several files, for example, TEO is formed by a main file "*TEOSimox.xml*" which contains a reference to the other files, each one of them defines a limb.

You can refer to another XML file in the same way to a visualization file (See line 8, Listing 5.4).

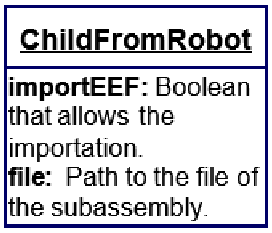

ChildFromRobot
importEEF: Boolean that allows the importation.
file: Path to the file of the subassembly.

Figure 5.9: *ChildFromRobot* class diagram.

```xml
<!-- Transformation to the BODY -->
<RobotNode name="TrafoToBody">
  <Transform>
    <DH units="degree" theta="0" alpha="0" d="193.2" a="0"/>
  </Transform>
  <ChildFromRobot>
    <File importEEF="true">xmlfiles/BodySimox.xml</File>
  </ChildFromRobot>
</RobotNode>
```

Listing 5.4: Code fragment that refers to a different XML file.

5.3.10 RobotNodeSets

When defining a robot in Simox we will end up with some *RobotNodes* that are only auxiliary, like transformations for external XML imports or **TCP**s (Tool Center point) so working with a lot of elements can be difficult and slowly. To solve this problem we can define a *RobotNodeSet*, which is a chain of *RobotNodes* that keep some kind of relationship, like a leg or the body articulations.

Having a handful of *RobotNodeSets* is suggested, you can work with a specific chain or even the whole body but without having auxiliary nodes around (See Figure 5.10). It is compulsory for later stages of the simulation to have at least one *RobotNodeSet*.

Defining a *RobotNodeSet* is a simple task, we just have to define the *root* node for the chain, a TCP and then writing down the exact names of the *RobotNodes* desired, as shown in Listing 5.5.

```xml
<RobotNodeSet name="RightLegChain" tcp="RightLegTCP" kinematicRoot="
    RightAxialHip">
  <Node name="RightAxialHip"/>
  <Node name="RightSaggitalHip"/>
  <Node name="RightFrontalHip"/>
  <Node name="RightFrontalKnee"/>
  <Node name="RightFrontalAnkle"/>
  <Node name="RightSaggitalAnkle"/>
</RobotNodeSet>
```

Listing 5.5: *RobotNodeSet* of the right leg.

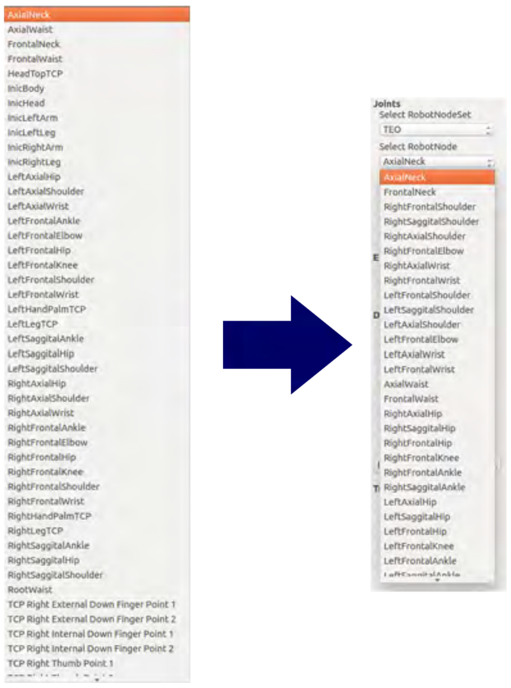


Figure 5.10: Example in Simox of the simplification achieved by making use of *RobotNodeSet*.

5.4 TEO Definition

One of the problems the department faced for a time was the difference of models, names and procedures used in various projects related with TEO, not all the teams used the same nomenclature or not even the 3D same models, summing up, a lack of standardization. Fortunately, this issue was solved long ago and now they share the same GitHub repository and there is no compatibility problems between different users. With that objective in mind, the aim was to follow the same tree structure and nomenclature, to a greater or lesser extent, used in the OpenRAVE models.

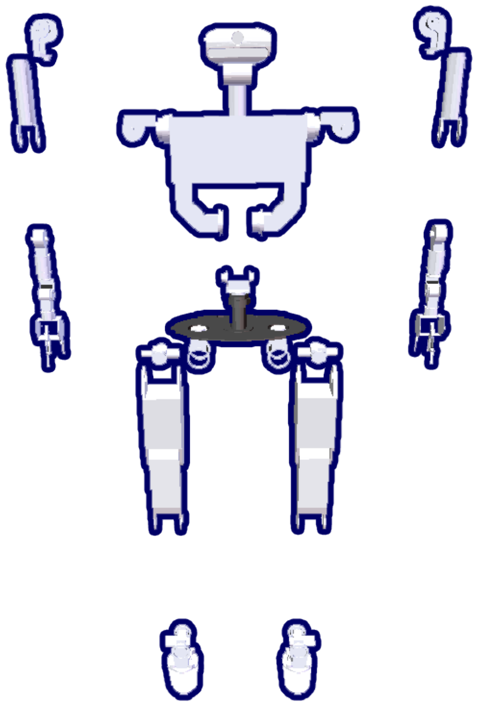


Figure 5.11: Expanded 3D model fo TEO in Blender, imported with *RobotEditor*, plug-in from H^{2}T.

5.4.1 Model Hierarchy

The hierarchy used in this model of TEO is the same in the rest of TEO's projects. It begins with setting-up a root in the pelvis, this will be our reference node. From it we will apply transformations to the following parts, the body and the legs. The body also follows a similar operation, from it we have the transformations to the arms and the head, and then the particular XML files are called.

In Figure 5.12 we can see the tree diagram followed. Each box is a different XML file and the arrows represent the auxiliary transformation previously mentioned.

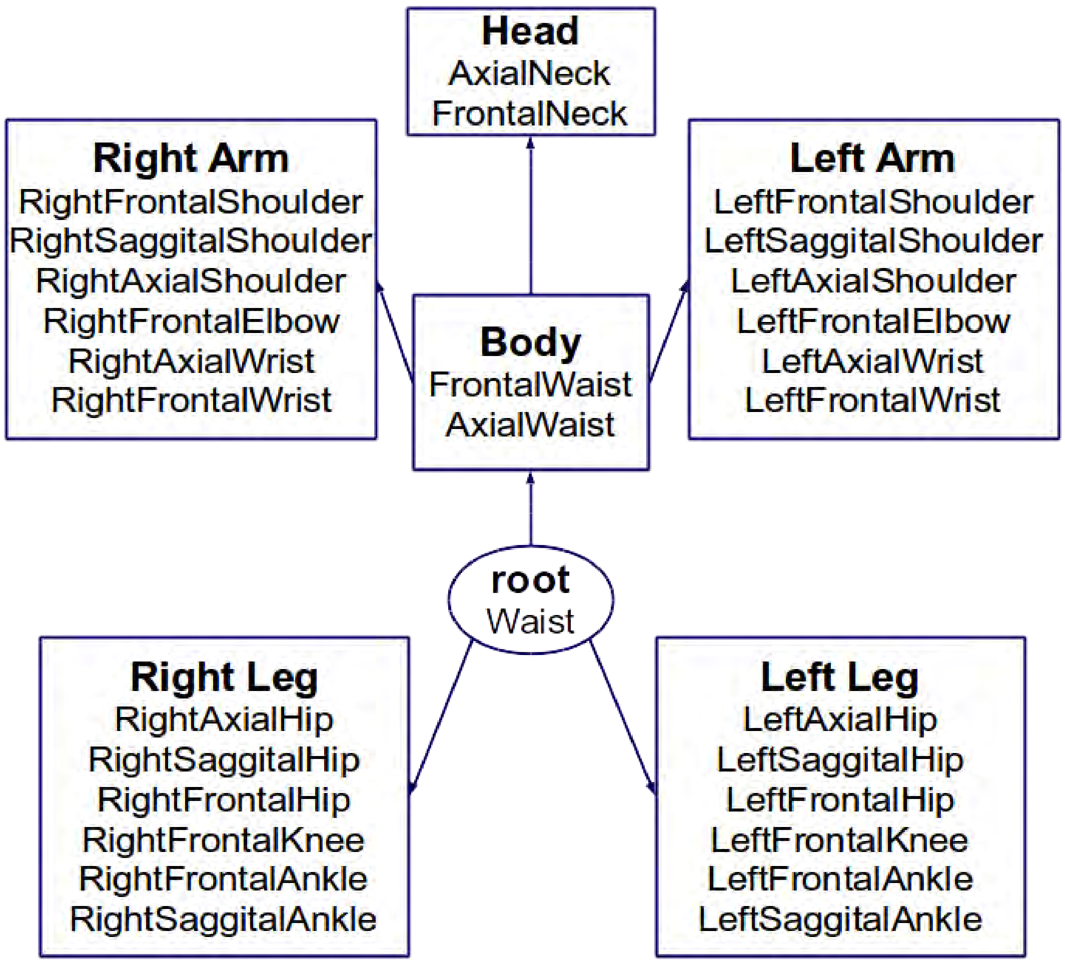


Figure 5.12: TEO's Hierarchy.

5.4.2 Definition process

This section will explain with more detail the process followed (and recommended) to define a robot.

Is highly advisable to start with something small, for example an arm. First we add the visualization so we have an idea of how is progressing.

Now we add one child, its visualization and the transformation from the first part to the next one. We add the visualization part so we can see each time we load in the part **RobotViewer** from Simox if we have the coordinates set correctly.

Repeating the process, the first limb is set. We can add parameters to have a more strict description of our robot: joint limits, parts masses, etc. Now we have one XML file with an arm like the one in Figure 5.13.

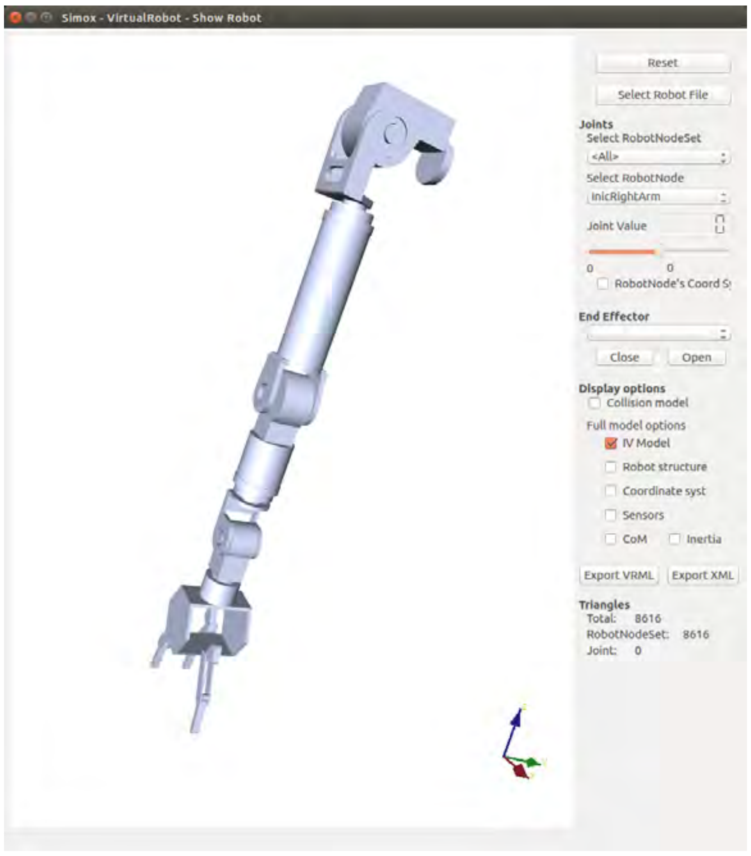


Figure 5.13: TEO's right arm.

After one limb is created, the remaining parts are a simple task, just repeating the same process. But the important part comes when we want to join them all in the same file. As exapmle ,like what was showed in Figure 5.12, The *Body* XML file holds both arms and head XML files inside it self. And in Listing 5.6 details the process: two auxiliary *RobotNodes* were defined, applied transformations to the shoulders initial points and then importing each arm XML file.

```
<RobotNode name="TrafoToRightArm">
  <Transform>
    <matrix4x4>
      <row1 c4="0" c3="0" c2="-1" c1="0"/>
      <row2 c4="-262.92" c3="-1" c2="0" c1="0"/>
      <row3 c4="305" c3="0" c2="0" c1="1"/>
      <row4 c4="1" c3="0" c2="0" c1="0"/>
    </matrix4x4>
    <DH units="degree" theta="-90" alpha="-90" d="0" a="0"/>
  </Transform>
  <ChildFromRobot>
    <File importEEF="true">RightArmSimox.xml</File>
  </ChildFromRobot>
</RobotNode>

<RobotNode name="TrafoToLeftArm">
  <Transform>
```

```
    <matrix4x4>
      <row1 c4="0" c3="0" c2="0" c1="1"/>
      <row2 c4="262.92" c3="0" c2="1" c1="0"/>
      <row3 c4="305" c3="1" c2="0" c1="0"/>
      <row4 c4="1" c3="0" c2="0" c1="0"/>
    </matrix4x4>
  </Transform>
  <ChildFromRobot>
    <File importEEF="true">LeftArmSimox.xml</File>
  </ChildFromRobot>
</RobotNode>
```

Listing 5.6: Arms referencing.

5.4.3 Complete Definition

This process was repeated with all limb sub-assemblies: the legs, the head and the arms; resulting in the whole model (See Figure 5.14) ready to work in ArmarX.

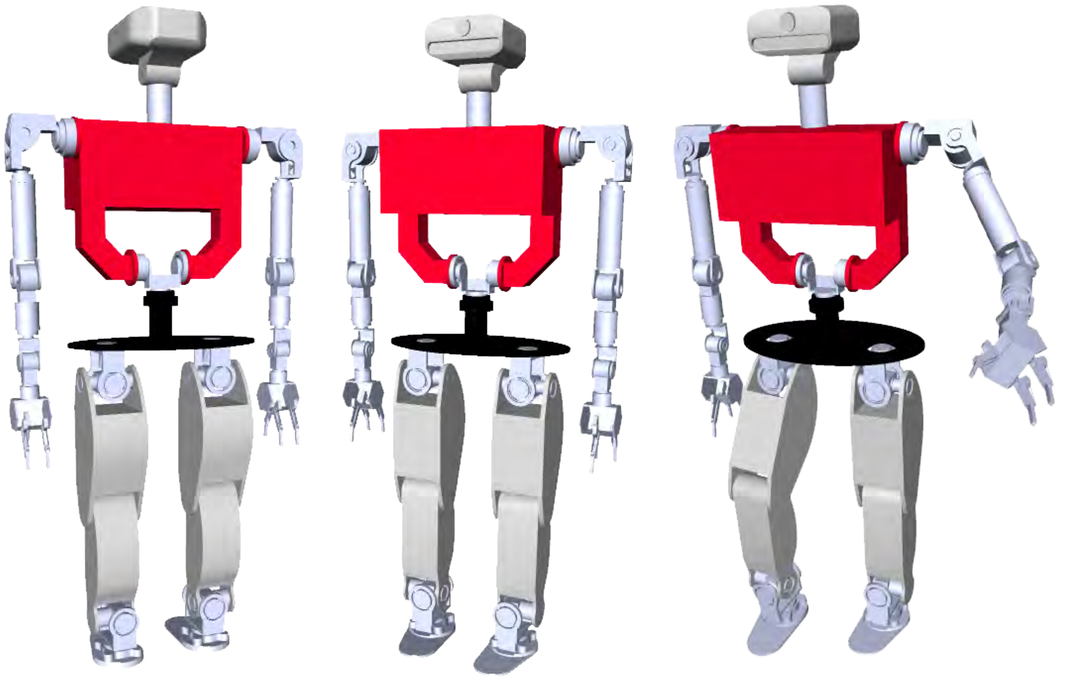


Figure 5.14: TEO in Simox.

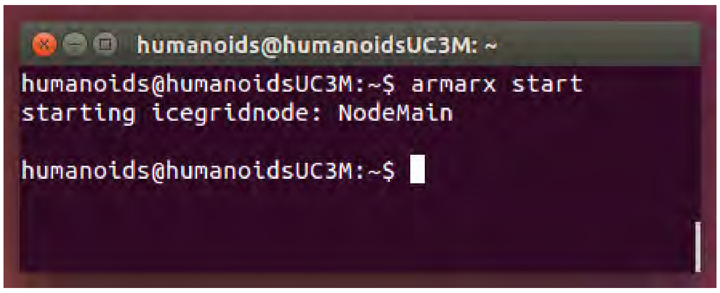
6 Simulations in ArmarX

This section will cover the process of running a simulation in ArmarX: from the first steps in the environment, to the final phase with TEO executing previously defined tasks. Here is documented the main objective of this work: **simulating a task in ArmarX for TEO**, and detailed information like custom parameters or exact procedures are listed in the Appendix.

6.1 Getting Started

One thing to note for the remaining parts of this project is the exclusive use of **Linux** operating system, in particular **Ubuntu** distribution, version **14.04**. The main reason behind this decision is that ArmarX is only supported on it. A worth mention here is that working with Linux in this environment is not a downside compared to Windows. Even an average user, who is more familiar with Microsoft operating systems. For our project the ease to work with repositories, files and compiling directly from our console is a great advantage.

As were explained previously on this document, ArmarX works over an **Ice** grid, so for the rest of the project we must deploy first the nodes to work with ArmarX. With this simple command ArmarX is ready to run.



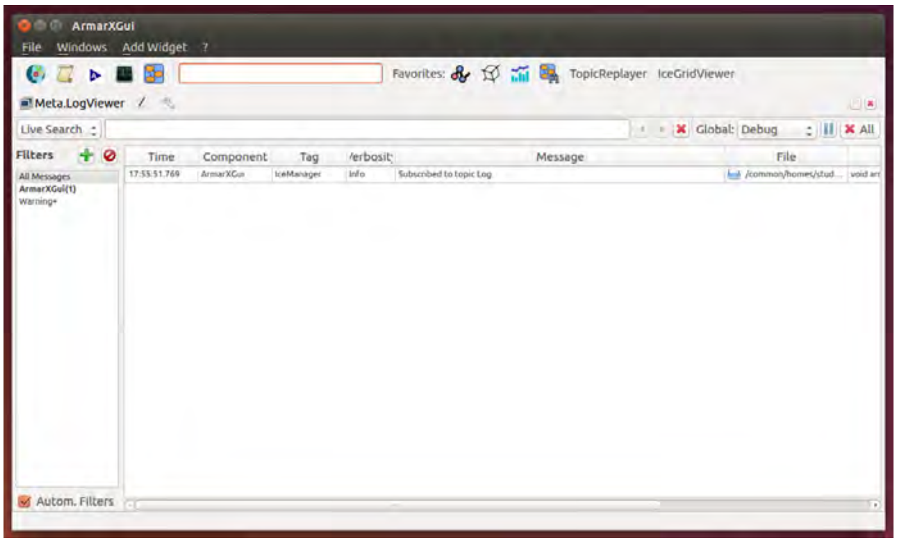
```
humanoids@humanoidsUC3M:~$ armarx start
starting icegridnode: NodeMain

humanoids@humanoidsUC3M:~$ 
```

Figure 6.1: ArmarX is running.

6.2 ArmarX Gui

ArmarX provides us with a custom **Gui** (Gaphics User Interface) that will allow us to interact with its plug-ins (VisionX, StateChart Editor, etc), while most of the internal processes go under the user's interface but still accessible with the loggers and the focus in debugging by the developers.



Time	Component	Tag	'erbosit	Message	File	
17:55:51.769	ArmarXGui	IceManager	Info	Subscribed to topic Log	/common/homes/stud...	void ar

Figure 6.2: ArmarXGui first impression.

This interface will give us access to the *ready-to-use* plug-ins, called **Widgets** within the program. Here we can use the, previously exposed in this document (VisionX, MemoryX, etc), and their sub-components. Also other main APIs like visualization and simulation can be programmed easily to report us the results of our projects.

6.3 ArmarX Packages

Having a good projects organization is mandatory, whether they are big or small, for a clear review and adding more components to them. ArmarX projects will hold several of this elements offered by the ArmarX framework. A way to organize all of these components with its source code and documentation in ArmarX is with its **packages**.

There is no file that describes a package, it is defined by its structure. Package's organization is based on CMake and can be modified or expanded. ArmarX includes **armarx-package** tool that will do most of the work for us. Once created a package we have to compile it and its ready to use in the framework.

For this project was chosen to make TEO wave one hand in the simulation, so a package called *TEOMove* was created and, after compiling it, we will create the proper StateChart.

6.4 Robot Profiles

ArmarX lets us build our simulation scenarios over custom robot profiles, which are databases with predefined parameters that simplify the work in our robot. f there is not a parameter defined in our current profile, ArmarX will climb up in the hierarchy tree until it founds this parameter defined.

Figure 6.3 we can see how it the program will start searching for default parameters: if we are working with *TEOReal* and the value for a variable is not defined there, it will try to search it no *TEOBase* and if is neither here, will try in the final default profile *Root*.

The profile structure for TEO was imitated from the one for Armar3, with a general profile, *TEOBase* and two additional profiles *TEOSimulation* and *TEOReal*, in case that were necessary to implement different parameters between the simulated and the real model.

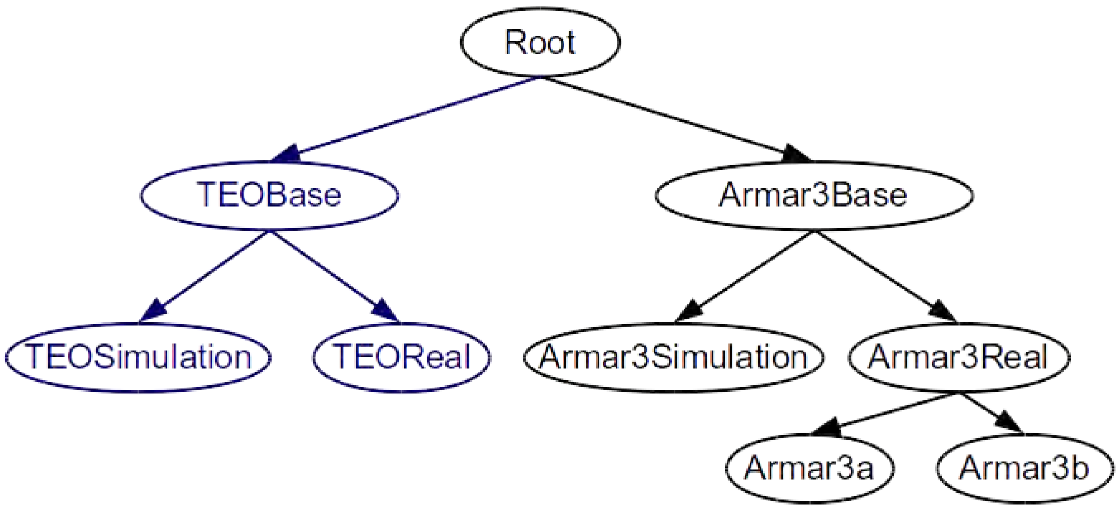


Figure 6.3: ArmarX profile hierarchy. The black part is the default profiles ArmarX comes with, and in blue, the ones added for this project.

6.5 StateChart Editor

The StateChart Editor is one of the main features of ArmarX. That is because all robot applications will use one statechart as a backbone for its logic: it will tell the robot which action it should do and in which orded.

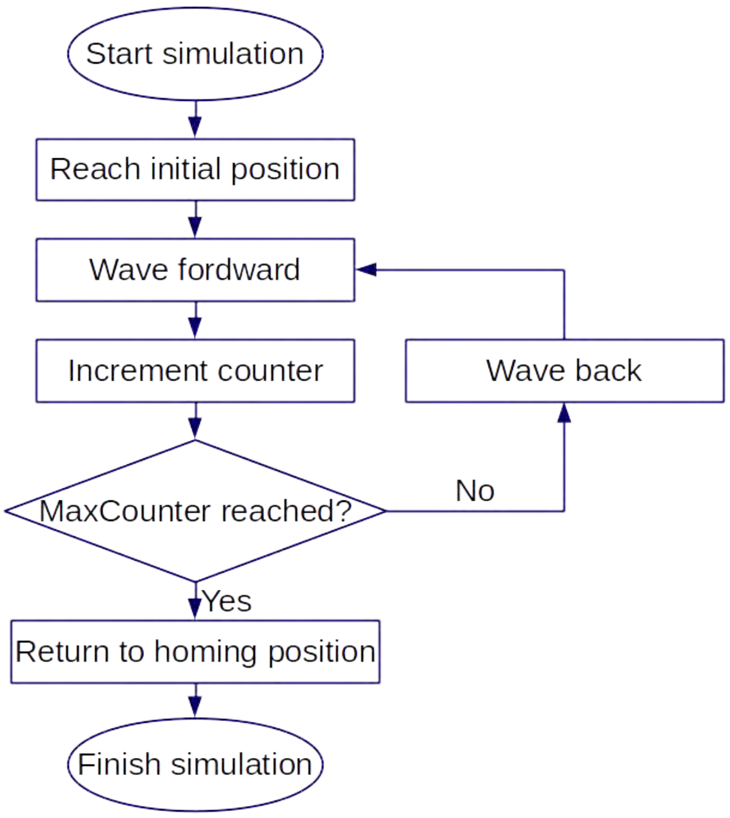


Figure 6.4: Final simulation process.

Figure 6.4 shows the course of action that TEO will follow in the simulation: From the rest pose, it will reach a initial position in order to make the movement of waving. It will move back and forward a number of times, in this case 4. It will start moving his hand forward. After completing this movement, ArmarX will evaluate if the value of a local parameter (counterId) has reach the maximum value of 4. Once the cycle is completed, TEO will come back to its resting (also named homing) position.

With the information about the statecharts given in Section 4.4.3, a statechart that follows the previously described flowchart is built and shown in Figure 6.5.

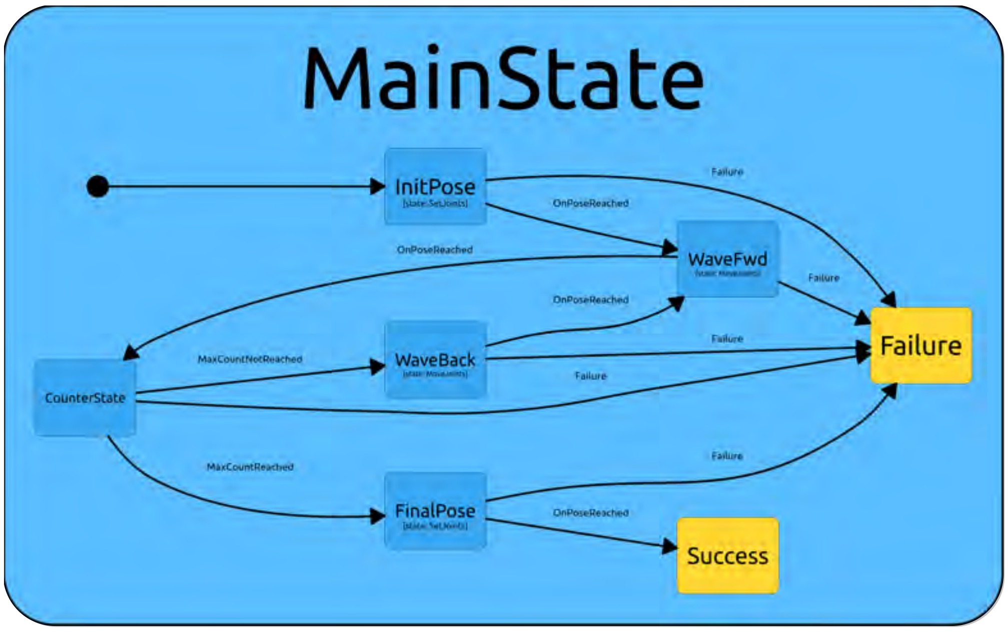


Figure 6.5: Final MainState StateChart.

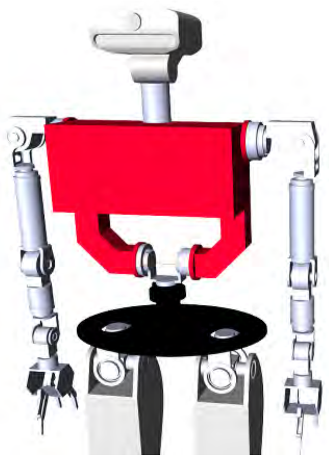
Where *OnPoseReached* are the transitions that point when a movement has reached it desired angular position. *MaxCountReached* and *MaxCountNotReached* are the ends of the decision in the flowchart.

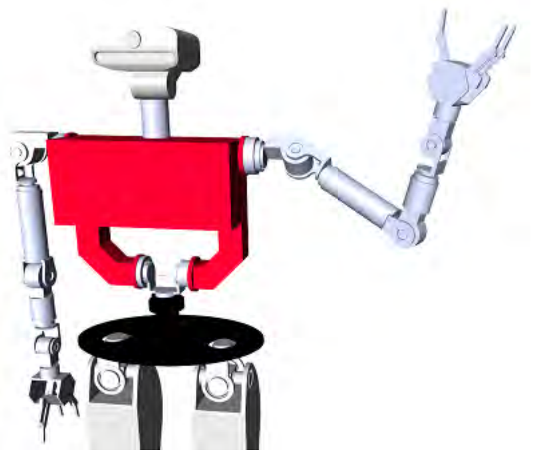
The failure state is not contemplated in the flowchart, but it is inherent in the design of every automatic system: if some transition cannot be made, the system must return to a safe state where it can be started again. In ArmarX's statecharts, upon being a high level and close to result syntax, must be included.

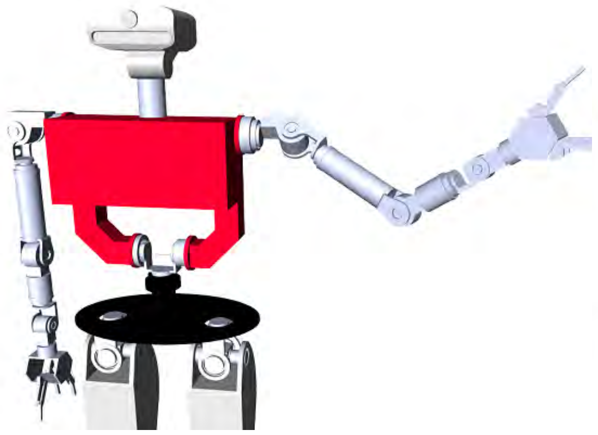
6.6 Kinematic Simulation

After adding all parameters and options to the ArmarX environment, we can begin to check our results. For that goal, we will use ArmarX's kinematic units. To use them, a scenario was created and all the applications required were added. Finally, it is possible to see TEO completing the task we programmed: waving. Figure 6.6 is the final result of TEO in Armarx.

For loading the 3D environment, ArmarX uses an already defined tool established within the architecture of ArmarX: *Armar3KinematicUnit*. This mechanism is responsible for loading the model and provides information to the programmer.


(a) Initial pose.


(b) TEO waving back.


(c) TEO waving forward.

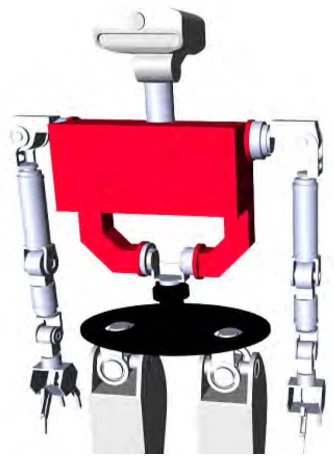

(d) Final pose.

Figure 6.6: From left to right and from top to bottom, the simulation of TEO waving. Note that (b) and (c) do repeat four times.

The kinematic observers also include a visual tool to monitor all joint parameters. ArmarX can directly provide in real time, if correctly parametrized, all joint information: angular position, velocity, acceleration, torque and current in the motor. Also can simulate temperatures of this motors.

For this demonstration, TEO was simulated alone, with the purpose to see its reaction and then compare with the real model. Nonetheless, ArmarX can incorporate complex scenarios for the robots, e.g. a kitchen.

Figure 6.7 shows angle position values (in radians). Being a simulation gives it that sharp and exact values. Here it is discernible the elbow going back and forward. The rest of the joints began at the homing position and reach their respective initial position for the duration of the waving. Some of them go in the negative part, this is due the initial coordinate system chosen when TEO was developed.

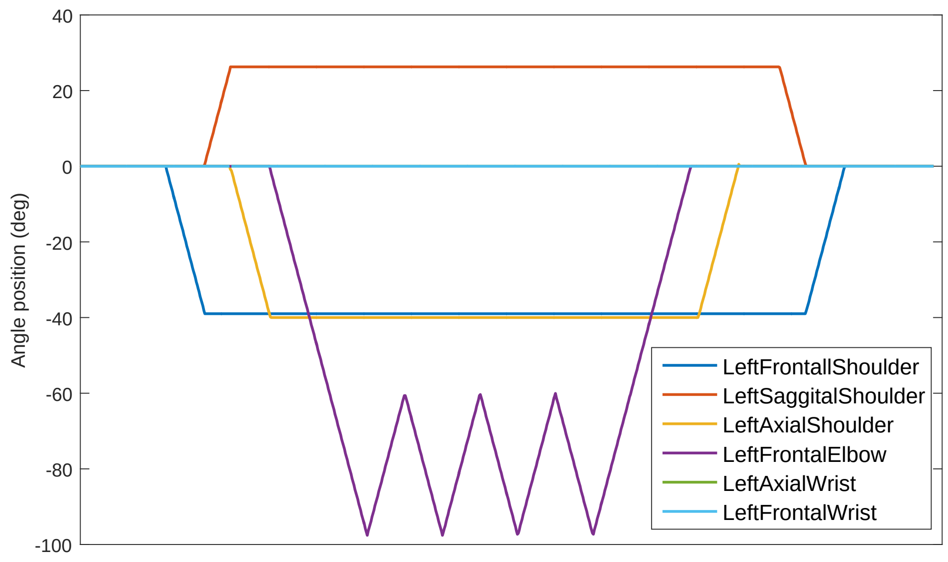
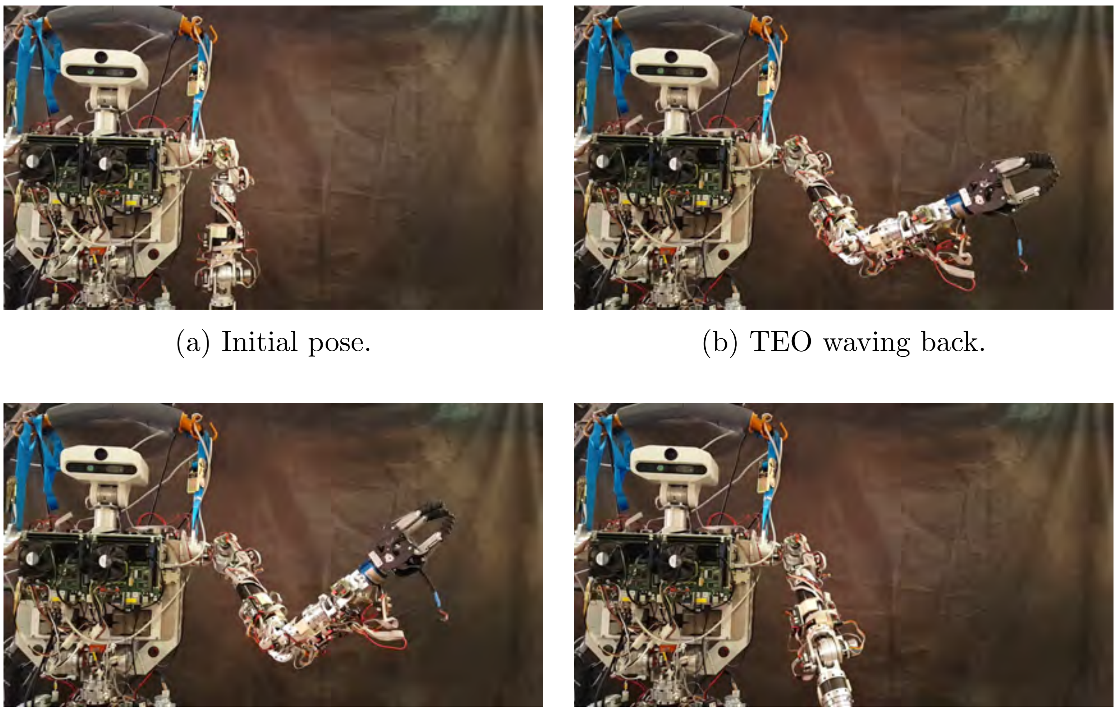


Figure 6.7: Joint angle values during the simulation.

6.7 Results

With the results obtained in Section 6.6, it is possible to load them in the real model of TEO. To do so, the department have already developed tools with the middleware YARP. This RDE is the main framework of TEO, provides port access and communication between the components such as actuators, PCs or sensors. Figure 6.8 shows succession of the real TEO performing the task defined in ArmarX.



(a) Initial pose. (b) TEO waving back.

(c) TEO waving forward. (d) TEO reaching its final pose.

Figure 6.8: From left to right and from top to bottom, the real TEO waving. Note that (b) and (c) do repeat four times.

YARP also provides a standard data recorder. it is possible to directly load the joint values from the encoders. Figure 6.9 is the plotted data of these values. We can see some expected differences:

- First, the slower response speed of the actuators. Especially in the *LeftFrontalElbow* repetitive movement. In the simulation, this part does abrupt changes (Figure 6.7). The problem comes when the real devices try to follow this quick changes: the controller is orders of magnitude faster then the actuators, so it goes several steps ahead the real-world movement.

- The appearance of real errors that are not considered in the ideal simulation, such as noise or actuators precision. An example to this appreciation can be seen at the beginning, right before any motor starts to move.

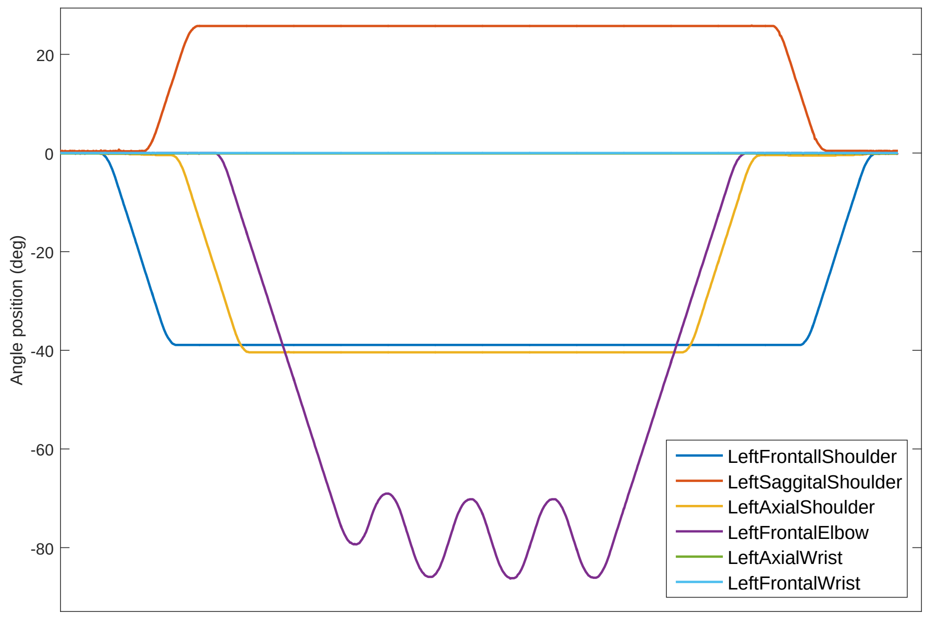


Figure 6.9: Joint angle values during the experiment on TEO.

7 Conclusions

ArmarX is presented as an outstanding robot framework. Provides a huge range of opportunities to the developer: creating an object recognizer, programming a robot to incorporate knowledge to its data base, planing trajectories, etc. In this work, only a slight fraction of its potential was used. Encouraging students to start learning this tool can result in new and elaborated applications for TEO.

Figure 7.1: ArmarX framework logo.

The humanoid robot of our university, TEO, still holds most of its potential. The University Carlos III has an astonishing resource at its disposal: a full sized humanoid robot with vision and grasping capabilities. Nonetheless, working in the field of robotics also highlights the difficulties and general slow speed that robot science has. In the other hand, assigning more projects to TEO can result in the development of new applications for TEO. This programs can contribute to make a name for TEO and the UC3M in the field of humanoid robots.

7.1 Contributions

Also, this project has created a complete model of TEO to be used in the ArmarX framework. It contains the 3D models and the robot definition ready to be used. This files can be found in the GitHub repository of RoboticLabs and can be freely accessed.

This project is presented as a guide for the first steps in the ArmarX environment. Hence, the idea is to become a guide for other students to increment the applications in TEO.

7.2 Future Works

This work is the first stage of the bigger project to create elaborated tasks that can be used in human environments, thus is the ultimate objective of all humanoid robots.

It was mentioned various times during this work, the immense opportunities ArmarX can provide. However, some can report more immediate results or integrate with the existing ones:

- Creating a collision model of TEO in the ArmarX environment. A expanded 3D model of TEO can be used to calculate collision-free trajectories.

- Adding the sensors and the cameras already integrated in the real robot to its ArmarX definition.
- Importing skills already developed by other teams, ArmarX applications are easy to import into another robot.

8 Project Overview

This section includes a review about the organization of this project, the time schedule and the cost the work-time and materials used.

8.1 Budget

8.1.1 Material Costs

This section includes the costs in both software and hardware platforms:

- **Software:** One of the guidelines of the project was to use as many open-source platforms as possible. At the end of the work, only free software was used.

Concept	**Amount (€)**
Ubuntu 14.04	0
ArmarX	0
Simox	0
YARP	0
TeXstudio	0
LibreOffice	0
Gimp	0
Total costs	0

Table 8.1: Software costs.

- **Hardware:** Includes two different aspects, the computers to simulate and used to write this document and the robot TEO:
  - **TEO:** The total cost invested in TEO cannot be measured. It compromises not only the hardware, which has been improved and expanded over numerous projects; but the maintenance and work-hours invested in it.
  - **Computers:** Most of this thesis was elaborated in the university, using a workstation in the *RoboticsLab* department. In addition, a personal computer was used.

Concept	**Amount (€)**
University's workstation	1000
Personal Asus Laptop	700
Total costs	1700

Table 8.2: Hardware costs.

8.1.2 Personnel Costs

This section includes the costs in the personal involved in this project:

Concept	Per hour cost (€/h)	Work hours	Amount (€)
Bachelor student	10	500	5000
Doctorate researcher	20	100	2000
Laboratory technician	15	30	450
Titular professor	40	20	800
Total costs			8250

Table 8.3: Personnel costs.

8.1.3 Total Costs

Adding both personal and material costs, the final project amount invested is 9950 € over a period of 7 months.

Concept	Amount (€)
Software costs	0
Hardware costs	1700
Personnel costs	8250
Total costs	9950

Table 8.4: Total costs.

8.2 Project Schedule

The tasks performed in this project could be grouped in three global activities, two are related with the learning of ArmarX and its components. The remaining consists in collecting all documentation of the processes followed in the other parts and comparing results. The Gantt diagram if Figure 8.1 resumes these process.

- **ArmarX:** This part includes intensive research and development with the framework. Compromises almost the complete of the project duration.
    - First a initial research about the software itself was made.
    - Once program was installed and ready to use, the following moths were invested in training with it, using some tutorials already made by the KIT.
    - Finally, when I was able to develop my own applications in ArmarX, the *Waving* simulation was created and loaded in the real robot.

- **Simox:** After realising the necessity of creating TEO models compatibles with ArmarX, they were created in this part.
- **Document Composition:** The last section of the work consists in including all the knowledge acquired and creating this report. Nonetheless, it was originally intended to be written with Microsoft Word, but the idea was discarded and a LaTeX environment was used.

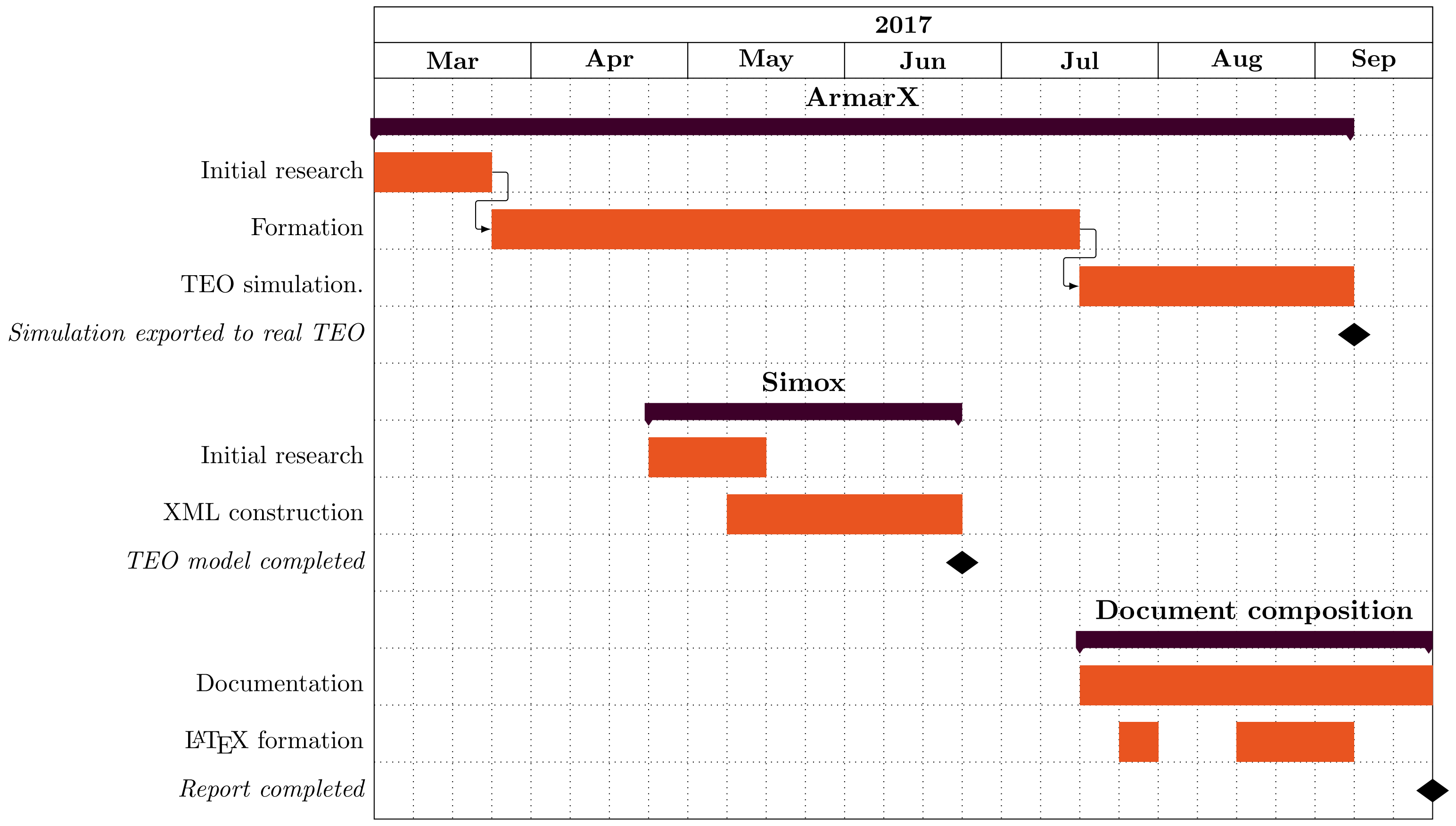


Figure 8.1: Gantt diagram for this thesis.

Appendices

A Robot Definition

This annex will explain and carry the process of creating a new robot ready for the ArmarX environment. Already in Section 5 general information about the capabilities was discussed and some examples were shown. Nonetheless here is a suggested *modus operandi* to build a XML file that describes a robot, in this case TEO.

It is also compulsory to have the 3D models of the pieces of our robot. Recommended to be visualized beforehand in a 3D development environment, such us Blender or SolidWorks, the former was used to create the design of TEO.

As mentioned in Section 5, we will use Simox tools to visualize and check the results of our robot assembly, specifically **RobotViewer** [23]. This tool inside Simox will allow us to see, each time we load our robot, how are we constructing it. Unfortunately, RobotViewer does not provide extensive information about errors when we made one. Due to that, is recommended to check every few additions, in order back trace our mistake.

In order to correctly set all TEO parameters such as: articulation distances, joint revolution directions, etc. We will follow its diagrams of Figure A.1.

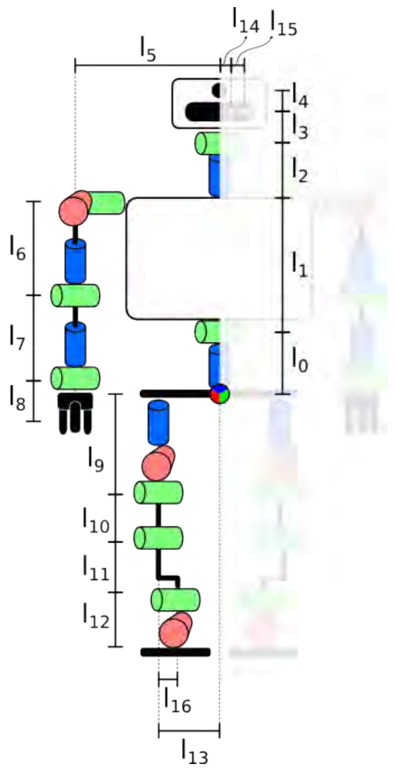


Figure A.1: TEO's Link Lengths.

Link length	Distance
l0	193.2
l1	305
l2	162.5
l3	59.742
l4	37.508
l5	346.92
l6	329.01
l7	202
l8	187.496
l9	92
l10	330
l11	300
l12	123.005
l13	146
l14	18
l15	26
l16	17.5

Table A.1: Link distance in milimeters, asociated with Figure A.1.

A.1 Root, Child and Visualization

We are going to begin our robot with its definition, we will declare a robot. In an empty XML file, we will write the code in Listing A.1. We have created our first robot. It only possess one node, its root node. In line 1, the *Type* field contains the name of the robot. This first file will be named **TEOSimox.xml**.

```
1 <Robot RootNode="root" Type="TEO">
2
3   <RobotNode name="root">
4   </RobotNode>
5
6 </Robot>
```

Listing A.1: Most basic robot in Simox

If we load this robot in RobotViewer, Figure A.2 is the result.

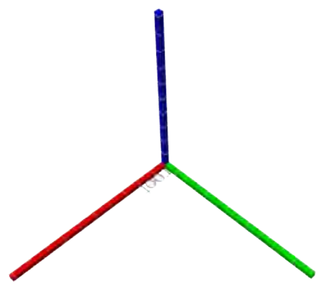


Figure A.2: Basic robot visualization in RobotViewer.

Since the objective is to build TEO's model for ArmarX, it is convenient to begin with its origin, the hip. In later stages, we will use the *root* as origin for other parts of TEO. For this project it was decided to leave it as clean as possible: only declaring children that will include the rest of the robot. The hip part that corresponds to the root node will be declared in a *child*. Creating a child is showed in Listing A.2. We just have to add the line 4 and then declare the next *RobotNode*, here called *RootWaist*. The child must be also be declared, even if its empty, to avoid error when loading the robot XML file from RobotViewer.

```
1 <Robot RootNode="root" Type="TEO">
2   <RobotNode name="root">
3     <Child name="RootWaist"/>
4   </RobotNode>
5
6   <RobotNode name="RootWaist">
7   </RobotNode>
8
9 </Robot>
```

Listing A.2: First RobotNode child.

Now we have two nodes, we should begin adding some visualization. By now we have only seen two overlapping coordinate systems. To add 3D pieces to our robot, we must use the *Visualization* class. When declaring a Visualization, we must refer to its path from the main XML file. **Incluir imagen de las carpetas**. Listing A.3 shows in line 8 how to include a 3D file to visualize. Figure A.3 is the result of Listing A.3 in RobotViewer.

```
 1 <Robot RootNode="root" Type="TEO">
 2
 3   <RobotNode name="root">
 4     <Child name="RootWaist"/>
 5   </RobotNode>
 6
 7   <RobotNode name="RootWaist">
 8     <File type="Inventor">xmlfiles/models/cintura_links.wrl</File>
 9   </RobotNode>
10
11 </Robot>
```

Listing A.3: Adding the Visualization.

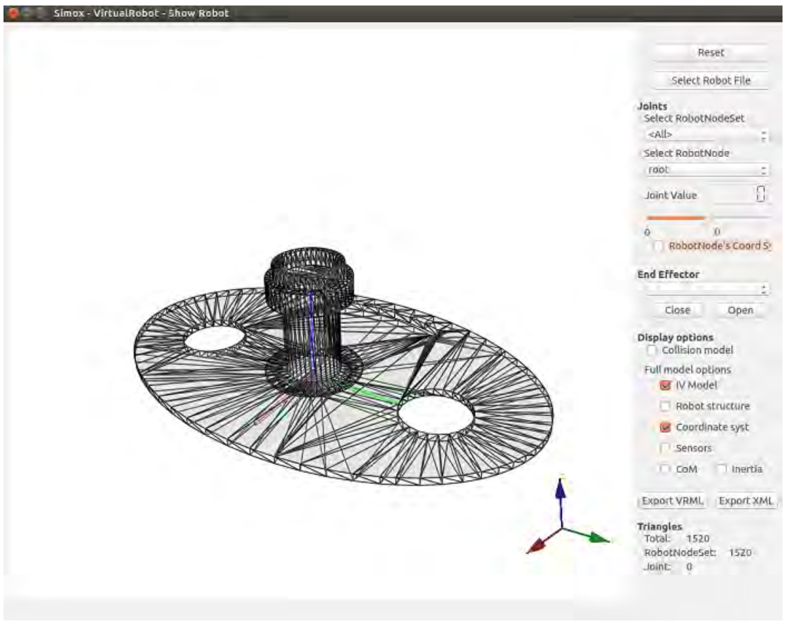


Figure A.3: Waist 3D visualization in RobotViewer. The visualization mode was changed because this exact piece is black and has no texture. So it shows the mesh.

A.2 Transformation and RobotNodeSets

As mentioned in Section 5.3, usually, a robot is not hold in a single file, they can become too complex and sometimes we want to make changes only in one part of the robot. Also in Section 5.4 it was mention the structure that the individual

XML files will have.

For now, we will leave the structure we created in Section A.1. Our focus will be creating a kinematic chain, applying transformations to a series of *RobotNodes* in order to create a limb. We will start with the **Left Leg**. This process will be applied to the rest of articulations. Each one will be included in a separate XML file.

The structure that each limb will include is a RootNode, for initializing, followed by the consecutive joints. Then we will add the transformations and visualization. Additionally, joint parameters will be added. When the chain is complete, a final TCP (Tool Centre Point) will be created. Once all this RobotNodes are finally created, they will be join in a kinematic chain, named *RobotNodeSet*. We will start wi the **Axial Hip** joint.

```xml
1  <Robot RootNode="InicLeftLeg" Type="LeftLeg">
2
3    <RobotNode name="InicLeftLeg">
4      <Child name="LeftAxialHip"/>
5    </RobotNode>
6
7  <!-- Link 0 -->
8
9    <RobotNode name="LeftAxialHip">
10     <!--<Transform>
11       <DH units="degree" theta="0" alpha="90" d="0" a="0"/>
12     </Transform>-->
13     <Visualization>
14       <File type="Inventor">models/5.0^pierna_izq_links.wrl</File>
15     </Visualization>
16   </RobotNode>
17
18 </Robot>
```

Listing A.4: First steps in creating TEO's left leg.

In Listing A.4 we have added the initial RobotNode of the left leg. Also, its first child and its visualization. Lines 10-12 perform a DH (Denavit-Hartenberg) transformation that resets the transition 4x4 matrix from the *root*. By now is not required so we will leave it commented.

Now we can begin joining two parts together, to do so, we declare a *Child*, the next is the **Saggital Hip** articulation. Also we can start adding some joint information. In Listing A.5 the joint directions and limits were added.

```xml
<Robot RootNode="InicLeftLeg" Type="LeftLeg">

  <RobotNode name="InicLeftLeg">
    <Child name="LeftAxialHip"/>
  </RobotNode>
```

```

<!-- Link 0 -->

  <RobotNode name="LeftAxialHip">
    <Joint type="revolute">
      <Axis x="0" y="0" z="-1"/>
      <Limits lo="-90" hi="90" units="degree"/>
    </Joint>
    <!--<Transform>
      <DH units="degree" theta="0" alpha="90" d="0" a="0"/>
    </Transform>-->
    <Visualization>
      <File type="Inventor">models/5.0^pierna_izq_links.wrl</File>
    </Visualization>
  </RobotNode>

<!-- Link 1 -->

  <RobotNode name="LeftSaggitalHip">
    <Joint type="revolute">
      <Axis x="-1" y="0" z="0"/>
      <Limits lo="-20" hi="20" units="degree"/>
    </Joint>
    <Transform>
      <DH a="0" d="-92" alpha="0" theta="0" units="degree"/>
    </Transform>
    <Visualization>
      <File type="Inventor">models/5.1^pierna_izq_links.wrl</File>
    </Visualization>
  </RobotNode>

</Robot>
```

Listing A.5: Two pieces joined together and with joint parameters.

If we load this file in RobotViewer, Figure A.4 is the result we obtain.

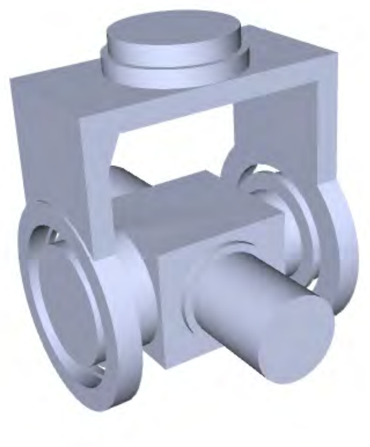


Figure A.4: Two joints loaded together.

Information about all of the joints revolution directions can be found in Figure A.5. The TEO diagrams for OpenRAVE and YARP change the system coordinates of each joint. In contrast, Simox does not. Even though, the rotation angles do not vary thus are referenced to the root coordinate system.

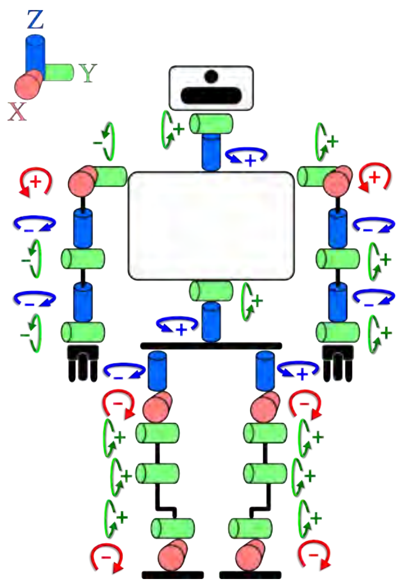
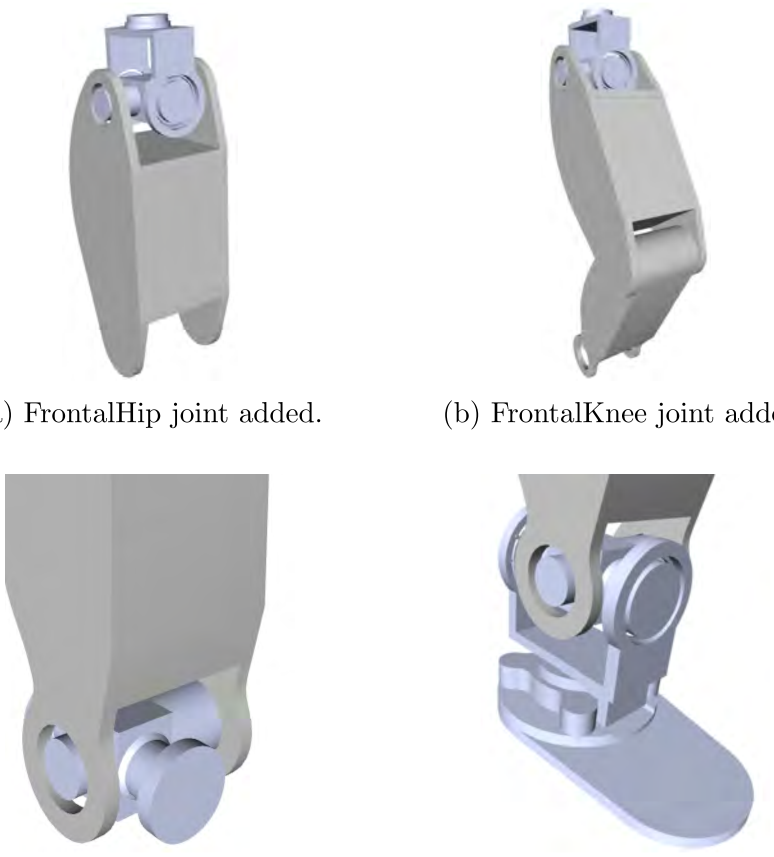


Figure A.5: TEO's joints directions.

Once two articulations are joined together, adding more to the chain is a repetitive task: declaring a child, adding its joint parameters, its transformation related to the past joint and finally adding the visualization. The process of adding all the remaining articulations of the left leg are summed up in Figure A.6.



(a) FrontalHip joint added. (b) FrontalKnee joint added.

(c) FrontalAnkle joint added. (d) SaggitalAnkle joint added.

Figure A.6: Process of building the leg step by step

A.3 RobotNodeSets

Once we have defined a limb, it is a standard procedure in robotics to add a TCP (Tool Control Point). This can be done by just simple create a RobotNode at the end of our chain and apply a transformation to ir so it is located at the physical end or the limb. For the legs, the TCP was defined in the points closest to the ground from the last joint. This points are found marching from the last RobotNode in the negative part of the Z axis. In Listing A.6, the TCP is defined.

```
<RobotNode name="LeftLegTCP">
  <Transform>
    <DH a="0" d="-123.005" alpha="0" theta="0" units="degree"/>
  </Transform>
</RobotNode>
```

Listing A.6: TCP created.

Now we have all the requirements to create a kinematic chain (*RobotNodeSet*): a root node, one or more intermediate nodes and a TCP. Listing A.7 shows how the left leg chain is declared.

```xml
<RobotNodeSet name="LeftLegChain" kinematicRoot="LeftAxialHip" tcp="
    LeftLegTCP">
  <Node name="LeftAxialHip"/>
  <Node name="LeftSaggitalHip"/>
  <Node name="LeftFrontalHip"/>
  <Node name="LeftFrontalKnee"/>
  <Node name="LeftFrontalAnkle"/>
  <Node name="LeftSaggitalAnkle"/>
</RobotNodeSet>
```

Listing A.7: Two pieces joined together and with joint parameters.

This same process was followed for each limb. In the end, we created both arms, legs and the head in a separate XML file. Each one of them were aggregated in a RobotNodeSet.

A.4 Assembling a Complete Robot

We as mentioned in Section 5, our robot will consist in a series of XML files referenced from a root file. Now, we will come back to our first file *TEOSimox.xml* and call the rest of them from it. Actually, we will recall the legs and the body from the root, then, the trunk will address to the arms and the head. This structure is described in Figure A.7.

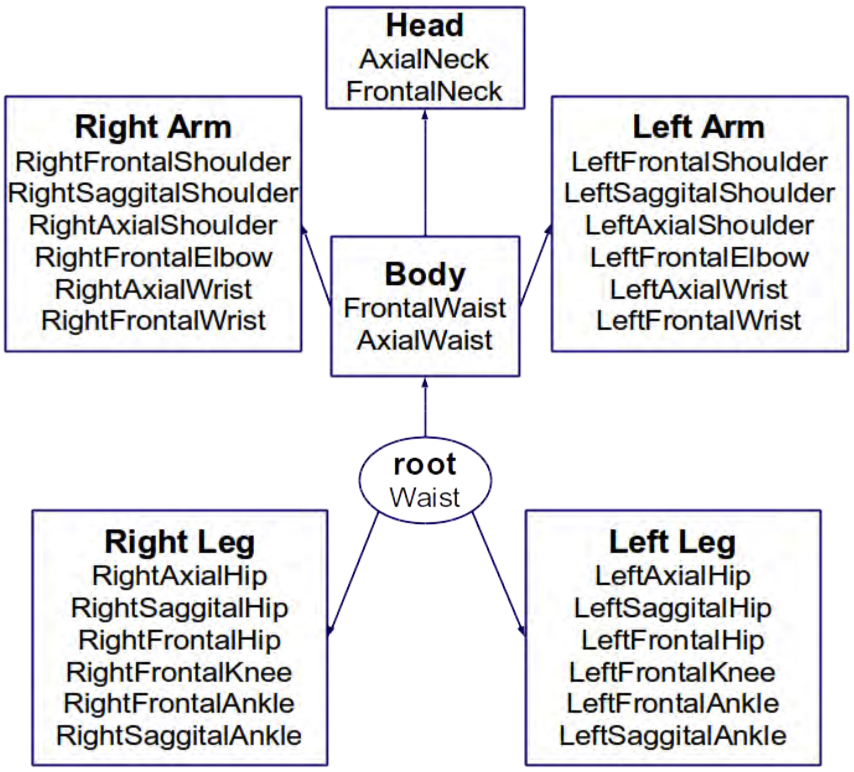


Figure A.7: TEO's Hierarchy.

The folder structure followed for TEO is quite similar to the one with the OpenRAVE files. Here we will have the main file, *TEOSimox.xml*, and a folder that contains the remaining parts of the robot. Also we can find the folder *models*, that contains the 3D models.

Listing A.8 describes an auxiliary *RobotNode*: *TrafoToLeftLeg*. This node will allow us to place the left leg in its real position. The transformation is made by a 4x4 matrix and a DH translation. Line PI is the interesting command: *ChildFromRobot* will add the first *RobotNode* in the file it imports as a child. Due to this nature, an hypothetical approach is to define each node in a separate XML file and reference them to its following. Obviously this proposition is highly impractical.

```
<RobotNode name="TrafoToLeftLeg">
  <Transform>
    <matrix4x4>
      <row1 c4="0" c3="0" c2="1" c1="0"/>
      <row2 c4="146" c3="0" c2="0" c1="1"/>
      <row3 c4="0" c3="-1" c2="0" c1="0"/>
      <row4 c4="1" c3="0" c2="0" c1="0"/>
    </matrix4x4>
    <DH units="degree" theta="90" alpha="90" d="0" a="0"/>
  </Transform>
  <ChildFromRobot>
    <File importEEF="true">xmlfiles/LefLegSimox.xml</File>
  </ChildFromRobot>
</RobotNode>
```

Listing A.8: Importing a sub-assembly.

This same method was used to refer the other leg and the body to the root file. Also the body contains both arms and the head. The final result is a robot ready for the ArmarX environment and is shown in Figure A.8.

In addition, a general kinematic chain was added. This *RobotNodeSet* includes all joints. Since the use of the whole robot is rather common, including both particular and general kinematic chains is a recommended procedure.

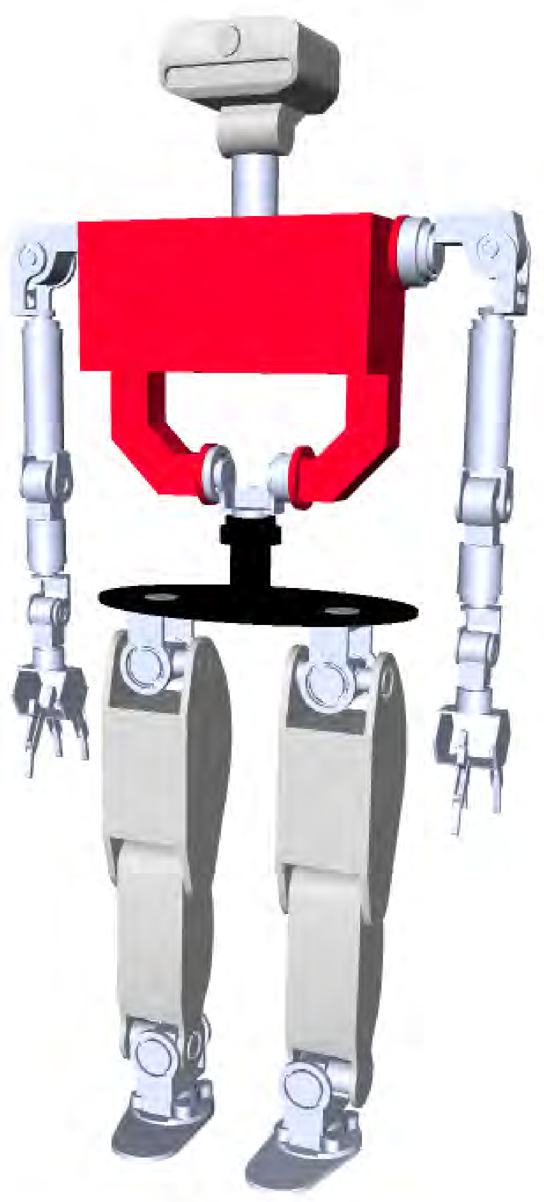


Figure A.8: TEO model completed.

B Implementing a Counter

This part of the appendix will initiate in the use of **ArmarX**, with basic guidelines to work with it and an overall description of what is been doing in every step. Also here we will learn how to use the **StateChart Editor** and the basics of the **ScenarioManager**.

To achieve this objectives, we will create a counter with ArmarX that will follow the same behaviour as Figure B.1

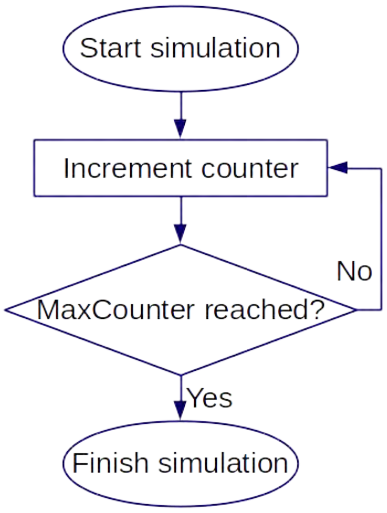
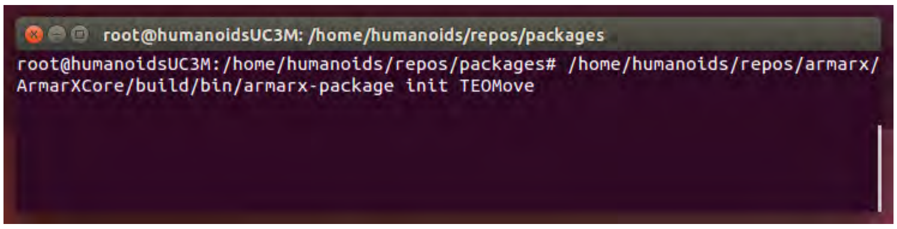


Figure B.1: Counter flowchart diagram.

B.1 Creating a Package

As was mentioned in 6.3, ArmarX wraps its projects in the so called *Packages*, and the framework comes with a tool to create them. You can start with:



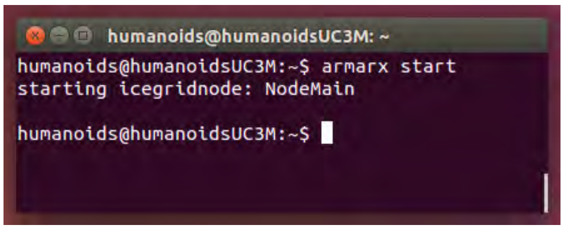
```
root@humanoidsUC3M: /home/humanoids/repos/packages
root@humanoidsUC3M:/home/humanoids/repos/packages# /home/humanoids/repos/armarx/
ArmarXCore/build/bin/armarx-package init TEOMove
```

Figure B.2: Creating our package for the project.

This command will create a package called **TEOMove** in your current location. It is necessary to invoke the tool from root. For the rest of this manual, we will work with the package called TEOMove we have just created [19].

B.2 Running ArmarX

Once we have our package, we can start using the framework itself: first we will deploy **Ice** first, if we do not, no component of ArmarX will run.

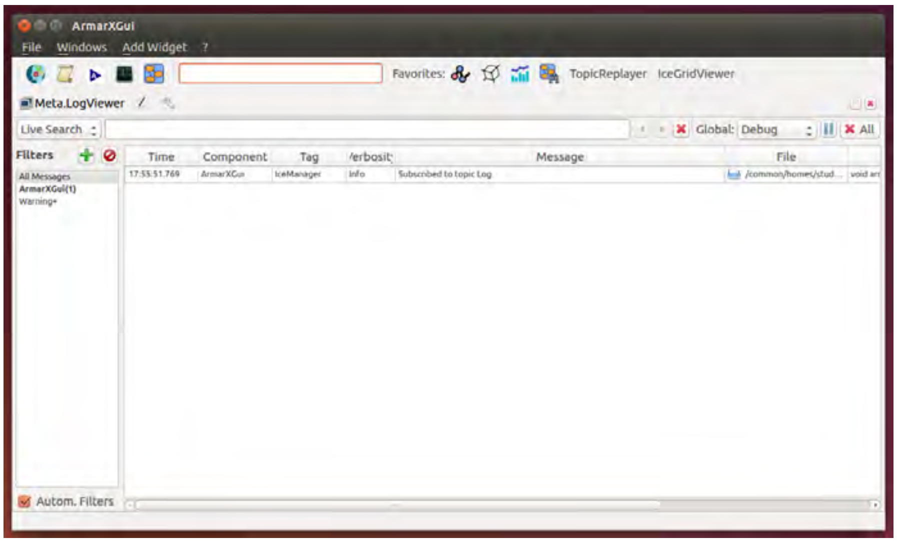


```
humanoids@humanoidsUC3M: ~
humanoids@humanoidsUC3M:~$ armarx start
starting icegridnode: NodeMain

humanoids@humanoidsUC3M:~$ 
```

Figure B.3: Deploying Ice.

Now ArmarX is running in the background and we can start over it the **GUI** by typing **armarx gui** in the console.



```
Time          Component  Tag         /erbosit  Message                   File
17:55:51.769  ArmarXGui  IceManager  Info      Subscribed to topic Log   /common/homes/stud...  void ar
```

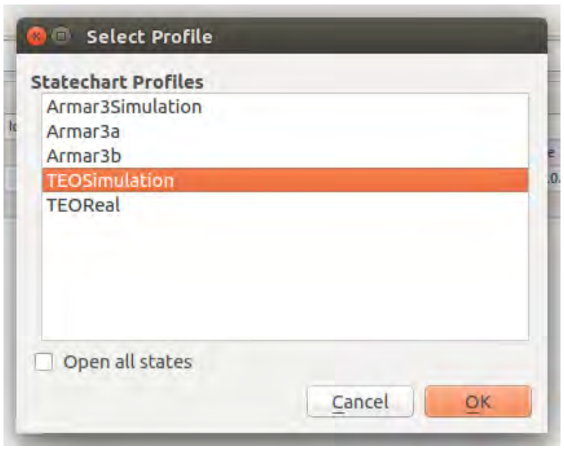
Figure B.4: ArmarXGui in our computer, ready to start working with it.

We must compile the package after is made, to do so, execute the following instructions in the terminal:

```
cd TEOMove/build
cmake ..
make
```

Listing B.1: Compiling a package

First, we will open the **StateChart Editor**, by *Add Widget >>StateCharts >>StateChart Editor*. Once the widget opens, it will ask us about a which robot profile to use. Information about robot profiles can be found in Section 6.4 and how to configure them in Annex **INCLUIR ANEXO PROFILES**. For this task we will choose *TEOSimulation*.


Select Profile

Statechart Profiles

Armar3Simulation
Armar3a
Armar3b
TEOSimulation
TEOReal

Open all states

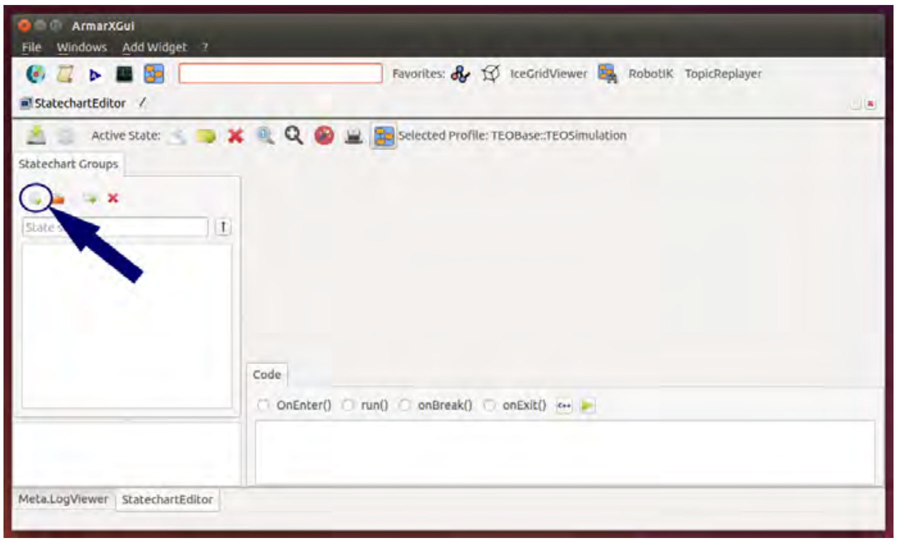
Cancel OK

Figure B.5: Profile selection.

B.3 Working with the StateChart Editor

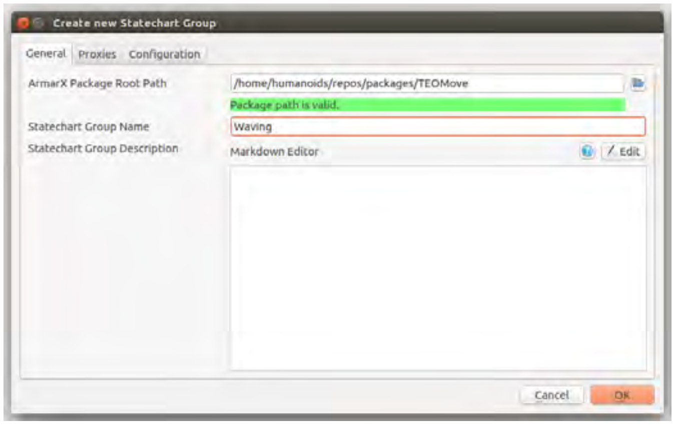
For this section we want to create a StateChart that counters up to 4, following the flowchart of Figure B.1.

Now, we will create a group to work with by clicking in *New StateChart Group* (See Figure B.6). Once the new window shows up, we will call it **Waving**. It is also necessary to include a package root directory, that will be **TEOMove**. This package was previously created in Section B.1.


Selected Profile: TEOBase::TEOSimulation

ArmarXGui
File Windows Add Widget ?
Favorites: IceGridViewer RobotIK TopicReplayer
StatechartEditor
Active State:
Statechart Groups
Code
OnEnter() run() onBreak() onExit()
Meta.LogViewer StatechartEditor

Figure B.6: Creating a StateChart Group.


Create new Statechart Group

General | Proxies | Configuration

ArmarX Package Root Path: /home/humanoids/repos/packages/TEOMove

Package path is valid.

Statechart Group Name: Waving

Statechart Group Description: Markdown Editor

Figure B.7: StateChart new group creation.

The StateChart group is defined, now it is time to begin creating the logic and the processes the robot will follow. The StateChart sintaxis was already discussed in Section 4.4. Now we will focus on creating the StateChart itself.

First, we must create a State (see Figure B.8) to start working. This one will be our **MainState**. We should also create another state that will be the **CounterState**.

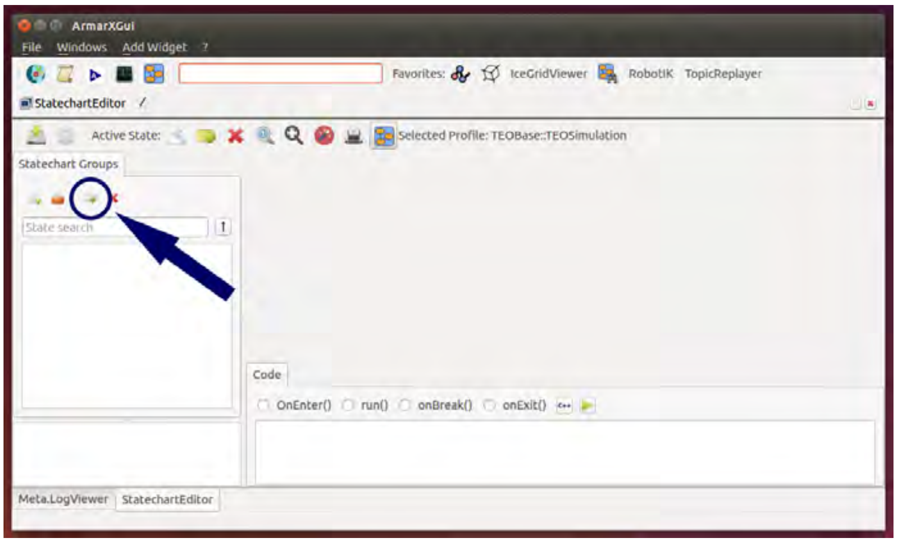
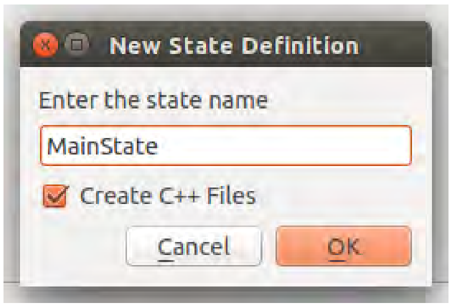


Figure B.8: Defining a new state.

A dialog will show up when creating a new state, add the name to the state and do not forget to check the *Create C++ files* (Figure B.9).



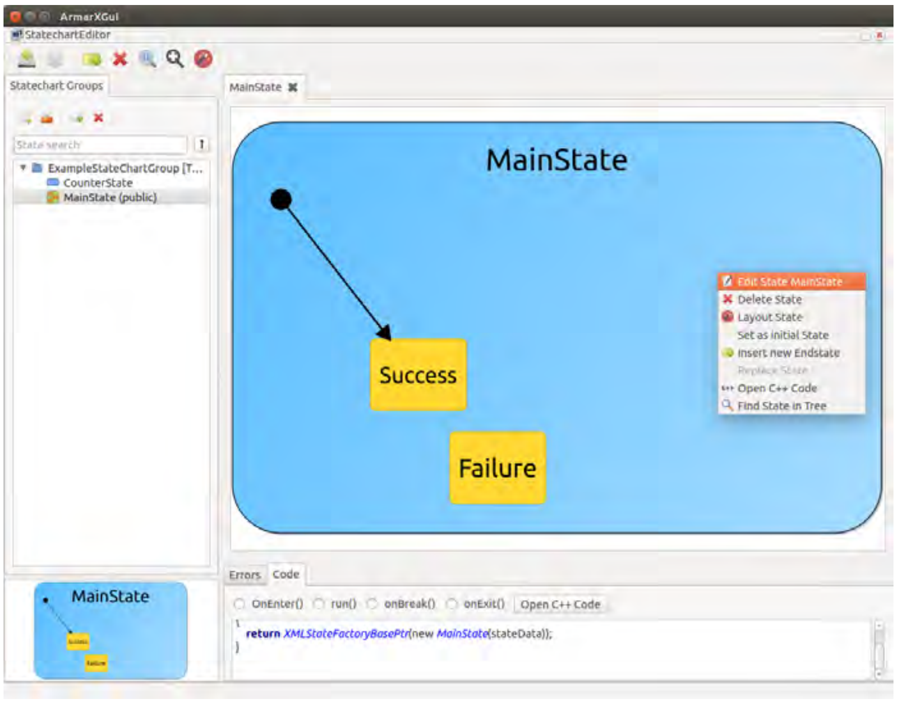
Enter the state name: MainState

[✓] Create C++ Files

Figure B.9: Newstate definition.

Also we need to make MainState public, by right clicking it and changing it to public.

In Figure B.10 we see a default creation of a state, with two **EndStates**. By default they are created with *Success*, as initial; and **Failure.**

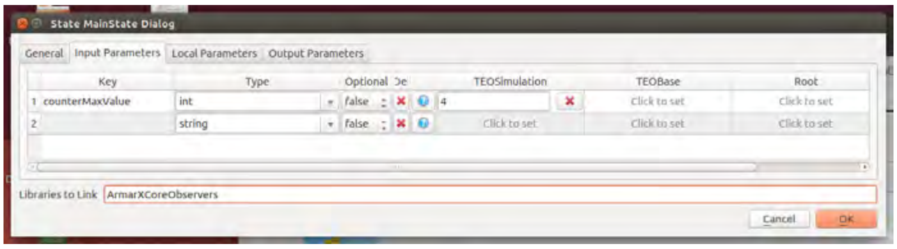


```
{
  return XMLStateFactoryBasePtr(new MainState(stateData));
}
```

Figure B.10: Generation of a new state.

Now, we are going to change start adding parameters to the MainState, by right-clicking it and then *Edit MainState*. A new window dialog will show up and we will define an input parameter called **counterMaxValue** (Figure B.11).

This variable will be an integer and we define it in the profile TEOSimulation with a value of **4**. Why we add the value to this profile is explained in Section 6.4.


State MainState Dialog

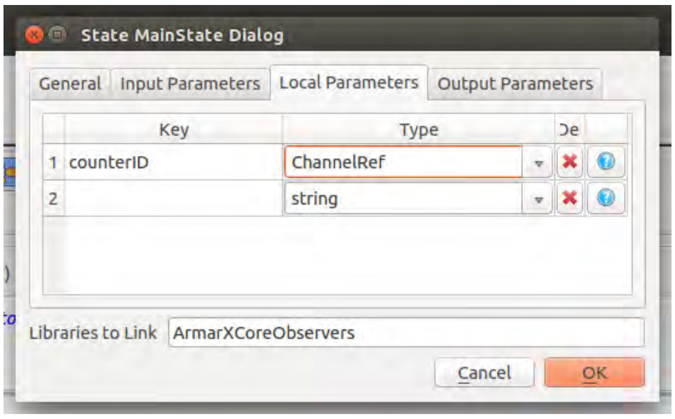
General | Input Parameters | Local Parameters | Output Parameters

	Key	Type	Optional	De		TEOSimulation		TEOBase	Root
1	counterMaxValue	int	false	✖	●	4	✖	Click to set	Click to set
2		string	false	✖	●	Click to set		Click to set	Click to set

Libraries to Link: ArmarXCoreObservers

Figure B.11: MainState input parameters.

We will repeat the similar process but in the local parameters tab. Adding a *ChannelRef* called **counterId**. The type *ChannelRef* will allow the observers to access its value and show it in the **LogViewer** widget.


State MainState Dialog

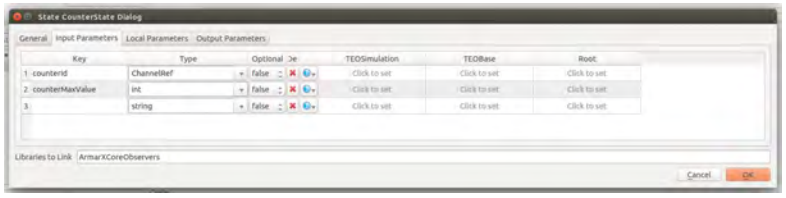
General | Input Parameters | Local Parameters | Output Parameters

	Key	Type	De	
1	counterID	ChannelRef	✖	●
2		string	✖	●

Libraries to Link: ArmarXCoreObservers

Figure B.12: MainState local parameters.

We must add parameters to the CounterState also, so we will edit the state and add as input parameters counterId and counterMaxValue so they can be used in this state too (Figure B.13).


State CounterState Dialog

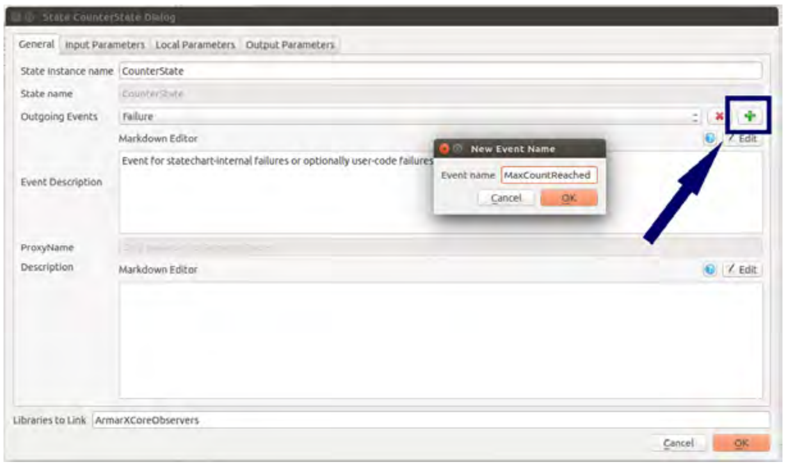
General | Input Parameters | Local Parameters | Output Parameters

	Key	Type	Optional	De		TEOSimulation	TEOBase	Root
1	counterId	ChannelRef	false	✖	●	Click to set	Click to set	Click to set
2	counterMaxValue	int	false	✖	●	Click to set	Click to set	Click to set
3		string	false	✖	●	Click to set	Click to set	Click to set

Libraries to Link: ArmarXCoreObservers

Figure B.13: CounterState input parameters.

Also we are going to edit the *Outgoing Events* of CounterState, in the General tab. Add two new outgoing events called **MaxCountReached** and **MaxCountNotReached**, like is shown in Figure B.14:



State CounterState Dialog

General | Input Parameters | Local Parameters | Output Parameters

State instance name: CounterState

State name: CounterState

Outgoing Events: Failure

Event Description (Markdown Editor): Event for statechart-internal failures or optionally user-code failures

New Event Name

Event name: MaxCountReached

Cancel | OK

ProxyName

Description (Markdown Editor)

Libraries to Link: ArmarXCoreObservers

Cancel | OK

Figure B.14: Outgoing events of CounterState.

Now we will drag the CounterState inside the MainState so now our StateChart looks like Figure B.15:

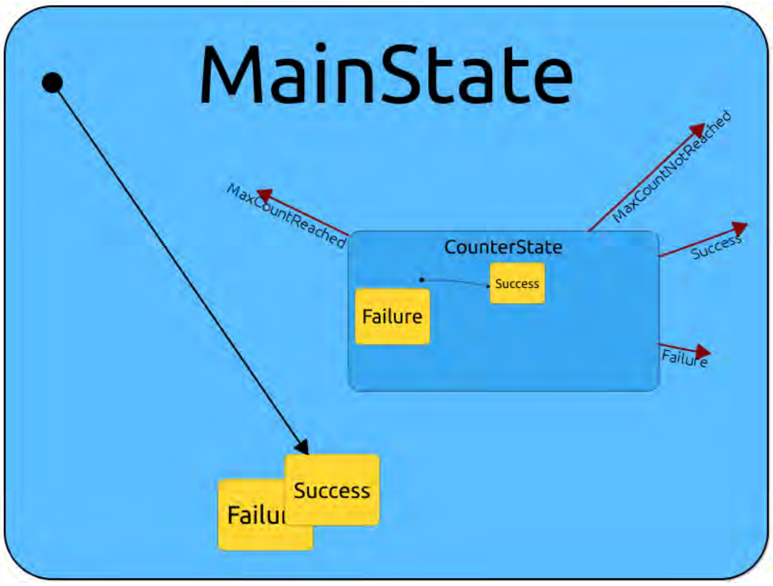


Figure B.15: CounterState dragged inside MainState.

We can see that we have another outgoing event in CounterState: *Success*. That is because we have its EndState in CounterState, we will delete both success and failure. Note that outgoing failure transition didn't disappear because it **needs to be present for all states**.

So now we can make our StateChart behave as the flowchart of Figure B.1. For that purpose the following steps were made:

- Set CounterState as *Initial State* by right-click on it and selecting the according option.
- Drag the end of **MaxCountReached** transition to the **Success** endstate.

- Drag the end of **MaxCountNotReached** transition back to **CounterState**.
- Drag the end of **Failure** transition to the **Failure** endstate. This will have no impact on the advance of our simulation, but we always have to include it for stability and error management.

The final StateChart should look like Figure B.16. This tidy result is achieved by clicking in the *Layout state* icon at the top of the StateChart Editor window.

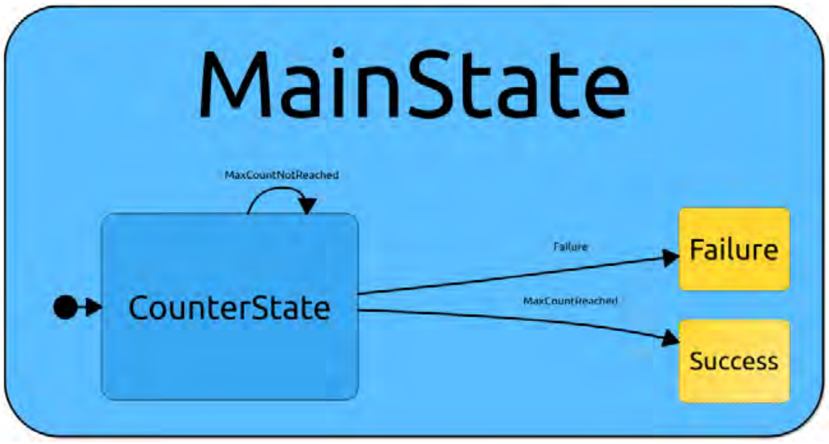


Figure B.16: StateChart final structure.

Most of the work with the StateChart is made, we need to edit the transitions in order to select the parameters they pass. To do so, right-click in a transition and select *Edit Transition* like in Figure B.17.

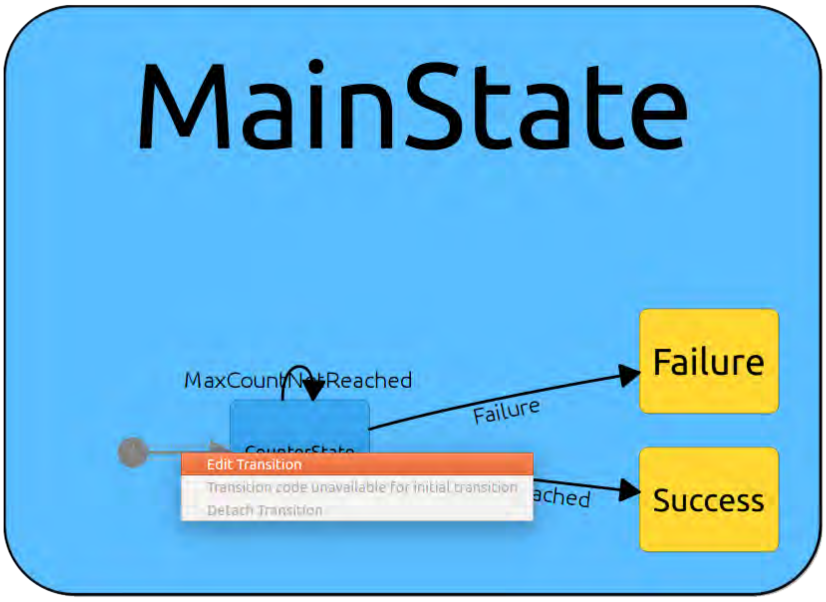
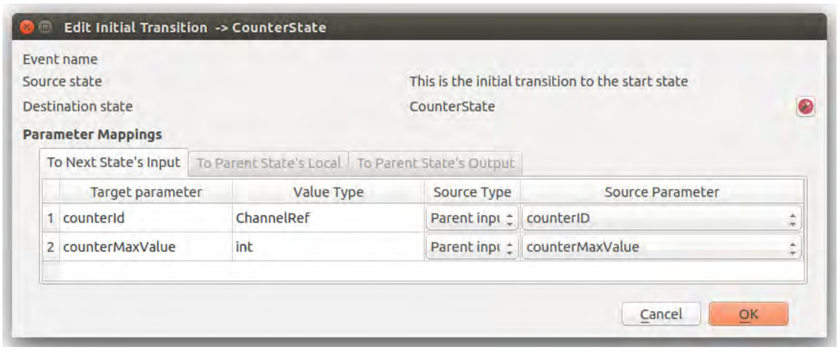


Figure B.17: StateChart final structure.

For the initial transition to CounterState, we will select the **Source Parameter** like in Figure B.18:



Edit Initial Transition -> CounterState

Event name	
Source state	This is the initial transition to the start state
Destination state	CounterState

Parameter Mappings

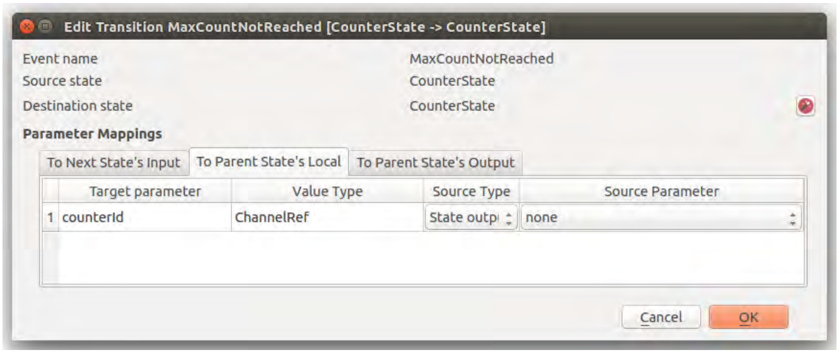
To Next State's Input | To Parent State's Local | To Parent State's Output

	Target parameter	Value Type	Source Type	Source Parameter
1	counterId	ChannelRef	Parent inpu	counterID
2	counterMaxValue	int	Parent inpu	counterMaxValue

Figure B.18: Initial transition parameters.

And finally, for the **MaxCountNotReached** transition, we need to repeat the process until the tab *To Next State's Input* looks like Figure B.19.

Do not forget to save before passing to the next stage.



Edit Transition MaxCountNotReached [CounterState -> CounterState]

Event name	MaxCountNotReached
Source state	CounterState
Destination state	CounterState

Parameter Mappings

To Next State's Input | To Parent State's Local | To Parent State's Output

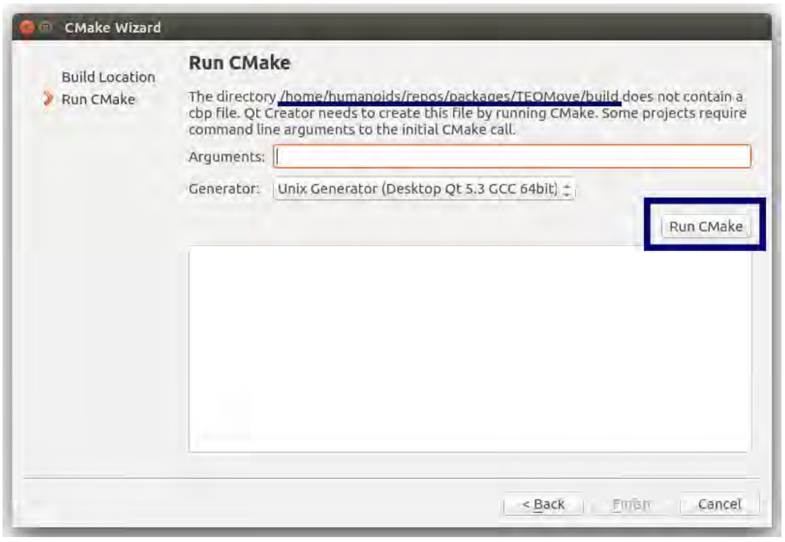
	Target parameter	Value Type	Source Type	Source Parameter
1	counterId	ChannelRef	State outp	none

Figure B.19: MaxCountNotReached transition parameters.

B.4 Editing the Source Code

Doing the StateChart correctly save us a big amount of work and time, but we still need to write a bit of code to finish. **QTCreator** is the IDE (Integrated Development Environment). It was used for

For opening the project we should go to *File >>Open File or Project* and select the *CMakeLists.txt* found in your package, in this case, TEOMove. It will ask for a build folder, we will select the build directory of the package **TEOMove Package**



Run CMake

The directory /home/humanoids/repos/packages/TEOMove/build does not contain a cbp file. Qt Creator needs to create this file by running CMake. Some projects require command line arguments to the initial CMake call.

Arguments:

Generator: Unix Generator (Desktop Qt 5.3 GCC 64bit)

Run CMake

Figure B.20: QTCreator project set-up.

We will first edit **MainState** files, both MainState.cpp and MainState.h. They are located in: *TEOMove >>source >>statecharts >>waving*

- Uncomment all void functions: *onEnter()*, *run()*, *onBreak()* and *onExit()* in the .cpp file and their declaration in the header file.

- Add the code to the *onEnter()* shown in Listing B.2 (lines 10 to 13).

```
 1 #include "MainState.h"
 2
 3 using namespace armarx;
 4 using namespace Waving;
 5
 6 MainState::SubClassRegistry MainState::Registry(MainState::
       GetName(), &MainState::CreateInstance);
 7
 8 void MainState::onEnter()
 9 {
10     ARMARX_LOG << "Enter MainState";
11
12     ChannelRefPtr counterId = ChannelRefPtr::dynamicCast(
           getContext()->systemObserverPrx->startCounter(0, "
           counterId"));
13     local.setcounterId(counterId);
14 }
15
16 void MainState::run(){}
17
18 void MainState::onBreak(){}
19
```

```cpp
void MainState::onExit()
{
  ARMARX_LOG << "Exit MainState, count reached";
}

XMLStateFactoryBasePtr MainState::CreateInstance(XMLStateConstructorParams stateData)
{
    return XMLStateFactoryBasePtr(new MainState(stateData));
}
```

Listing B.2: MainState.cpp

For both CounterState.h and CounterState.cpp we will follow a similar procedure:

- Uncomment all void functions: *onEnter()*, *run()*, *onBreak()* and *onExit()* in the .cpp file and their declaration in the header file.
- Add the code to the *onEnter()* shown in Listing B.3 (lines 10 to 21).

```cpp
 1 #include "CounterState.h"
 2 
 3 using namespace armarx;
 4 using namespace Waving;
 5 
 6 CounterState::SubClassRegistry CounterState::Registry(CounterState::GetName(), &CounterState::CreateInstance);
 7 
 8 void CounterState::onEnter()
 9 {
10     ARMARX_LOG << "Enter CounterState";
11     ChannelRefPtr counterId = in.getcounterId();
12     int maxValue = in.getcounterMaxValue();
13     getContext()->systemObserverPrx->incrementCounter(counterId);
14     int counterValue = counterId->getDataField("value")->getInt();
15     if (counterValue >= maxValue)
16     {
17         emitMaxCountReached();
18     }
19     else
20     {
21         emitMaxCountNotReached();
22     }
23 }
```

```

void CounterState::run(){}

void CounterState::onBreak(){}

void CounterState::onExit(){}

XMLStateFactoryBasePtr CounterState::CreateInstance(
    XMLStateConstructorParams stateData)
{
    return XMLStateFactoryBasePtr(new CounterState(stateData));
}
```

Listing B.3: CounterState.cpp

Now we must build the project, both from QT or the terminal, like in Listing B.1, can be used and work the same.

B.5 Working with the ScenarioManager

The final part is to run the simulation, for that, we will use the **Scenario Manager** widget. We can open it by *Add Widget >>Meta >>ScenarioManager.*

First, we have to register the package **TEOMove** by clicking in the configure button (the small wrench in the top of the widget). The SettingsView window will appear, click in open, now write the **exact name** of the package. The build directory will be automatically added if the name is correctly written. In Figure B.21 is detailed step by step.

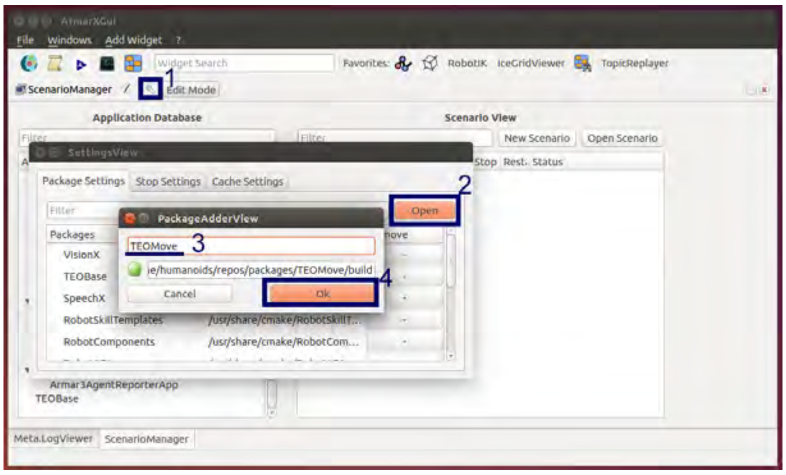


Figure B.21: Registering a new package in ScenarioManager.

Now, we add our Scenario, click in *New Scenario*, we will call it **TEOWave** and we will select the package

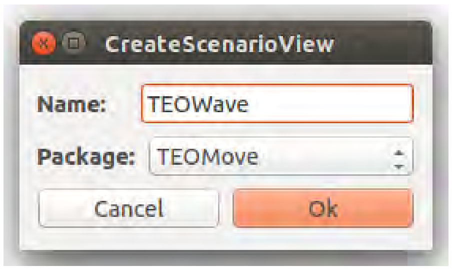


Figure B.22: Creating a new scenario.

We will add the StateChart group we created earlier by dragging it to the scenario. It is easier if we open both ScenarioManager and StateChart Editor and keep both tabs together like in Figure B.23.

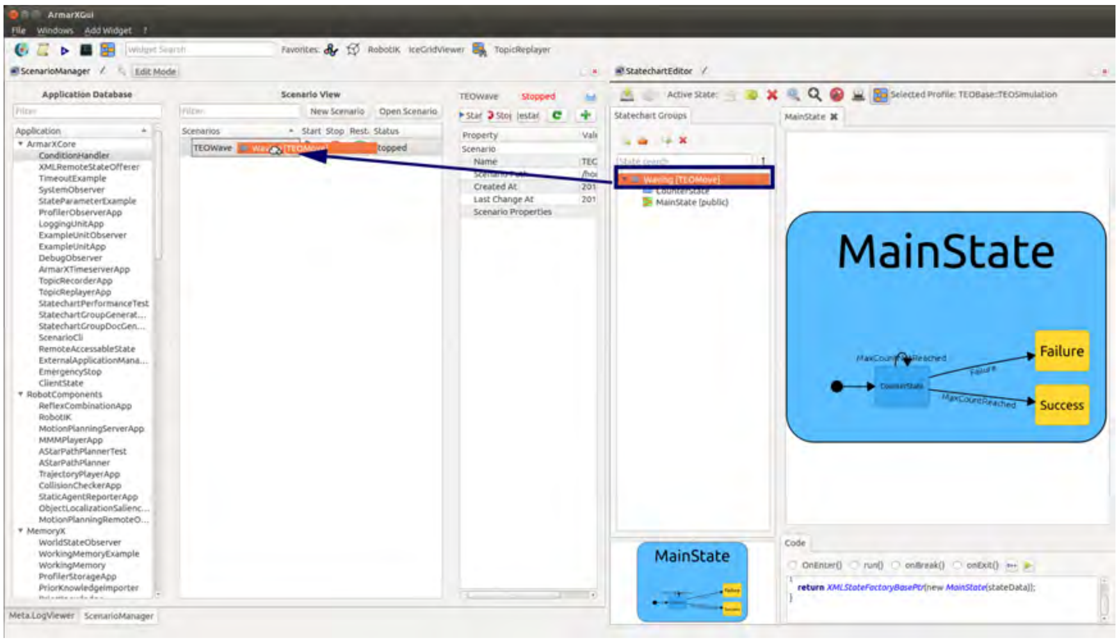
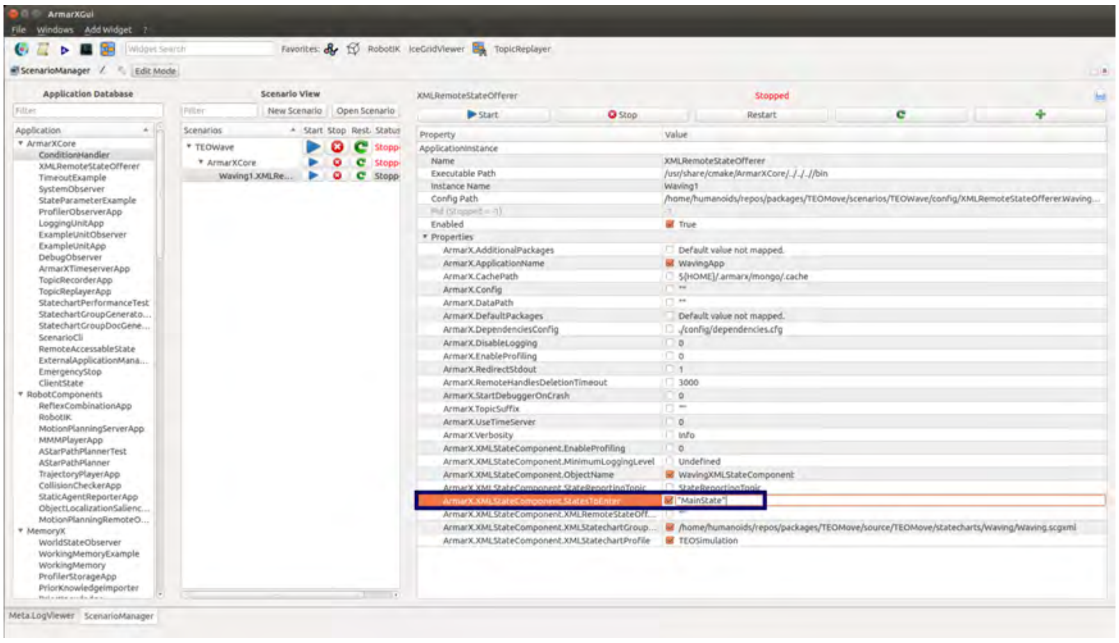


Figure B.23: Adding the StateChart group to the scenario.

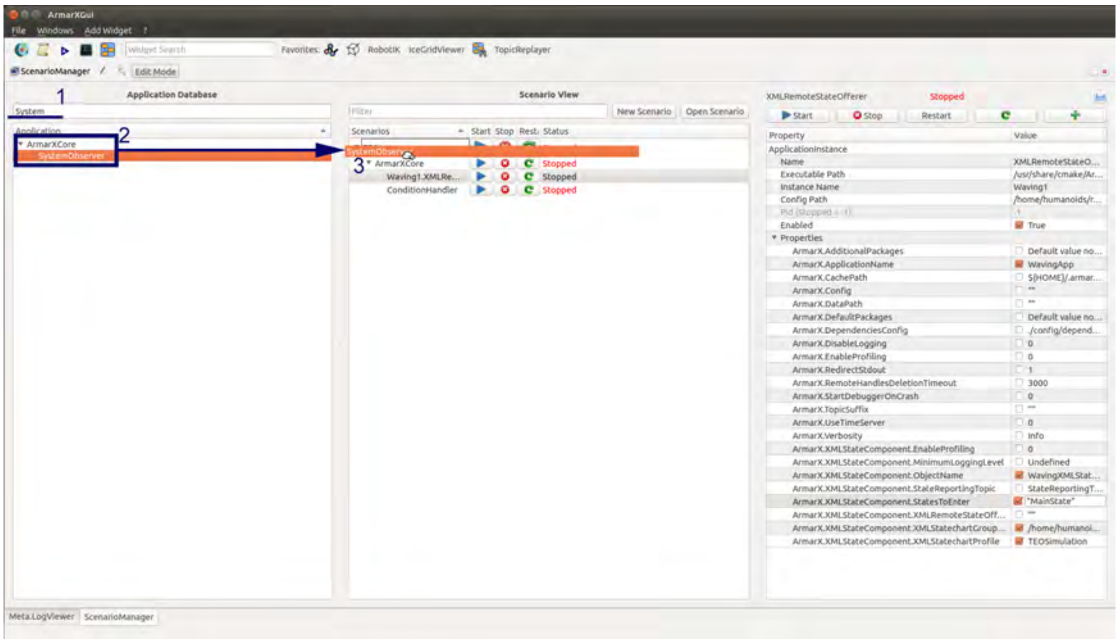
Now we have to choose the state to begin with, in this case is MainState, so we change the parameter *ArmarX.XMLStateComponent.StatesToEnter* to **MainState**, as shown in Figure B.24



Property	Value
ApplicationInstance	
Name	XMLRemoteStateOfferer
Executable Path	/usr/share/cmake/ArmarXCore/../../..//bin
Instance Name	Waving1
Config Path	/home/humanoids/repos/packages/TEOMove/scenarios/TEOWave/config/XMLRemoteStateOfferer.Waving...
Pid (Stopped = -1)	-1
Enabled	☑ True
Properties	
ArmarX.AdditionalPackages	☐ Default value not mapped.
ArmarX.ApplicationName	☑ WavingApp
ArmarX.CachePath	☐ ${HOME}/.armarx/mongo/.cache
ArmarX.Config	☐ ""
ArmarX.DataPath	☐ ""
ArmarX.DefaultPackages	☐ Default value not mapped.
ArmarX.DependenciesConfig	☐ ./config/dependencies.cfg
ArmarX.DisableLogging	☐ 0
ArmarX.EnableProfiling	☐ 0
ArmarX.RedirectStdout	☐ 1
ArmarX.RemoteHandlesDeletionTimeout	☐ 3000
ArmarX.StartDebuggerOnCrash	☐ 0
ArmarX.TopicSuffix	☐ ""
ArmarX.UseTimeServer	☐ 0
ArmarX.Verbosity	☐ Info
ArmarX.XMLStateComponent.EnableProfiling	☐ 0
ArmarX.XMLStateComponent.MinimumLoggingLevel	☐ Undefined
ArmarX.XMLStateComponent.ObjectName	☑ WavingXMLStateComponent
ArmarX.XMLStateComponent.StateReportingTopic	☐ StateReportingTopic
ArmarX.XMLStateComponent.StatesToEnter	☑ "MainState"
ArmarX.XMLStateComponent.XMLRemoteStateOff...	☐ ""
ArmarX.XMLStateComponent.XMLStatechartGroup...	☑ /home/humanoids/repos/packages/TEOMove/source/TEOMove/statecharts/Waving/Waving.scgxml
ArmarX.XMLStateComponent.XMLStatechartProfile	☑ TEOSimulation

Figure B.24: Changing StateChart group parameters.

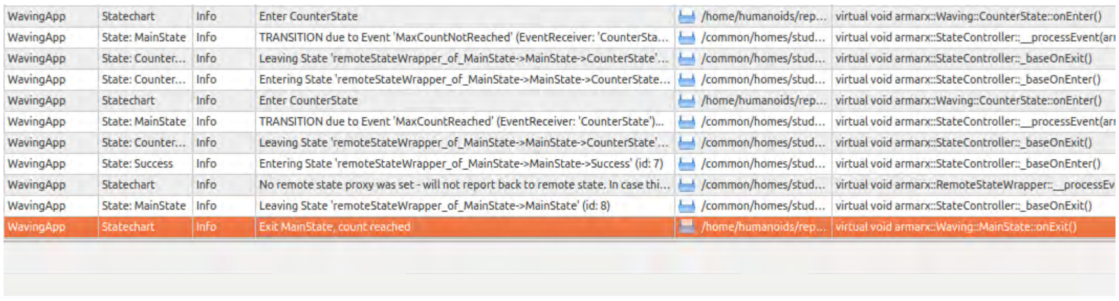
Is necessary to add a couple more applications to the scenario in order to run, those are: *ConditionHandler* and *SystemObserver*. They can be added by searching for their names and dragging them into the scenario like we did with the Waving group, see Figure B.25.



Property	Value
ApplicationInstance	
Name	XMLRemoteStateO...
Executable Path	/usr/share/cmake/Ar...
Instance Name	Waving1
Config Path	/home/humanoids/r...
Pid (Stopped = -1)	-1
Enabled	☑ True
Properties	
ArmarX.AdditionalPackages	☐ Default value no...
ArmarX.ApplicationName	☑ WavingApp
ArmarX.CachePath	☐ ${HOME}/.armar...
ArmarX.Config	☐ ""
ArmarX.DataPath	☐ ""
ArmarX.DefaultPackages	☐ Default value no...
ArmarX.DependenciesConfig	☐ ./config/depend...
ArmarX.DisableLogging	☐ 0
ArmarX.EnableProfiling	☐ 0
ArmarX.RedirectStdout	☐ 1
ArmarX.RemoteHandlesDeletionTimeout	☐ 3000
ArmarX.StartDebuggerOnCrash	☐ 0
ArmarX.TopicSuffix	☐ ""
ArmarX.UseTimeServer	☐ 0
ArmarX.Verbosity	☐ Info
ArmarX.XMLStateComponent.EnableProfiling	☐ 0
ArmarX.XMLStateComponent.MinimumLoggingLevel	☐ Undefined
ArmarX.XMLStateComponent.ObjectName	☑ WavingXMLStat...
ArmarX.XMLStateComponent.StateReportingTopic	☐ StateReportingT...
ArmarX.XMLStateComponent.StatesToEnter	☑ "MainState"
ArmarX.XMLStateComponent.XMLRemoteStateOff...	☐ ""
ArmarX.XMLStateComponent.XMLStatechartGroup...	☑ /home/humanoi...
ArmarX.XMLStateComponent.XMLStatechartProfile	☑ TEOSimulation

Figure B.25: Changing StateChart group parameters.

The first simulation is done, to run it, click on the play button of the Scenario. To review the results we go to the **LogViewer** and check its output, if out log looks like Figure B.26.



WavingApp	Statechart	Info	Enter CounterState	/home/humanoids/rep...	virtual void armarx::Waving::CounterState::onEnter()
WavingApp	State: MainState	Info	TRANSITION due to Event 'MaxCountNotReached' (EventReceiver: 'CounterSta...	/common/homes/stud...	virtual void armarx::StateController::__processEvent(ar
WavingApp	State: Counter...	Info	Leaving State 'remoteStateWrapper_of_MainState->MainState->CounterState'...	/common/homes/stud...	virtual void armarx::StateController::_baseOnExit()
WavingApp	State: Counter...	Info	Entering State 'remoteStateWrapper_of_MainState->MainState->CounterState...	/common/homes/stud...	virtual void armarx::StateController::_baseOnEnter()
WavingApp	Statechart	Info	Enter CounterState	/home/humanoids/rep...	virtual void armarx::Waving::CounterState::onEnter()
WavingApp	State: MainState	Info	TRANSITION due to Event 'MaxCountReached' (EventReceiver: 'CounterState')...	/common/homes/stud...	virtual void armarx::StateController::__processEvent(ar
WavingApp	State: Counter...	Info	Leaving State 'remoteStateWrapper_of_MainState->MainState->CounterState'...	/common/homes/stud...	virtual void armarx::StateController::_baseOnExit()
WavingApp	State: Success	Info	Entering State 'remoteStateWrapper_of_MainState->MainState->Success' (id: 7)	/common/homes/stud...	virtual void armarx::StateController::_baseOnEnter()
WavingApp	Statechart	Info	No remote state proxy was set - will not report back to remote state. In case thi...	/common/homes/stud...	virtual void armarx::RemoteStateWrapper::__processEv
WavingApp	State: MainState	Info	Leaving State 'remoteStateWrapper_of_MainState->MainState' (id: 8)	/common/homes/stud...	virtual void armarx::StateController::_baseOnExit()
WavingApp	Statechart	Info	Exit MainState, count reached	/home/humanoids/rep...	virtual void armarx::Waving::MainState::onExit()

Figure B.26: The first program in ArmarX was successful.

C Introducing a New Robot in ArmarX

After completing our first successful application in ArmarX, we want to keep working and introduce **TEO** to the environment. This is the main goal of this appendix. To do so, we will expand the StateChart that we created in the Appendix B. We will add the necessary items and parameters to handle TEO's movement. Later we will begin working with some of the 3D observers that ArmarX provide, to visualize the kinematic simulation of the real robot [19].

Some of the procedures were earlier explained in the Appendix B, so the already mentioned steps will not be as detailed as before. Still, the new parts that appear will be explained.

The objective is to make TEO follow the flowchart shown in Figure C.1.

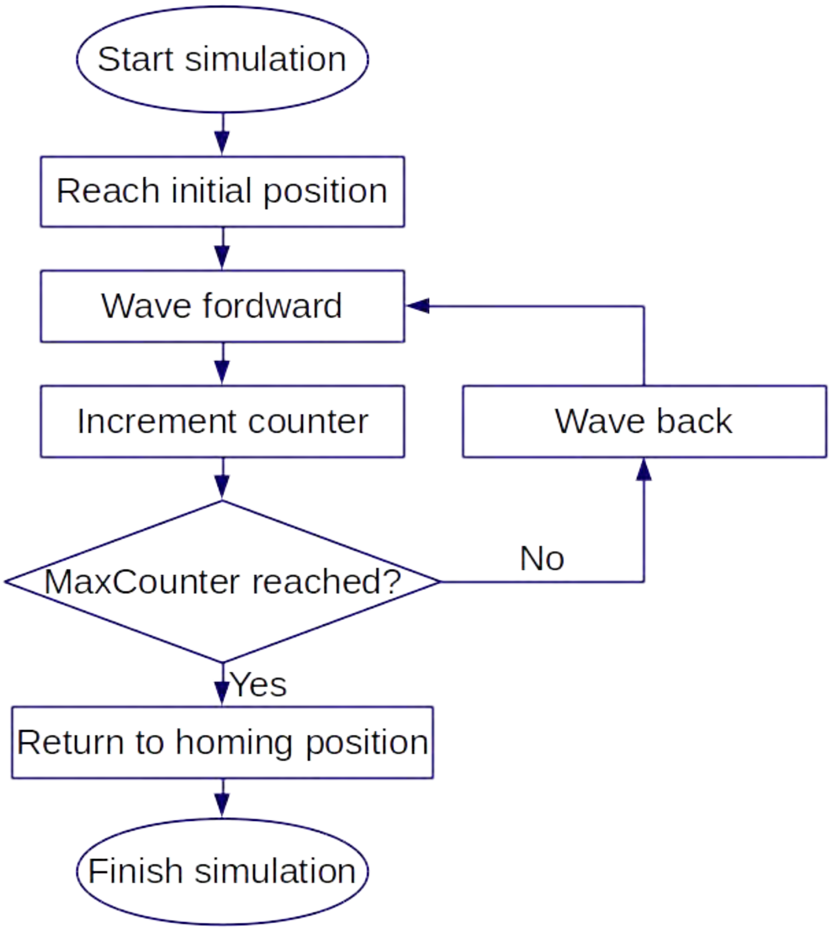


Figure C.1: Final simulation process.

C.1 Expanding the StateChart

In case ArmarX or the StateChart Editor were closed, we can load a StateChart Group. To open it again, we click in *Open StateChart Group*, see Figure C.2. We are looking for a **.scgxml** file (StateChartGroup XML), and it will be located in **TEOMove/source/TEOMove/statechats/Waving**.

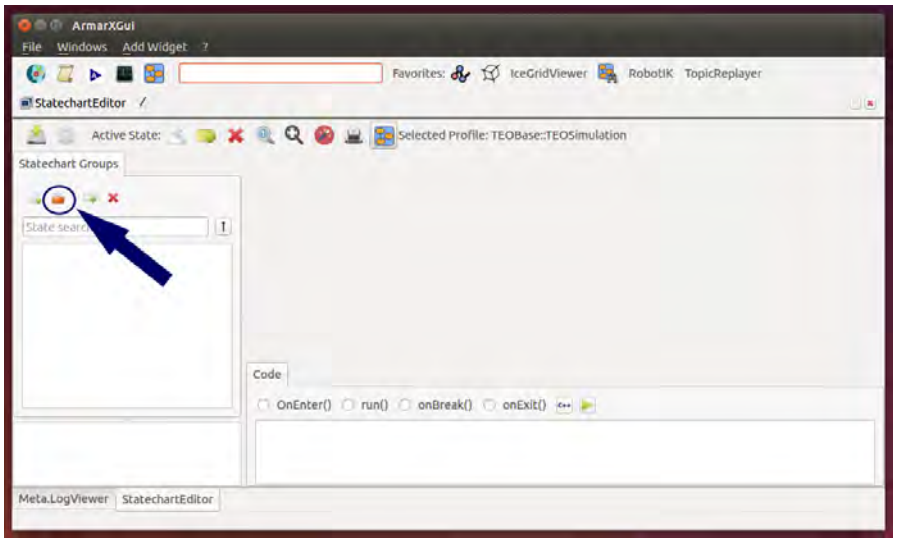

ArmarXGui
File Windows Add Widget ?
Favorites: IceGridViewer RobotIK TopicReplayer
StatechartEditor
Active State:
Selected Profile: TEOBase::TEOSimulation
Statechart Groups
State search
Code
OnEnter() run() onBreak() onExit()
Meta.LogViewer StatechartEditor

Figure C.2: Loading a StateChart group.

Now we will add two new states, by clicking the button *New State Definition.* One should be named **MoveJoints** and the other, **SetJoints**. Do not forget to check *Create C++ Files.* We can delete the EndStates within this new States (Failure and Success). Now, our StateChart Group will look like Figure C.3

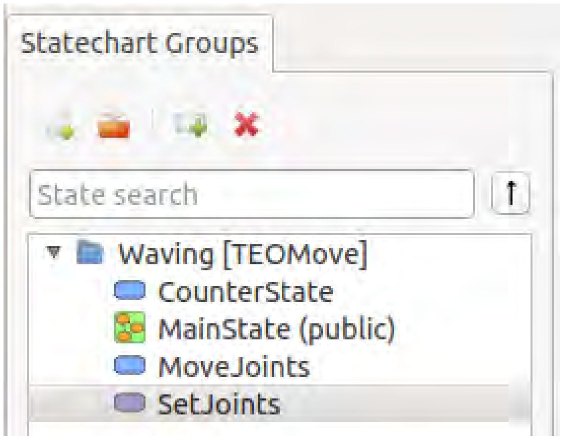
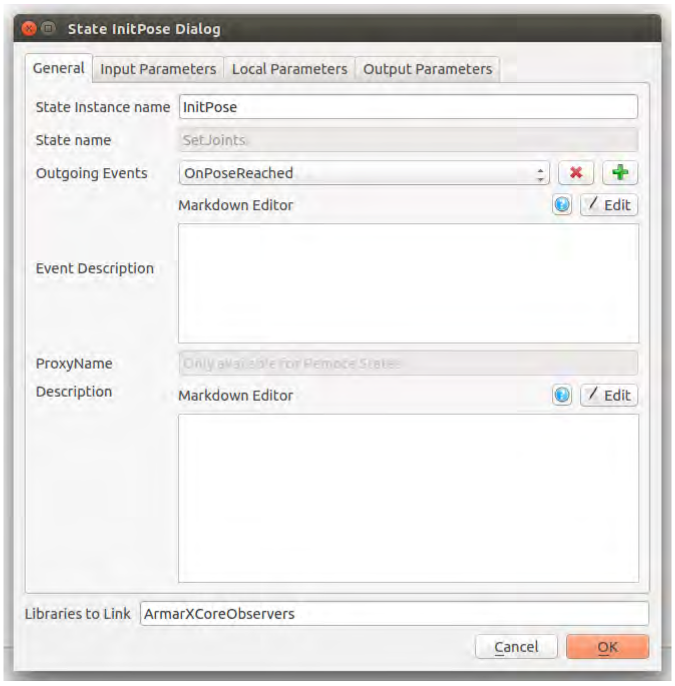


Figure C.3: MoveJoints and SetJoints created.

We will add **two MoveJoints** and **two SetJoints** by dragging them inside **MainState**. Now they should be renamed, one of the MoveJoints to **WaveFwd**, and the other one to **WaveBack**. Also SetJoints will be named **InitPose** and **FinalPose**. To change the name of a state, right-click on the State $\Longrightarrow$ Edit State

$\Longrightarrow$ General tab $\Longrightarrow$ State Instance Name.

Once we are editing every State, we can also add the outgoing events. For both (SetJoints class and MoveJoints class) will be **OnPoseReached**. It does not matter whether we edit WaveBack or WaveFwd, as they are both instances of the same class MoveJoint. Figure C.4 shows, for example, InitPose with OnPoseReached already declared.



State Instance name: InitPose
State name: SetJoints
Outgoing Events: OnPoseReached
Event Description:
ProxyName: Only available for Remote States
Description:
Libraries to Link: ArmarXCoreObservers

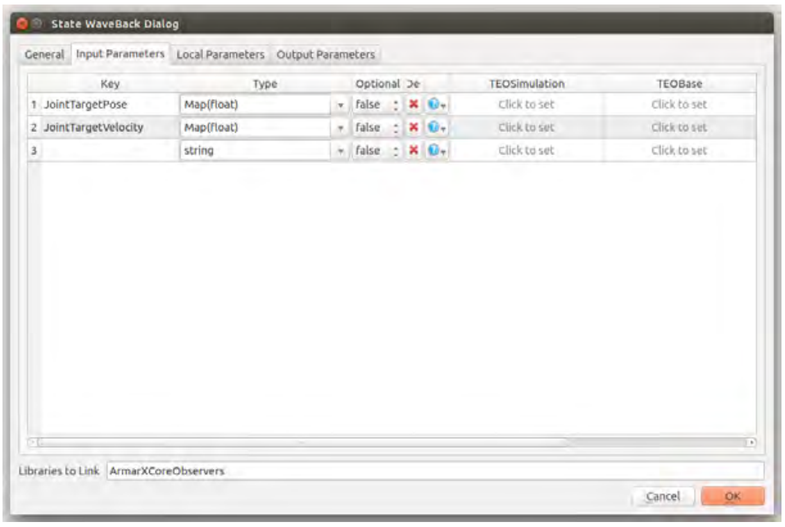
Figure C.4: MoveJoints and SetJoints created.

Now it is time to add the parameters for each instance. Like editing the outgoing events, it does not matter if we change WaveBack or WaveFwd. These changes will be applied in both. The variables shown in Table C.1 were added.

State	**Key**	**Type**
SetJoints	JointTargetPose	Map(Float)
MoveJoints	JointTargetPose	Map(Float)
MoveJoints	JointTargetVelocity	Map(Float)

Table C.1: Input parameters for each instance

JointTargetPose will be the position each state will try to achieve and **JointTargetVelocity** the velocity to which the joint will move. In Figure C.5 we can see how WaveBack input parameters should be added.


State WaveBack Dialog

General | Input Parameters | Local Parameters | Output Parameters

	Key	Type	Optional	De	TEOSimulation	TEOBase
1	JointTargetPose	Map(float)	false		Click to set	Click to set
2	JointTargetVelocity	Map(float)	false		Click to set	Click to set
3		string	false		Click to set	Click to set

Libraries to Link: ArmarXCoreObservers

Figure C.5: WaveBack input parameters.

Next step is to arrange the StateChart in order to follow the flowchart of Figure C.1. Then, our *MainState* will look like Figure C.6.

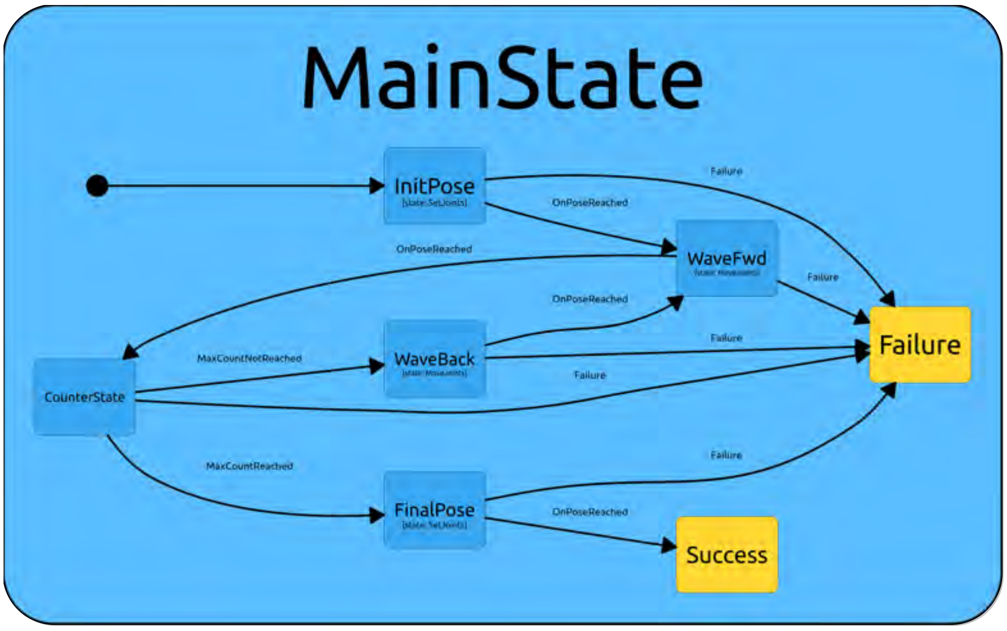


Figure C.6: Final MainState StateChart.

C.2 Managing the Parameters

We can see that no values were added in the parameters of the previous States. We want some defined values for the joint's positions. They go in the input parameters of *MainState*. The default values will be the angle value in radians of the joints in the final position. Also the angular velocity will be added.

In the input parameters of *MainState*, we will add the values shown in Table C.2. Here we will create the variables with particular name and value, while in the other states (*SetJoints* and *MoveJoints*) only the type was defined. So now we must match the type so we can pass them to the States we created earlier.

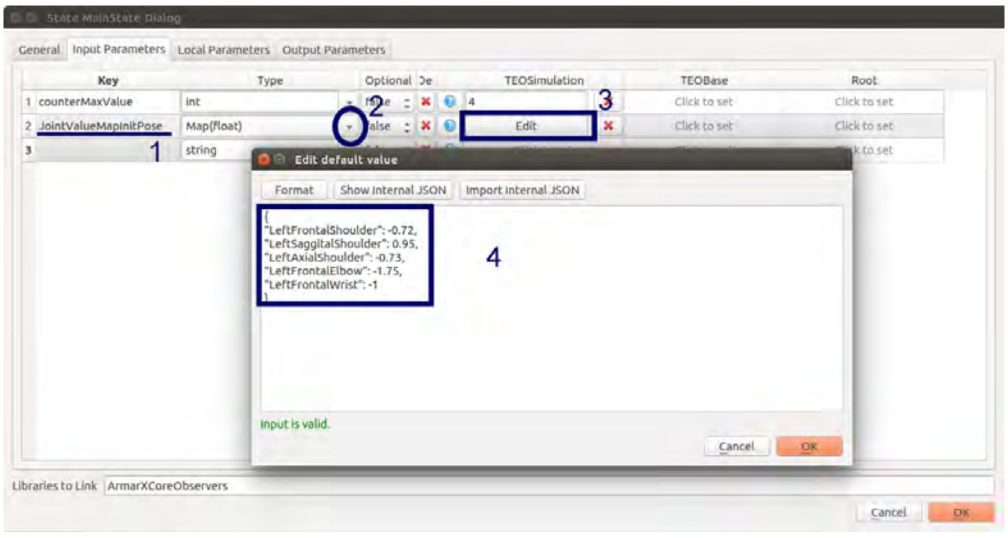
The type *Map(float)* admits plain text with the **JSON** (JavaScript Object Notation) syntax. The values we add can be fount in Table C.2.

State	**Type**	**Value**
JointValueMapInitPose	Map(Float)	{"LeftFrontalShoulder": -0.72, "LeftSaggitalShoulder": 0.3, "LeftAxialShoulder": -0.73, "LeftFrontalElbow": -1}
JointValueMapFinalPose	Map(Float)	{"LeftFrontalShoulder": 0, "LeftSaggitalShoulder": 0, "LeftAxialShoulder": 0, "LeftFrontalElbow": 0}
JointValueMapWaveBack	Map(Float)	{"LeftFrontalElbow": -1}
JointValueMapWaveFwd	Map(Float)	{"LeftFrontalElbow": -0.2}
JointVelocityMapWaveBack	Map(Float)	{"LeftFrontalElbow": -0.5}
JointVelocityMapWaveFwd	Map(Float)	{"LeftFrontalElbow": 0.5}

Table C.2: Joint positions and velocities parameters.

To add this values: right-click on *MainState* $\Longrightarrow$ Edit State $\Longrightarrow$ InputParameters. Then we proceed like in Figure C.7. Here the *JointValueMapInitPose*'s values are added.

A small note here: it is important for adding the values in JSON to make sure we added the “” char correctly. ArmarX is sensible to the difference and will only accept "".



	Key	Type	Optional	De	TEOSimulation	TEOBase	Root
1	counterMaxValue	int	false		4	Click to set	Click to set
2	JointValueMapInitPose	Map(float)	false		Edit	Click to set	Click to set
3		string					k to set

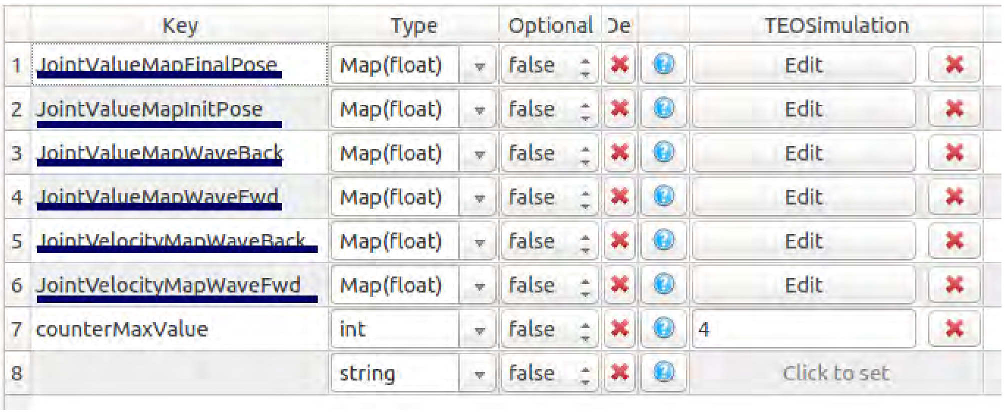
```
{
"LeftFrontalShoulder": -0.72,
"LeftSaggitalShoulder": 0.95,
"LeftAxialShoulder": -0.73,
"LeftFrontalElbow": -1.75,
"LeftFrontalWrist": -1
}
```

Input is valid.

Libraries to Link: ArmarXCoreObservers

Figure C.7: Adding value to the *JointValueMapInitPose* variable.

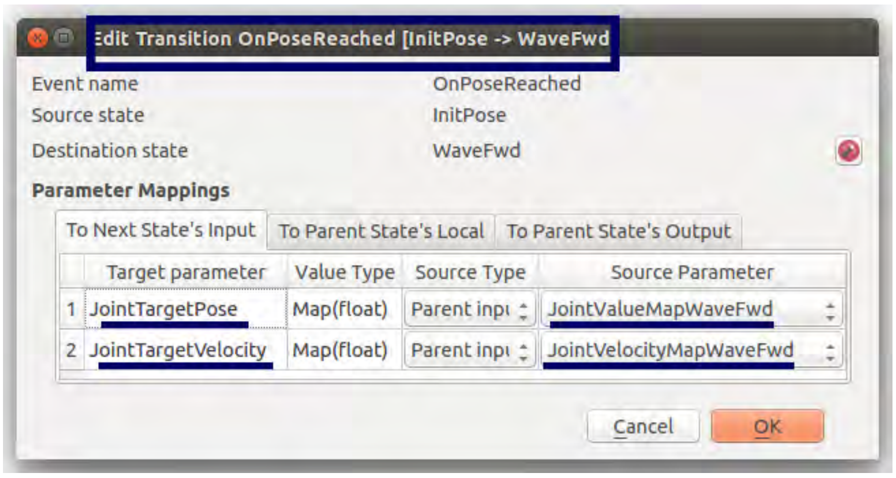
After adding all the parameters in Table C.2, the input parameters tab looks like Figure C.8 now.



	Key	Type	Optional	De	TEOSimulation
1	JointValueMapFinalPose	Map(float)	false		Edit
2	JointValueMapInitPose	Map(float)	false		Edit
3	JointValueMapWaveBack	Map(float)	false		Edit
4	JointValueMapWaveFwd	Map(float)	false		Edit
5	JointVelocityMapWaveBack	Map(float)	false		Edit
6	JointVelocityMapWaveFwd	Map(float)	false		Edit
7	counterMaxValue	int	false		4
8		string	false		Click to set

Figure C.8: Input parameters of *MainState*.

Now we need to edit all the transitions in our statechart diagram to tell them which parameters to pass and from where they come. To do so, right click on the arrow that represents a transition. The parameters expected by the target state will already be displayed. All we have to do is to select *Parent input* as Source Type and choose a feasible source parameter from the drop-down list (See Figure C.9). We already prepared the names of the parameters, so it would be a simple task. Nonetheless, Table C.3 summarizes which parameter goes with each transition.



Edit Transition OnPoseReached [InitPose -> WaveFwd

Event name	OnPoseReached
Source state	InitPose
Destination state	WaveFwd

Parameter Mappings

To Next State's Input | To Parent State's Local | To Parent State's Output

	Target parameter	Value Type	Source Type	Source Parameter
1	JointTargetPose	Map(float)	Parent inp	JointValueMapWaveFwd
2	JointTargetVelocity	Map(float)	Parent inp	JointVelocityMapWaveFwd

Figure C.9: Transition from *InitPose* to *WaveFwd*.

Source ⇒ Destination	**Target Parameter**	**Source Paramter**
InitPose ⇒ WaveFwd	JointTargetPose JointTargetVelocity	JointValueMapWaveFwd JointVelocityMapWaveFwd
WaveFwd ⇒ CounterState	counterId counterMaxValue	counterId counterMaxValue
CounterState ⇒ WaveBack	JointTargetPose JointTargetVelocity	JointValueMapWaveBack JointVelocityMapWaveBack
WaveBack ⇒ WaveFwd	JointTargetPose JointTargetVelocity	JointValueMapWaveFwd JointVelocityMapWaveFwd
CounterState ⇒ FinalPose	JointTargetPose	JointValueFinalPose

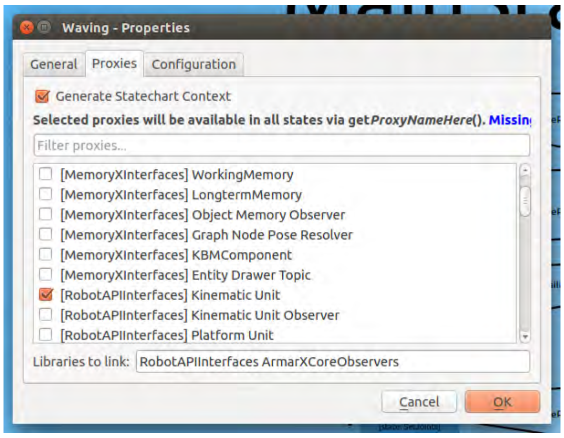
Table C.3: StateChart transition parameter mapping

C.3 Preparing the Kinematic Unit

In order to manipulate the joint angles, we need access to a KinematicUnit-Interface. A kinematic unit, which implements a KinematicUnitInterface, is a sensor-actor-unit that allows us to control the joints of a robot. Here, this will allow us to make an arm movement of TEO simulation. The GUI plugin and its widget will provide a visual feedback of this simulation.

First, right-click the *Waving* and select *Group Properties*. In the tab *Proxies*, we must select **[RobotAPIInterfaces] Kinematic Unit**. Also, we must go to the tab *Configuration* to select a profile (in this case, *TEOSimulation*). Now in the configuration box we write:

$ArmarX.WavingRemoteStateOfferer.KinematicUnitName = Armar3KinematicUnit$


Waving - Properties

General | Proxies | Configuration

☑ Generate Statechart Context

Selected proxies will be available in all states via get*ProxyNameHere*(). Missin

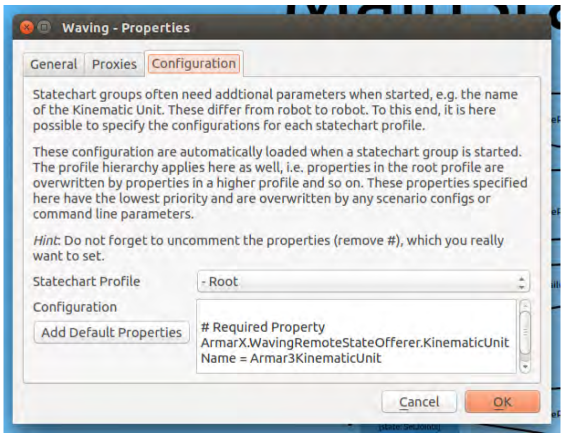
Filter proxies...

- ☐ [MemoryXInterfaces] WorkingMemory
- ☐ [MemoryXInterfaces] LongtermMemory
- ☐ [MemoryXInterfaces] Object Memory Observer
- ☐ [MemoryXInterfaces] Graph Node Pose Resolver
- ☐ [MemoryXInterfaces] KBMComponent
- ☐ [MemoryXInterfaces] Entity Drawer Topic
- ☑ [RobotAPIInterfaces] Kinematic Unit
- ☐ [RobotAPIInterfaces] Kinematic Unit Observer
- ☐ [RobotAPIInterfaces] Platform Unit

Libraries to link: RobotAPIInterfaces ArmarXCoreObservers

Cancel | OK

(a) Proxies tab.


Waving - Properties

General | Proxies | Configuration

Statechart groups often need addtional parameters when started, e.g. the name of the Kinematic Unit. These differ from robot to robot. To this end, it is here possible to specify the configurations for each statechart profile.

These configuration are automatically loaded when a statechart group is started. The profile hierarchy applies here as well, i.e. properties in the root profile are overwritten by properties in a higher profile and so on. These properties specified here have the lowest priority and are overwritten by any scenario configs or command line parameters.

Hint: Do not forget to uncomment the properties (remove #), which you really want to set.

Statechart Profile: - Root

Configuration

Add Default Properties

```
# Required Property
ArmarX.WavingRemoteStateOfferer.KinematicUnit
Name = Armar3KinematicUnit
```

Cancel | OK

(b) Configuration tab.

Figure C.10: Waving StateChart Group properties.

C.4 Adding dependencies

In order to use the Kinematic Unit proxy we need to add a dependency on RobotAPI. To do so, we must edit the **CMakeLists.txt** in the package *TEOWave*. The Listing C.1 shows the final look of the file. Under the commented lines (17-18) we added the line 20, that will include the RobotAPI dependency.

```
1  # TEOMove
2
3  cmake_minimum_required(VERSION 2.8)
4
5  find_package("ArmarXCore" REQUIRED)
6  # Include provides all necessary ArmarX CMake macros
7
8  include(${ArmarXCore_USE_FILE})
9
10 set(ARMARX_ENABLE_DEPENDENCY_VERSION_CHECK_DEFAULT FALSE)
11
12 set(ARMARX_ENABLE_AUTO_CODE_FORMATTING TRUE)
13
```

```
# Name for the project

armarx_project("TEOMove")
# Specify each ArmarX Package dependency with the following macro
# depends_on_armarx_package(ArmarXGui "OPTIONAL")

depends_on_armarx_package(RobotAPI)

add_subdirectory(etc)

add_subdirectory(source)

install_project()
```

Listing C.1: TEOMove/CMakeLists.txt

Another CMakeLists must be edited, the one of our Statechart Group (TEOMove/source/TEOMove/Statecharts/Waving/CMakeLists.txt), we need to add the dependencies ”Eigen3” and ”Simox”. To do so, uncomment the lines 14 to 27. Also we must check that we edit the component list so it appears like lines 29 to 31. Listing C.2 already shows the changes made in these lines.

```
 1  # Waving CMakeLists
 2
 3  armarx_component_set_name("Waving")
 4
 5  #find_package(MyLib QUIET)
 6  #armarx_build_if(MyLib_FOUND "MyLib not available")
 7  #
 8  # all include_directories must be guarded by if(Xyz_FOUND)
 9  # for multiple libraries write: if(X_FOUND AND Y_FOUND)....
10  #if(MyLib_FOUND)
11  #    include_directories(${MyLib_INCLUDE_DIRS})
12  #endif()
13
14  find_package(Eigen3 QUIET)
15  find_package(Simox QUIET)
16
17
18  armarx_build_if(Eigen3_FOUND "Eigen3 not available")
19  armarx_build_if(Simox_FOUND "Simox-VirtualRobot not available")
20
21
22  if (Eigen3_FOUND AND Simox_FOUND)
23    include_directories(
24      ${Eigen3_INCLUDE_DIR}
25      ${Simox_INCLUDE_DIRS}
26  )
27  endif()
28
29  set(COMPONENT_LIBS
30  RobotAPIInterfaces RobotAPICore
31  ArmarXCoreInterfaces ArmarXCore ArmarXCoreStatechart ArmarXCoreObservers)
32
33  # Sources
```

```

set(SOURCES
WavingRemoteStateOfferer.cpp
./MainState.cpp
./CounterState.cpp
./SetJoints.cpp
./MoveJoints.cpp
#@TEMPLATE_LINE@@COMPONENT_PATH@/@COMPONENT_NAME@.cpp
)

set(HEADERS
WavingRemoteStateOfferer.h
Waving.scgxml
./MainState.h
./CounterState.h
./SetJoints.h
./MoveJoints.h
#@TEMPLATE_LINE@@COMPONENT_PATH@/@COMPONENT_NAME@.h
./MainState.xml
./CounterState.xml
./SetJoints.xml
./MoveJoints.xml
./MoveJoints.xml
#@TEMPLATE_LINE@@COMPONENT_PATH@/@COMPONENT_NAME@.xml
)

armarx_add_component("${SOURCES}" "${HEADERS}")
```

Listing C.2:
TEOMove/source/TEOMove/Statecharts/Waving/CMakeLists.txt

Now we must build the project by running cmake inside QTCreator.

C.5 Editing the Source Code

So far we only defined the OnPoseReached event in MoveJoints and SetJoints. We have not yet specified when these events will be triggered. This is what will do now in the .cpp-source of our two states.

Now we will open both **MoveJoints.cpp** and **SetJoints.cpp** and add the code fragment in Listing C.3.

```
// get the target joint values
std::map<std::string, float> jointValueMap = in.
    getJointTargetPose();
//build conditions for OnPoseReached
Term poseReachedConditions;
const float eps = 0.05f; //This will trigger the OnPoseReached
    event if the actual pose is very close to the specified pose
     (+/- 0.05).
for (const auto& jointNameValue : jointValueMap)
```

```
{
    std::string jointNameDatafield = "
       Armar3KinematicUnitObserver.jointangles." +
       jointNameValue.first;
    float jointValue = jointNameValue.second;
    Literal jointValueReached(jointNameDatafield, "inrange",
                              Literal::createParameterList(
                                 jointValue - eps, jointValue +
                                 eps));
    poseReachedConditions = poseReachedConditions &&
       jointValueReached;
}
installConditionForOnPoseReached(poseReachedConditions);
```

Listing C.3: *onEnter* code fragment for both *SetJoints* and *MoveJoints*.

By far, we have not told the two states what they are going to do, this part goes in the *run()* function. First, for *MoveJoints*, all content was replaced by the Listing C.4.

```
void MoveJoitns::run(){
std::map<std::string, float> jointVelocityMap = in.
    getJointTargetVelocity();
NameControlModeMap velocityControlModeMap;
for (const auto& jointVelocity : jointVelocityMap)
{
    velocityControlModeMap[jointVelocity.first] =
       eVelocityControl;
}
KinematicUnitInterfacePrx kinUnit = getKinematicUnit();
kinUnit->switchControlMode(velocityControlModeMap);
kinUnit->setJointVelocities(jointVelocityMap);
}
```

Listing C.4: MoveJoints.cpp *run()* function.

First, we tell the kinematic unit that we wish to velocity control all the joints defined in our input joint velocity vector. Then, in the last line we set the desired velocity.

Now with *SetJoints* we will do similar, but the code will be the one in Listing C.5. The core of what it does is in the last line: **it sets the value of the angles to the Map Value we introduced**.

```
void SetJoints::run(){
std::map<std::string, float> jointValueMap = in.
    getJointTargetPose();
```

```cpp
NameControlModeMap positionControlModeMap;
//sets to position control mode the joints in the map
for (const auto& jointNameValue : jointValueMap)
{
    positionControlModeMap[jointNameValue.first] =
        ePositionControl;
}
KinematicUnitInterfacePrx kinUnit = getKinematicUnit();
//switch to position control
kinUnit->switchControlMode(positionControlModeMap);
// set the angles defined by the joint target pose
kinUnit->setJointAngles(jointValueMap);
}
```

Listing C.5: SetJoints.cpp *run().* function

Before building out project (Run CMake and Build Project), we uncomment the declarations of the functions: *onBreak(), run() and onExit()* in the .h file.

C.6 Importing TEO model

To work with our robot **TEO**, we first must clone the files from the RoboticsLab-Uc3m GitHub repository. Under the name of **teo-simox-models**. To do so, we execute the commands of Listing C.6 in the terminal. It is highly recommended to clone them in a repository folder, here *repos*, not directly in the working location.

```
git clone https://github.com/roboticslab-uc3m/teo-simox-models.git
```

Listing C.6: Cloning TEO directory.

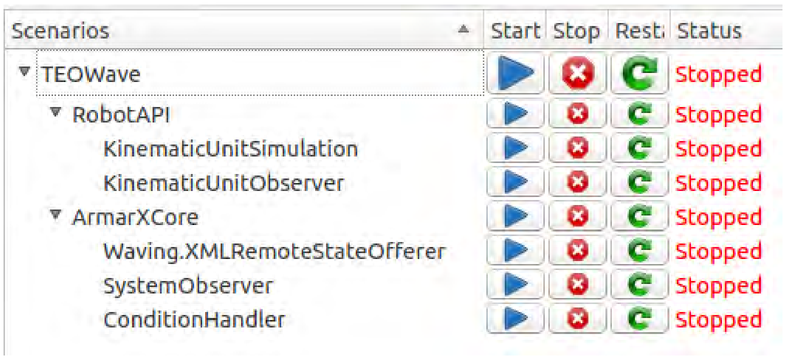
Then we will copy it to the package, but it is very important the location. Because later in the Scenario we will refer to the robot file from the **data** folder. To do so, we will just copy from the terminal, with *root* permissions (Listing C.7).

```
cp -a teo-simox-models/ packages/TEOMove/data/TEOMove/
```

Listing C.7: TEO models copied and ready to be used.

C.7 Deploying the Scenario

Now the simulation almost finished. We will start the ScenarioManager and open the scenario we created earlier in Appendix B: *TEOWave*. Once open, we must add two new applications: *KinematicUnitObserver* and **KinematicUnit-Simulation**. The application KinematicUnitSimulation creates a simulated kinematic unit. KinematicUnitObserver creates an observer that observes our simulated kinematic unit, which we need to detect when we reached our target poses. After adding both applications, our scenario should look like Figure C.11.



Scenarios	Start	Stop	Rest	Status
TEOWave	▶	✖	↻	Stopped
RobotAPI	▶	✖	↻	Stopped
KinematicUnitSimulation	▶	✖	↻	Stopped
KinematicUnitObserver	▶	✖	↻	Stopped
ArmarXCore	▶	✖	↻	Stopped
Waving.XMLRemoteStateOfferer	▶	✖	↻	Stopped
SystemObserver	▶	✖	↻	Stopped
ConditionHandler	▶	✖	↻	Stopped

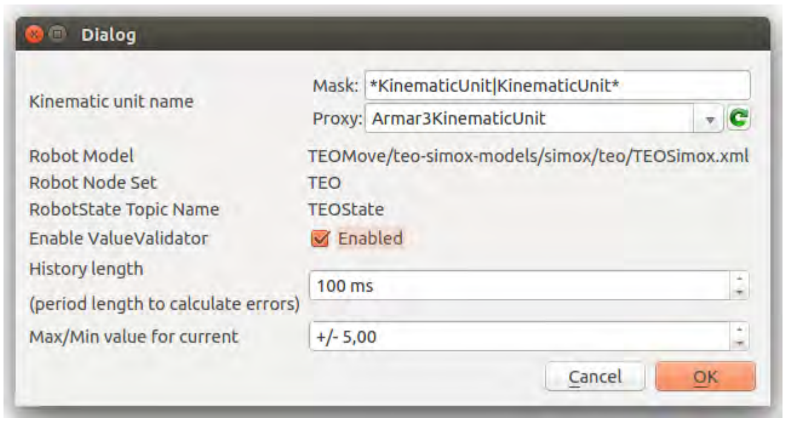
Figure C.11: Final stage of our scenario.

Some parametrization of the applications must be done now, in order to include the robot. Table C.4 shows the fields we must change and its values for both *KinematicUnitObserver* and *KinematicUnitSimulation*.

ArmarX.KinematicUnitObserver .RobotFileName	TEOMove/teo-simox-models/ simox/teo/TEOSimox.xml
ArmarX.KinematicUnitObserver .RobotNodeSetName	TEO
ArmarX.KinematicUnitObserver .ObjectName	Armar3KinematicUnitObserver

Table C.4: KinematicUnitObserver parameters

Now we start the scenario. In order to visualize the 3D model of TEO, we go to AddWidget $\Longrightarrow$ RobotControl $\Longrightarrow$ KinematicUnitGui.
Automatically a dialog will show up, in order to check the parameters. If the window looks like Figure C.12, everything is correct, so we click on **OK**.

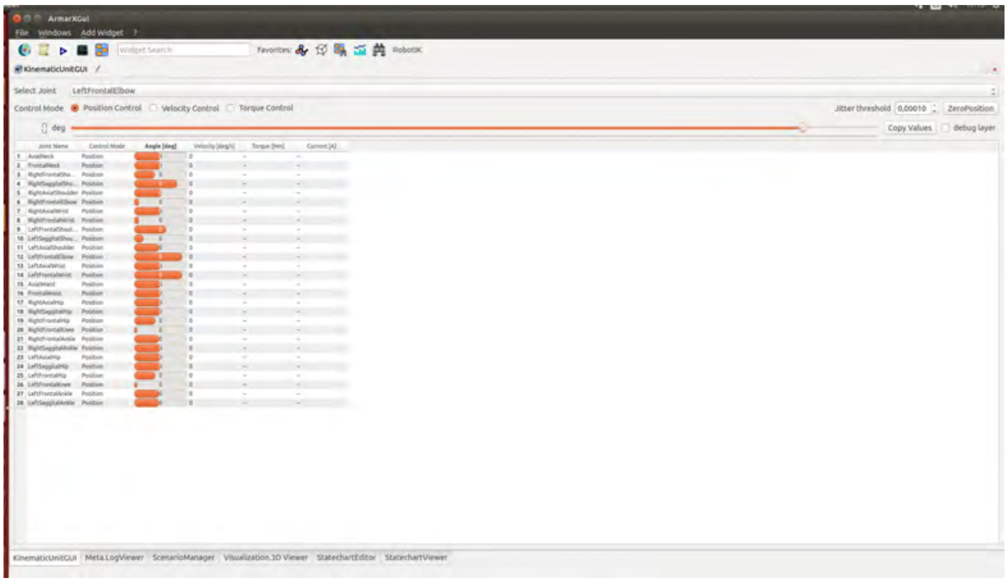


Kinematic unit name
Mask: *KinematicUnit|KinematicUnit*
Proxy: Armar3KinematicUnit
Robot Model: TEOMove/teo-simox-models/simox/teo/TEOSimox.xml
Robot Node Set: TEO
RobotState Topic Name: TEOState
Enable ValueValidator: [✓ Enabled]
History length (period length to calculate errors): 100 ms
Max/Min value for current: +/- 5,00

Figure C.12: KinematicUnitGui dialog

C.8 Visualizing the Simulation

Two new Widgets have appear, **KinematicUnitGUI** will allow us to control manually the joints of our robot. Also provides real-time values of each aspect (position, velocity, acceleration and torque) for every joint.



Select Joint: LeftFrontalElbow

Control Mode: [✓ Position Control] Velocity Control Torque Control

deg

Jitter threshold: 0,00010

	Joint Name	Control Mode	Angle [deg]	Velocity [deg/s]	Torque [Nm]	Current [A]
1	AxialNeck	Position	0	0	-	-
2	FrontalNeck	Position	0	0	-	-
3	RightFrontalSho...	Position	0	0	-	-
4	RightSaggitalSho...	Position	0	0	-	-
5	RightAxialShoulder	Position	0	0	-	-
6	RightFrontalElbow	Position	0	0	-	-
7	RightAxialWrist	Position	0	0	-	-
8	RightFrontalWrist	Position	0	0	-	-
9	LeftFrontalShoul...	Position	0	0	-	-
10	LeftSaggitalShou...	Position	0	0	-	-
11	LeftAxialShoulder	Position	0	0	-	-
12	LeftFrontalElbow	Position	0	0	-	-
13	LeftAxialWrist	Position	0	0	-	-
14	LeftFrontalWrist	Position	0	0	-	-
15	AxialWaist	Position	0	0	-	-
16	FrontalWaist	Position	0	0	-	-
17	RightAxialHip	Position	0	0	-	-
18	RightSaggitalHip	Position	0	0	-	-
19	RightFrontalHip	Position	0	0	-	-
20	RightFrontalKnee	Position	0	0	-	-
21	RightFrontalAnkle	Position	0	0	-	-
22	RightSaggitalAnkle	Position	0	0	-	-
23	LeftAxialHip	Position	0	0	-	-
24	LeftSaggitalHip	Position	0	0	-	-
25	LeftFrontalHip	Position	0	0	-	-
26	LeftFrontalKnee	Position	0	0	-	-
27	LeftFrontalAnkle	Position	0	0	-	-
28	LeftSaggitalAnkle	Position	0	0	-	-

Figure C.13: KinematicUnitGui widget.

But we will focus on the **Visualization.3D Viewer**. Here we can see the 3D representation of the robot's actual state. It we followed this steps and finally run the scenario, we can see TEO in the ArmarX environment, like in Figure C.14.

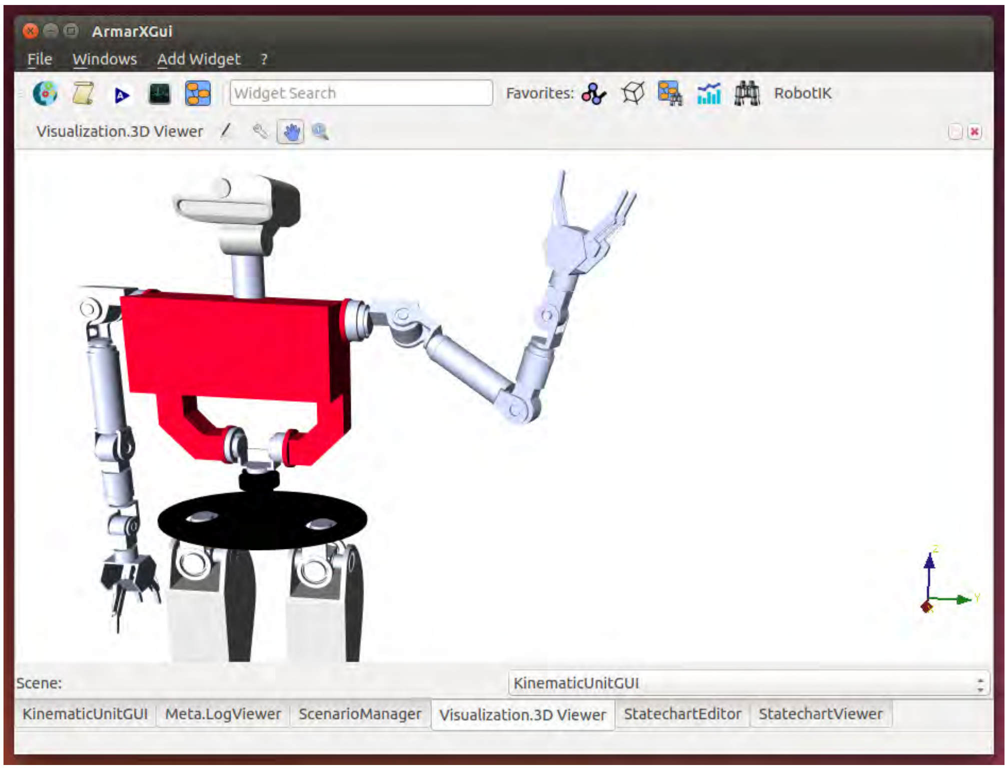


Figure C.14: TEO waving ArmarX.

References

[1] J. Kramer and M. Scheutz, “Development environments for autonomous mobile robots: A survey,” *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, Feb 2007. [Online]. Available: https://doi.org/10.1007/s10514-006-9013-8

[2] N. Ando, T. Suehiro, and T. Kotoku, *A Software Platform for Component Based RT-System Development: OpenRTM-Aist*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 87–98. [Online]. Available: https://doi.org/10.1007/978-3-540-89076-8_12

[3] F. Kanehiro, H. Hirukawa, and S. Kajita, “Openhrp: Open architecture humanoid robotics platform,” *The International Journal of Robotics Research*, vol. 23, no. 2, pp. 155–165, 2004. [Online]. Available: https://doi.org/10.1177/0278364904041324

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[5] H. Bruyninckx, “Open robot control software: the orocos project,” in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3, 2001, pp. 2523–2528.

[6] G. Metta, P. Fitzpatrick, and L. Natale, “Yarp: Yet another robot platform,” *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006. [Online]. Available: https://doi.org/10.5772/5761

[7] N. Vahrenkamp, M. Wächter, M. Kröhnert, K. Welke, and T. Asfour, “The robot software framework armarx.” *it - Information Technology.*, vol. 57, no. 2, pp. 99–111, 2015.

[8] T. Asfour, K. Regenstein, P. Azad, J. Schroder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann, “ARMAR-III: An integrated humanoid platform for sensory-motor control,” in *2006 6th IEEE-RAS International Conference on Humanoid Robots*, Dec 2006, pp. 169–175.

[9] T. Asfour, J. Schill, H. Peters, C. Klas, J. Bücker, C. Sander, S. Schulz, A. Kargov, T. Werner, and V. Bartenbach, “Armar-4: A 63 dof torque controlled humanoid robot,” in *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, Oct 2013, pp. 390–396.

[10] A. Paikan, D. Schiebener, M. Wächter, T. Asfour, G. Metta, and L. Natale, “Transferring object grasping knowledge and skill across different robotic platforms,” in *2015 International Conference on Advanced Robotics (ICAR)*, July 2015, pp. 498–503.

[11] G. Metta, G. Sandini, D. Vernon, L. Natale, and F. Nori, “The iCub humanoid robot: An open platform for research in embodied cognition,” in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent*

Systems, ser. PerMIS '08. New York, NY, USA: ACM, 2008, pp. 50–56. [Online]. Available: http://doi.acm.org/10.1145/1774674.1774683

[12] P. Pierro, S. Martinez, A. Jardon, C. Monje, and C. Balaguer, "Teo: Full-size humanoid robot design powered by a fuel cell system," *An International Journal on Cybernetics and Systems*, vol. 43, no. 3, pp. 163 – 180, 2012.

[13] M. D. P. del Valle, "Balance control of humanoid robot teo using force/torque sensors," Master's thesis, University Carlos III, sep 2017.

[14] S. Morante, J. G. Victores, and C. Balaguer, "Automatic demonstration and feature selection for robot learning," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*, Nov 2015, pp. 428–433.

[15] D. Estevez, J. G. Victores, R. Fernandez-Fernandez, and C. Balaguer, "Robotic ironing with 3d perception and force/torque feedback in household environments," *CoRR*, vol. abs/1706.05340, 2017. [Online]. Available: http://arxiv.org/abs/1706.05340

[16] J. Hernández, J. Miguel García Haro, S. Martinez, J. Lorente, and C. Balaguer, "Manipulation balance control system by computer vision tools," 2016.

[17] J. Lorente, J. Miguel García Haro, S. Martinez, J. Hernández, and C. Balaguer, "Waiter robot: Advances in humanoid robot research at uc3m," 05 2016.

[18] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, Jan 2004.

[19] N. Vahrenkamp and M. Wächter, "Armarx, the event-driven component-based robot software development environment." [Online]. Available: https://armarx.humanoids.kit.edu/index.html

[20] K. Welke, P. Kaiser, A. Kozlov, N. Adermann, T. Asfour, M. Lewis, and M. Steedman, "Grounded spatial symbols for task planning based on experience," in *Humanoid Robots (Humanoids), 2013 13th IEEE-RAS International Conference on*. IEEE, 2013, pp. 484–491.

[21] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987.

[22] M. Wächter, S. Ottenhaus, M. Kröhnert, N. Vahrenkamp, and T. Asfour, "The armarx statechart concept: Graphical programing of robot behavior," *Frontiers in Robotics and AI*, vol. 3, p. 33, 2016. [Online]. Available: http://journal.frontiersin.org/article/10.3389/frobt.2016.00033

[23] N. Vahrenkamp, "Simox, a platform-independent for robot simulation, motion and grasp planning." [Online]. Available: https://gitlab.com/Simox