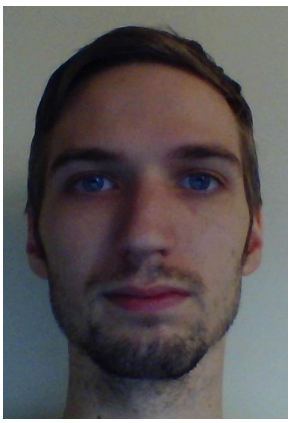
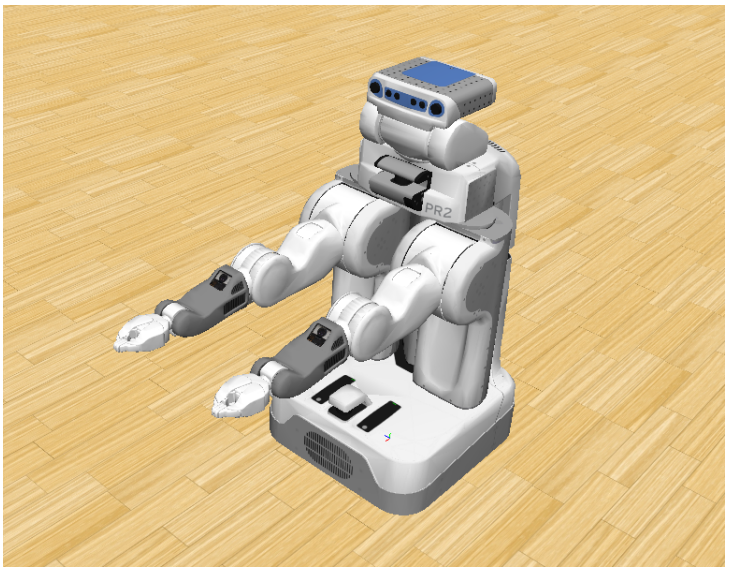


ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Master Thesis
LSRO

ROS interface and URDF parser for Webots



Simon Puligny

Professor : Mondada Francesco
Advisor : Olivier Michel
Date : 14 February 2014

Mobile Robot Modeling, Simulation and Programming

Simon Puligny, Section Microtechnique

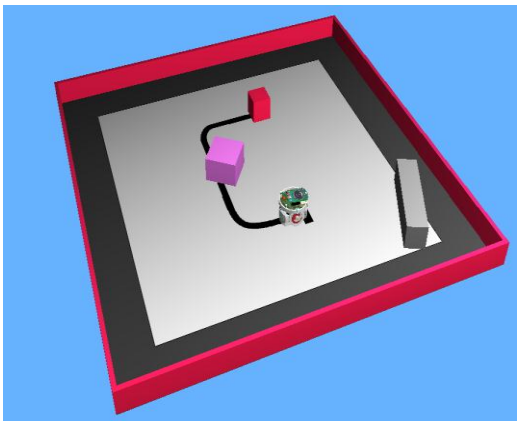
Supervisor: *Olivier Michel*

Professor: *Mondada Francesco*

Ros controller

The main goal of this project has been to develop a robot controller for the Webots simulator that would interface with the ROS framework. The controller has been written in C++ language. It can be run on Windows, Mac OS X and Linux platforms and on any robot available in the simulator.

It will connect to a ROS network and use topics and services to communicate with other ROS-nodes on the network. The other nodes will be able to read the value of the sensors of the robot and send it commands and instructions to control the robot inside the simulator.



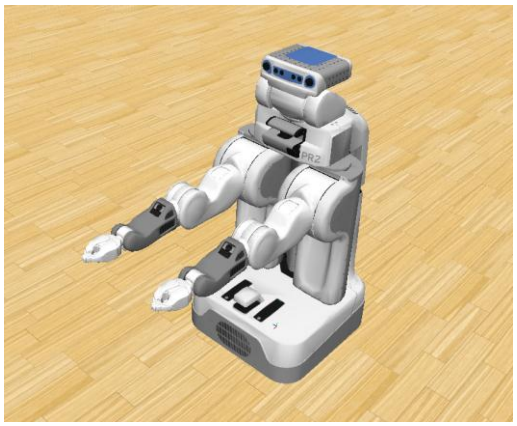
the ros controller is running on an e-puck robot to follow a line and avoid obstacle. The instructions are given by a ROS-node running on another computer.

The focus has been on making the controller generic for any application, very to use and with the minimum amount of dependencies. A particular attention has been given to the quality of the code thanks to a long work of review.

A set of examples have been designed to illustrate some of the possible applications of the controller.

URDF to VRML tool

The second part of the project has been to design a Python script that take a URDF file and creates a robot model equivalent in the Webots simulator. URDF is a file format used to describe a mechanical model of robots. The script developed will parse the file to get the different part of the robot and create a model in VRML format that have the same mechanic properties inside the simulated world.



Webots model of the PR2 produced from the URDF description of the robot

The PR2 robot of Willow Garage has been chosen to illustrate this script. The model is very heavy, has complex geometries and has a large number of joints and sensors.

This project has been done in Cyberbotics company at the Innovation Park of the EPFL.

Table des matières

1 Introduction

There is many actors in robotics across the world and it is really hard for all of them to agree on standards. Standard helps researcher share their knowledge and science to move forwards. When a good solution emerges and become more and more used, it should be adopted by as many persons as possible. The ROS framework and the URDF robot description format are two of them and this is why the goal of this project has been to integrate them in the Webots software to allows users to work with these tools.
ROS was developed mainly by Stanford University and Willow Garage and many other institutions contributed to it since 2007. It is an open-source project maintained by Willow Garage that is now widely used around the globe and keeps growing on every day.
There exists many different kinds of robots for a multitude of applications and its almost impossible to find an intuitive and smart way to describe any of them. While it has some limitations, the URDF format allows to represents a lot of robot while having a fairly simple structure that makes him esteemed by many researchers.

2 ROS controller

ROS is a great tool that is now commonly used in robotic research and Cyberbotics wants to offer to its users the possibility to easily use it with Webots. The goal was to be able to use ROS with the robots inside the simulated world of Webots just like you would do with robots in the real world.

2.1 ROS overview

ROS is a 100% free software framework for robotic purpose that offers many tools for communication between robots and other devices or software through a TCP/IP network. ROS is split in two parts :
The core software is provided and maintained by Willow Garage. It contains the different API to create a ROS network and to create the ROS-nodes that can use the network. It also contains scripts and command-lines to monitor the connections, nodes and messages on the network and to interface ros with other software such as rviz or gazebo. The homepage have a long list o tutorials that go from setting up your workspace to create nodes able to control a robot from a remote computer and stream data to another software at the same time.
The other side of ROS is a very large database of packages that are referenced on ROS website. These packages are provided by many different companies or groups and try to follow the same norms. They all have a different goal and are usually very specific to an application or a device. You can find new complex kinds of message to extend the existing list, libraries to process some data or controllers and drivers for a specific robot. Most of these packages have a really good documentation and some tutorials. you can even find an e-mail address to get support if ever you have a problem or wishes to contribute to it. All of this make creating a ROS-node really easier.

2.2 ROS communication

The best argument of ROS is its communication network. Communication is based on the TCP/IP protocol with each node connecting with a socket. The server is administrated by a master that handles all the connection and addressing details.
The communication is anonymous between the devices offering hardware abstraction. The topics are also asynchronous meaning you can store messages and read them afterwards so that you don't miss anything or have to slow down a process to wait for another one. Of course you may sometimes wants this communication to be synchronous and ROS also have an alternative for that with services.
Since ROS has been purely designed for robotic purpose, the standard definitions of message doesn't contains complex structures that you may find with other framework and that would overload the robot but instead have some unique messages specially for robots.

There is a few key concepts that the user must know in order to understand ROS communication :

- Nodes : They are process that can perform computation, execute some tasks and communicate thanks to the ROS network. It will register to the network with a unique id and a list of topics and services that he wants to use to send or receive messages and some additional connection parameters. ROS provides libraries to write the nodes with the C++ or the Python languages.

- Master : The master is a special node that will be launched when you start the network. It will handle the registration, subscriptions and disconnection of every node to the network and links for each topic or service so that messages will be able to reach its target successfully. The master also store all of the parameter server.

– Messages : Packets sent on the network are defined in ROS messages. A message is a simply a data structure containing typed fields. The following standard type of fields are supported by ROS : boolean, (un)signed int 8/16/32 bits, float 32/64 bits, string, two ros-specific types : time and duration. You can also uses are of these types. Additional types can be found in packages that are structures of combination of the standard types.

– Topics : The topics are a transport system with publish and subscribe semantics used to send messages. A node can connect to a topic by its name either as a publisher in order to send data or as a subscriber in order to receive these data. There is no limit to the number of publisher and subscriber to a topic. The node won't have any information about the other users of the topic. Messages sent by a publisher will be received by all subscribers and each subscriber will receive the messages from all publishers. The topic have a defined message template and it is not possible to send other kind of data with this topic.

– Services : They are an alternative to the topics that doesn't use the publish/subscribe system but instead uses a request/response model. A node that uses service will only receive data in response to the query it made. The response may vary depending on the information given in the request. A service is provided by only one node and only one request can be send at the same time. The topic contain description of both the request and the response messages type.

2.2.1 ROS package creation

ROS comes with its own build system. This system has been reshaped in ROS Groovy version and it is the new catkin system that is described below.

Once you have installed ROS on your computer, you can make a directory where you will build your package. With the *catkin_ init_ workspace* command, you will setup your directory to be able to build ROS packages and this directory will be referenced as your catkin workspace(catkin_ws). You can now create your new node or new library file that uses the ROS C++ or Python inside this directory.

This directory can also contain two sub-folders *msg* and *srv*. In the first one you can define a new structure of messages in a text file with the .msg extension. The new message structure will have the same name as the file. The structure is directly written in the file. It will only contain a list where the first element of each row will be the type of the field and the second element will be the name of the field. Optionally, you can define constant too inside this file by adding the equal sign and the constant value after the name of the field.

The second optional folder contain definition of the services. Each service has a file with the extension .srv associated with. The structure of these files is very similar to the definition of message. It first contain the definition of the fields of the request followed by the definition of the fields of the response. A row of three dashes is used to separate the two messages.

Once you have created all your file, there is only two file that needs to be updated : a package.xml that provides meta information about your package and a CMakeLists.txt in which you will add all rules to build the files you created before. Once this is done the build command *catkin_ make* will build your package and update the setup.sh file of your package. Once compilation has been successful, your package is ready to be used or distributed.

2.3 Previous version of the controller

An example of a robot controller interfacing Webots with ROS already existed within Webots. This example used a ROS package to interface a joystick with a robot in Webots to drive it. This was only an example working for this application but that would need to be almost entirely

rewritten in order to serves for another application. The compilation is also trickery as it uses both the CMake build of ROS and gcc of Webots simultaneously and need both of them to be installed and setup correctly. This solution is not scalable and we decided to build a new version from scratch for a new controller. The old controller has been kept within Webots files. It is now designated as a *custom* controller that the user can look at if he wants to build his own controller for an application that could not be done with the new controller.

2.4 Generic Controller for Webots

2.4.1 Specifications

When designing the new controller for Webots that would interface with ROS, we settled a several objectives :
- The controller can be run on any robot existing in Webots.
- It must be very user-friendly as Webots users may not know ROS API very well or ROS user may not know Webots API either.
- It should have the minimal set of dependencies over ROS or other libraries in order to be easy to maintain
- There should be no issue with running multiple instances of the controller in the same world or on multiple instances of Webots connected to the same ROS network.

2.4.2 Language choice

Our first choice have been to write the ros controller in C language as this is the most common language for robotic programming. While I was still at the beginning, I encountered many difficulties to mix the ROS API and the c controller of Webots together and avoid memory leaks while handling all the topics and services.
Therefore we decided to drop the C controller and instead choose one of the language used by ROS. We had choice between C++ and Python and Webots has a controller library for both of them too. We choose C++ over Python because it is often used for embedded software while Python is very rare as it is generally slower and uses more memory. There is also much more support and a larger community for ROS with the C++ API.

2.4.3 Design of the controller

For the design of the ros controller, we chose a structure very close to the structure of the C++ controller of Webots. There is a main class for the robot defined in Ros.cpp and then a class for each existing device in Webots with 2 intermediate abstract classes. The device classes have dependencies on the corresponding device class in the C++ controller of Webots and on the ROS API for services and topics.
In order to be as user-friendly as possible we chose to stay as close as possible to the C++ controller for the functions too. Each device or robot will have a topic or a service available for each function of the C++ controller. This way it will be easy for users that have already used the C++ controller of Webots to write ROS nodes for this new ros controller or to transform a previous C or C++ controller into a ROS node.

A lot of thinking has been done before defining whether the ros controller should use topics or services. Topics allow to constantly send or receive data while services requires a trigger. While it could be easier to only use one of them, they both have advantages that make them more suitable to a specific situation and less to others.
Another aspect has been the performances they both have. An important difference between both that is often overlook is how they handle their connections to the network. For topics, once a publisher and a subscriber have connected to it, a connection is created between them and remain open until either of them shut-down or leave the topic. For services, a new connection is initialized each time a request is sent and it is immediately closed once the response has been

sent back. this means that topics have almost no delay before sending data, while topics have to wait for the initialization every time.
Since Webots can have a large number of robots running the ros controller, we had to make sure we don't have too much connection opened at the same time and overload the network. This is the reason why we decided to use mainly services and keeps topic only for absolute necessity. We used them only for sending data from the sensors when the user enable a sensor and measure its value at each time step.

2.4.4 Device classes in ros controller

In this section, I'll explain how in general a device class is organized in the ros controller.

The class always contains a pointer to an object of the device class of the C++ controller and inherits from the abstract class RosDevice. They then have services for each method of the device class. The constructor will call the constructor of the device C++ class and then add their services to the list of available services of the robot's ROS-node on the network.

If the device is a sensor the ros controller class doesn't directly inherits from RosDevice but instead from an intermediate RosSensor that itself inherits from RosDevice. While devices are all different, the sensor have always the same 4 functions among others : a enable and a disable function, a function to get the sampling period set on the device and a getValue function to get the sensor measurement. The RosSensor class have a virtual function for each of them. The getValue function of the sensors is the only one that uses the topic functionality of the ROS API.

Multiple ROS-nodes can interact with the ros controller at the same time and they can each want to use the sensor's data and not always with the same sampling period. The RosSensor class will creates a topic to publish the data for each required sampling period and will close them dynamically if there is no more subscriber. It will store the list of the topics it created so that the robot can check at each time step on which topic it should publish the value of the sensor.

In order to reduce the dependencies of the controller over ROS API to the minimum, we choose to use only the standard messages and only the functions to create and close topics and services. Some message types were quite interesting to use but that would have mean that the user should have the package for these message on every machine that uses the ROS-node and that we would have to maintain this packages too as it isn't made by ROS development team.

2.4.5 Ros main class

The principal and most important class of the ros controller is the robot class. This main class will manage the robot, each device, the ROS-node and the communication with the server.

The controller must be able to run on any kind of robot existing in Webots. This robot can either be a simple robot, a DifferentialWheels robot or a supervisor with any set of devices on it. This means we can't have a fixed class with the exact members written inside. The solution has been to have only empty pointers to the three robot classes and an empty list of RosDevice pointers instead. The controller will create the objects it needs according to the robot description. We mixed a small part of the C controller of Webots in order to be able to detect the type of the robot and call the right constructor.

The class also have the ROS-node of the controller. It will handle registration and shut-down of the node and control the publication of messages on the topics. since several ros controller can run simultaneously, we had to find a way so that ROS-nodes could contact specifically any of them. We decided to build a unique name for the controller thanks to the robot name, the process ID of the Webots instance and the hostname of the computer it is running on. With this

we make sure that the name is both unique to the robot and that the user can directly recognize which robot it corresponds to. This unique name is putted at the beginning of the name of each topic or service so that other nodes immediately know with which robot they are talking.

In order to connect to a ROS network, the node must know the address to contact it. The ROS framework use an environment variable called ROS_MASTER_URI to designate the address of the network. There is 3 way to set this variable for the ros controller : you can simply define it in your environment before starting Webots, you can set its value in the controller arguments or in a runtime.ini file.
As any robot controller in Webots, the ros controller will also manage the time step of the robot and detect the end of simulation. The class will also detect if the ROS network can be reached and is working correctly.

This is the core of the ros controller and I will show in the following paragraphs with more details the constructor and the main function of the Ros class.

Here is the pseudo-code of the constructor of the Ros class :

```
Ros::Ros(int argc, char **argv){

  detect the type of Robot;
  call the corresponding constructor;

  initialize some variables of the class;

  get PPID of the controller;
  get hostname of the computer;
  add both of them to the name of the robot to get the controller's unique name;

  get the ROS_MASTER_URI environment variable;
  check if the ROS network can be reached;
  if it can be reached, connect to the ROS network with its unique name;

  publish the robot's name;

  advertise all of the robot's services;

  if Robot is supervisor: new RosSupervisor;
  if Robot is DifferentialWheels: new RosDifferentialWheels;

  for each device of the robot:
    test which type of device it is;
    create the corresponding Ros object; // and the services that comes along
    add the object to the list of RosDevices;
    if it is a sensor: also add the object to the list of RosSensors;
  also do the same for the batterySensor;

}
```

When a device or the robot advertise a service during this constructor, it also associate a callback function to it. This function is automatically called when a request is sent to the service so the controller doesn't have to manage the requests and the services and only focus on running the simulation and publish on the different topics on the main(*run()*) function :

```
Ros::run() {

  init tic,toc variables;

  infite loop until Webots stop the simulation or the network hang out:
    check if ROS network is alive;
    refresh the ROS-node to get request and published value;

    for each sensor in RosSensors list:
      publish needed values;

    if a ROS-node uses the time step service:
      measure actual time;
      if no ROS-node has called a new time step:
        call a null time_step to check if Webots wnats to end simulation;

    else:
      call time step of 32 ms;

}
```

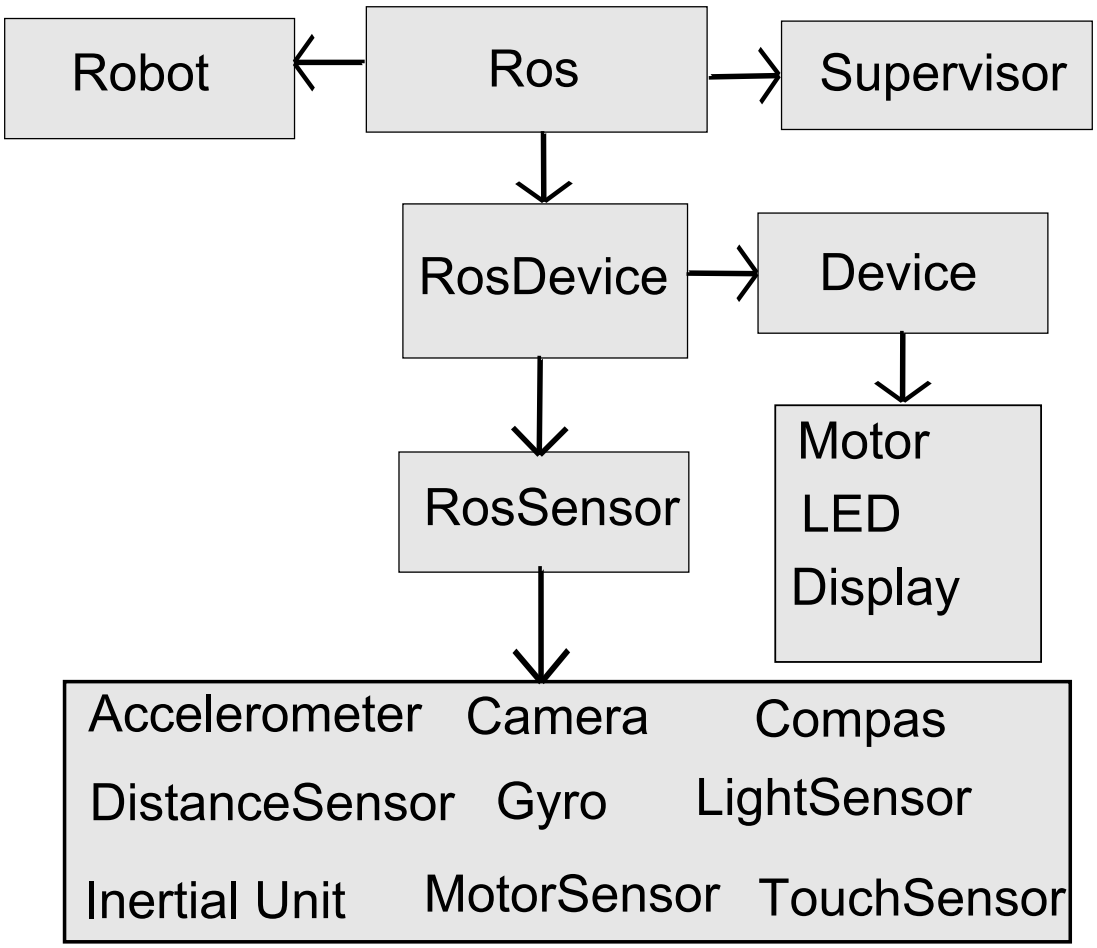


FIGURE 1 – Controller ClassChart. The main class Ros contain the Webots robot or supervisor as well as a list of RosDevice, each one containing a Webots Device

2.4.6 Linux version to Cross-platform version

The biggest challenge I encountered during the development of the ros controller has been to make it work on every platform. Since Webots is available on the three platforms - Linux, Mac OS X and Windows - the new ros controller also have to be available on all of them. Since the controller is written in C++, it uses the C++ version of ROS and the Webots controller.
Unfortunately, the ROS framework was developed on the Linux platform and the official version is only released on it. Other versions exist on Mac OS X and Windows but are only in beta state for now. Since it has been designed to be used on all platform, the c++ code of ROS already contains cross-platform code. The main issue came from the third-party dependencies.

ROS is heavily based on the Boost libraries. Boost uses the libstdc++ library which was the standard for several years and is now slowly replaced by libc++. While the gcc compiler still uses libstdc++ by default, the last clang compiler used on Mac uses the new version. As a temporary solution until boost switch to the new standard, we compiled two version of the Webots c++ controller : one with the libstdc++ library and one with the libc++. This way we can switch between both of them with a simple flag in the controller's Makefile. The version used by ros groovy is 1.46 but the last release of boost is 1.55. The last version still uses libstdc++ but has few bugfix useful and is backward compatible with older version. We uses this version on every platform with only a minor modification on Windows to be able to compile with MinGW instead of Visual Studio.
The other dependency is on log4cxx, a logging API from the Apache company. It is also a free software, but it has really less maintainers. While it is quite easy to compile on unix platforms, it is much harder to compile on Windows with gcc. Log4cxx has dependencies on libapr and libapr-utils. The 3 of them have no pre-compiled version for MinGW on Windows and it has to be recompiled from the sources. Most of the compilation setup was not up-to-date and many fixes had to be done to be able to compile them with the version of gcc used by MinGW. There was also some problems of compatibility between them because each version of the libraries was not backward compatible and I had to find the good set of version to make them work together.

The other big issue I had to solve has been the compilation of ROS on Windows. A beta version exists but it require Visual Studio to work. The Visual Studio versions used on Windows 8(VC2012) and on Windows 7(VC2010) are not compatible and Webots doesn't want to have a dependency on this API.
Therefore I had to recompile it from the sources. The compilation uses CMake to generates the libraries instead of standard Makefiles. The different files to configure compilation are mostly generated and extremely hard to read or even edit correctly. The easiest solution has been to create a Makefile and reconfigure the compilation from scratch. ROS has many over dependencies on Windows other than the ones cited previously and we choose to only recompile the very minimal part of ROS needed for the controller, dropping all support to create ROS-node, generate a master, or use non-standard message types.

Finally, after extracting the correct rules and files from the CMake configuration, I managed to reduce down to 5 libraries for ROS : rosconsole, roscpp, roscpp_serialization, rostime and xmlrpcpp.

2.4.7 Runtime.ini file

While developing the cross-platform part of the controller, we experimented it on many different computers. One of the annoying thing was to redefine the ROS_MASTER_URI environment variable every time. In order to address this point and to solve other similar issue in Webots, I implemented a file named runtime.ini. this file can be used in the directory of any controller of Webots and it will be read just before starting the controller. Inside, the user can define environment variable for different platforms following this example :

```
[environment variables for Windows]
NAOQI_LIBRARY_FOLDER="bin;C:\Program Files\Naoqi"

[environment variables for Mac OS X]
NAOQI_LIBRARY_FOLDER=lib

[environment variables for Linux]
NAOQI_LIBRARY_FOLDER=lib

[environment variables for Linux 32]

[environment variables for Linux 64]

[environment variables]

ROS_MASTER_URI = http://localhost:11311

[environment variables with relative UNIX paths]

NAOQI_DIR  = $(WEBOTS_HOME)/resources/projects/robots/nao/aldebaran/naoqi-runtime
NAOSIM_DIR = $(WEBOTS_HOME)/resources/projects/robots/nao/aldebaran/simulator-sdk
CLASSPATH = ../lib/java:myjar.jar:$(CLASSPATH)
PYTHONPATH = lib/my_python_classes:../lib/other_python_classes:$(PYTHONPATH)

WEBOTS_LIBRARY_PATH = $(WEBOTS_LIBRARY_PATH):$(NAOQI_DIR)/$(NAOQI_LIBRARY_FOLDER)
```

With this feature, we could solve a runtime linking problem for the ros controller and modifying the ROS_MASTER_URI became much more easier.

2.4.8 Python script to generate header files for services

As we said in the objectives in mind while developing the ros controler, we want to avoid as much as possible third-party dependencies. There is a lot of services defined for the ros controller(over a hundred). Normally services are shortly defined in a .srv file and ros build the service header files along with the packages according to this file. We wanted to not depend on the compilation system of ROS framework for our ros controller so we had to come up with a solution for all these services. I studied deep on how ROS generate these files and came with a really short solution.

I built a python script that use the description of the service in the .srv file along with three template header files to generate the headers for this service. ROS needs three files : one for the service, one for the request message and one for the response.

The script will parse the .srv file and then create a copy of the template files. In each of them, it will looks for keywords and replace them according to the type and name of the fields of the service. It took several iterations to get a version with no issue because ROS have sometimes unique ways to declare some types of message but in the end there was no bug left and the solution was really scalable.

This script allowed us to reduce a lot the amount of code hand-written and therefore avoid human mistakes. On top of it, it also makes creating new services for future API functions in Webots much more easier.

3 URDF parser

The second part of this project have been to design a tool that could transform a robot described in a URDF file into a simulated model inside Webots. Before rushing into the development it is very important to get a clear picture of how robots are described inside URDF file, which properties we can find inside and which norms are followed and do the same reflection for the models described in VRML for Webots.

3.1 Unified Robot Description Format

The URDF file format is an XML format for representing robot model. It allows to describes the mechanical parts of a robot and all their specifications. It can be used on most types of robot but has some limitations such as the parallel robots for which loops cannot be represented. The technical specifications of the robot covered in URDF files are :
– Kinematic and dynamic description
– Visual representation
– Collision model
Each node and sub-node inside the model always have at least these two properties : a name and a pose(position and orientation).
Robots in URDF are described by using only two different types of elements : **links** and **joints**.

3.1.1 Links

Links represents physical bodies of the robot. It can have 3 types of child node :

– The inertial node contains the physical properties of the solid, i.e. the mass of the solid in kg and its inertial matrix along the x, y and z axis.

– The collision node simply contains a geometry of the collision body of the solid. It is generally more simple than the real solid because collision computation between two bodies is really expensive.

– The visual node also contains a geometry like the collision node, but it is generally as accurate as possible as it only serves for the visual rendering and doesn't affect the physic properties of the robot. It also contains a material element with color and/or texture information.

Links are generally a whole physical part of a robot between two motors or transmissions but sometimes they are too complex and need to be split in different links with more simple geometries.

3.1.2 Joints

Joints are the part between two links. They usually represent motors and actuators of the real robot. They can be either of the following type :
– revolute : a hinge joint that rotate around an axis with a minimal and maximal position.
– continuous : also a hinge joint but it has no lower or upper limit.
– prismatic : a sliding joint that translate alone an axis. It has a limited range fixed by lower and upper maximal position.
– fixed : this a special case as it has no degree of freedom. It is a convenient way to link two parts of a solid that has been split in two links but that actually can't move one from another.
– floating : this joint allows all 6 degrees of freedom but can impose some position or dynamic limits.
– planar : This joint has two degrees of freedom allowing motion in a plane defined by its orthogonal vector.

Joints always have two children nodes called parent and child. they contain the name of the links that are linked by this joint. The coordinate system of the child is determined by a transformation given in the origin child applied to the coordinate system of the parent. If no transformation is given the child link will simply have the same coordinate system as his parent.
Most types of joint require an axis to describe the direction(s) of the motion. It is also a child node of the joint given b< the x,y,z coordinates in the parent's coordinate system.
Joints can also have other optional child-nodes that provides mechanical information such as damping and friction of the joint, physical limits or soft limits for the controller.

3.1.3 Geometry

Simple geometries in URDF models can be described if they are either a box, a cylinder or a sphere.
For more complex geometries, it is too complex to describe them in a simple way inside an xml file. therefore the geometry node can also contains a mesh with the name of a trimesh file. This file can either be a Collada (.dae) file or a stereolithography file (.stl). Collada files are preferred as they offers much more information than just the set of points composing the geometry. They can contains texture, colors, animation and other FX effects.

3.1.4 Plug-ins

The URDF file fromat is enough to describes the core of a robot, but it lacks many information that can be useful for simulation or calibration. Some other nodes are used in URDF files but there is no official template for them and are generally specific to a certain robot, software or application. The most common one is the gazebo node, used to give additional information to the gazebo simulator when he reads an URDF robot file.

3.2 Virtual Reality Modeling Language

The VRML is a standard file format for representing 3D interactive vector graphics. It is widely used for virtual reality world and for web pages animations and is the language used by Webots for its worlds and robots.
VRML is more oriented towards simulation of virtual worlds and wasn't designed specifically for robotic purpose. Therefore it doesn't have anything to represents motors and actuators at all but has many features for visual rendering and world characteristics. Cyberbotics made some extensions and arrangements to compensate the missing elements of VRML.

3.2.1 World

VRML can describe a whole virtual world and all it's properties such as gravity, light, ground and surroundings or collision rules. Although any VRML file can contains these information, only .wbt files use them and they are just ignored in other files.

3.2.2 Solid

Solids are VRML nodes used for any physical object in the virtual world. They can contain physical properties such as mass and inertia matrix as well as a BoundingObject with a geometry used for collisions. It can contain any number of children which can be other solids and various shape for visual rendering. Solids can have an initial motion but they won't move on their own and only changes from reactions with other element of the simulation.

3.2.3 Robots

Robot nodes are entirely new from VRML. Robots have a solid body, but they also have a set of electronic devices including, battery, many different sensors and motors. It also have a controller program that can access data from the sensors and drives motors.

3.3 URDF to VRML

After looking at the description of both file format, there clearly is many differences between both of them and they simply cannot be entirely transformed into each other. Therefore we had to do some compromises and drop certain parts and arbitrary choose some values.
The link nodes from URDf file were easy to translate into solids in VRML as they share almost all their elements. The joints were much harder to translate.
The revolute, continuous and prismatic types of joint also exist in VRML's Webots format but the floating and planar types have no equivalent in VRML. We had to drop support for these two types until some similar motor is created in Webots. The last type, fixed, isn't translate into a new node in VRML, but the child link is directly appended to its parents with a transform node between them if they don't share the same coordinate system. This comes from the different design between the two file format. URDF always has a joint between two links while VRML allows a solid to have other solids as its children. The calibration element as well as part of the safety_controller one neither have a counterpart in VRML and have been dropped too.
The lost of information about the calibration or safety_controller isn't a big trouble have they don't exist in a virtual world and they aren't fundamental to the robot. On the opposite, the joints that can't be transformed in VRML are more problematic. They can't be simply ignored because it will break the structure of the robot and the robot will loose part of his body or his mobility. It has to be fixed by the user manually when it's possible.
The basic geometries from URDF have their exact equivalent in VRML but meshes are just url to a file. Geometries in both STL and Collada files are triangulated surfaces and they can be translated into IndexedFaceSet.

3.3.1 STL trimesh file

STL files are basic binary files that only contain triangulated geometry. They start with a header of 80 characters followed by the number of triangles. The rest of the file is purely data describing each triangle one by one : the normal vector, the 3 vertex and an attribute byte.

3.3.2 Collada file

The Collada files also contain triangulated geometry like STL but they can be much more complex. They can contain not only one geometry but an unlimited amount. They can also contains colors, textures, FX effects, spatial transformations and many more. As it can be really tough to retrieve information, the Khronos Group that made the Collada format also propose a python module called pycollada that help a lot to move across the different nodes of these xml files.

3.3.3 Script

The script to transform the URDF model into Webots VRML proto is composed of three Python files.

The main script is really short. It open the file given as argument with an xml parser model and check if the file is a valid URDF model. If it is, it will call the second script with the opened file as argument.

The second file is the URDF parser. A set of class is defined inside corresponding more or less to the possible nodes we can find in an URDF file. The script will go over the URDF file and store every link and joint into object of the corresponding class and do the same for each children node. If it finds a TIFF texture file, it will convert it to a PNG. If it finds a mesh file for a geometry, it will extract the triangles of the STL or Collada geometry and additional visual rendering information if it is a Collada file. It also have functions to retrieve relationship between links.

Once the URDF file has been parse, the main script will build the parenting tree for the link and find the root link that is the parent of all other links in the model. It will then call the third script to initialize the PROTO file and give the root link to this last script along with the list of the parent-child pairs.

The third script is the one that write the output VRML for Webots. It contain functions to write a Solid from a link object, a joint from a joint object, BoundingObject from a collision geometry and a Shape from a visual object. As the VRML is written following the parenting tree, the script will start by writting from the root link and then go to the joints that have this root in their parent field and follow with the link in the child field. It will bounce back and forth between the tow functions following the parenting tree until reach the last child and simply close the proto as it will have write every node of the URDF and it will be complete.

4 Results

Once the development of the code had reached a satisfying level, I created some examples and tests to check if every feature is working correctly and have good performances. The different results and observations resulting are presented in the following section.

4.1 Objectives Reached

The new ros controller is a Webots controller able to work on any robot available in Webots and on any platform. It is able to connect to a ros network and communicate with other nodes connected to it. It offers the same functionalities as any other c++ controller in Webots. Multiple instances of the ros controller can be running at the same time and each one will be unique and provides only the functions available on the robot it is running on. Although the code for this controller contains more than a hundred files, it has been reduced to its limits and has the minimal set of dependencies in order to make it easy to maintain in the future. As users can be totally new to Webots, or robotic programming, the design has aimed to make the controller as simple as possible to use while still being scalable to complex simulations across multiple machines or various types of robot.

The robot file translating script is able to transform an URDF file describing a robot into a VRML file that can be directly used in Webots. The script can retrieve all the information contained in the URDF file. It can then use it to create a robot file for Webots (.proto) that contain the same body as the original. All the joints that have an equivalent in Webots are reported in the proto robot, but only with the specification that have a meaning in the simulated world. The script can also read trimesh files(stl and collada) linked in the URDF to get traingulated geometries and related information for collision and visual bodies.

4.2 Limitations

4.2.1 ros controller

The ros controller is really exhaustive and can be used in many different situations but after running experiments, some problems and drawbacks appeared.

First, the controller can't wait for a master to be connected : when the controller start it will init the ros part with the given parameter and then connect to the server by addressing its master. ROS provides tools to check if the master can be reached before connecting to it but they fall into a freezing loop if you call them several times in a row. The controller will fall in the same freezing state if we try to directly connect to a non-reachable server. The solution has been to only check once if the master is alive and simply exit the controller if it is not. It is not the best behaviour but it avoid blocking other process of Webots and allow to stop the controller in a clean way.

When running different examples, I tried to run the controller at higher speed than real-time to see how it performs. Experiments have been done on the same machine for each world with the ROS-nodes and the server running on one computer and Webots running on another one. Speeds expressed in the following table are given as a factor of the real-time speed :

world	speed with a ROS-node using the controller	speed with the ros controller idle
keyboard_teleop	x3.6	x17.5
catch_the_bird	x2.8	x14
e-puck_line	x9	x18

TABLE 1 – Max simulation speed with and without ROS controller running operations and sending message

Performances can vary a lot between worlds, but there is no significant difference if the user has to send large data message like camera images(catch_the_bird world) or a large number of message at each time step(e-puck_line world). The differences we can observes mostly come from the computation and functions called in the ROS-node to treat data or set new commands at each time step. There may be differences depending on the platform as some libraries are not the same but it is really hard to observe because the hardware used has a very important impact.

The last limit of the ros controller comes from its design. The controller is supposed to be able to answer all needs of the user. Yet, if he can't do some actions or obtain the behaviour he is looking for, the user won't be able to easily modify the controller. If ever this situation appears in the future, we kept the old ros controller inside Webots files. While it have almost nothing already prepared inside, it makes it easy to shape for a special application.

4.2.2 URDF to VRML script

The translating script is able to parse well URDF files into robots in Webots but it also have a few issues that couldn't be fixed during the project.
As there is several differences between URDF specifications and Webots VRML specifications, some joints cannot be translated into the simulator and robots that contain these joint will be broken as the link between two parts cannot be represented. This can be a big problem and it cannot be easily solved until new big features are added to Webots to complete the actual set of motors available. Hopefully the joints that cannot be translated are really rarely used in robotics.
The script is not perfect and can have some failure : Webots accepts either PNG or JPEG files for texture but not TIFF. The script use a module to transform the TIFF files into PNG but TIFF files are not all encoded identically and the module sometimes can't read them.
When developping the script, I used the model of the Atlas robot because it uses complex collada geometries. I was able to handle multiple geometries in a same file, but some solids ended with a wrong position or rotation :

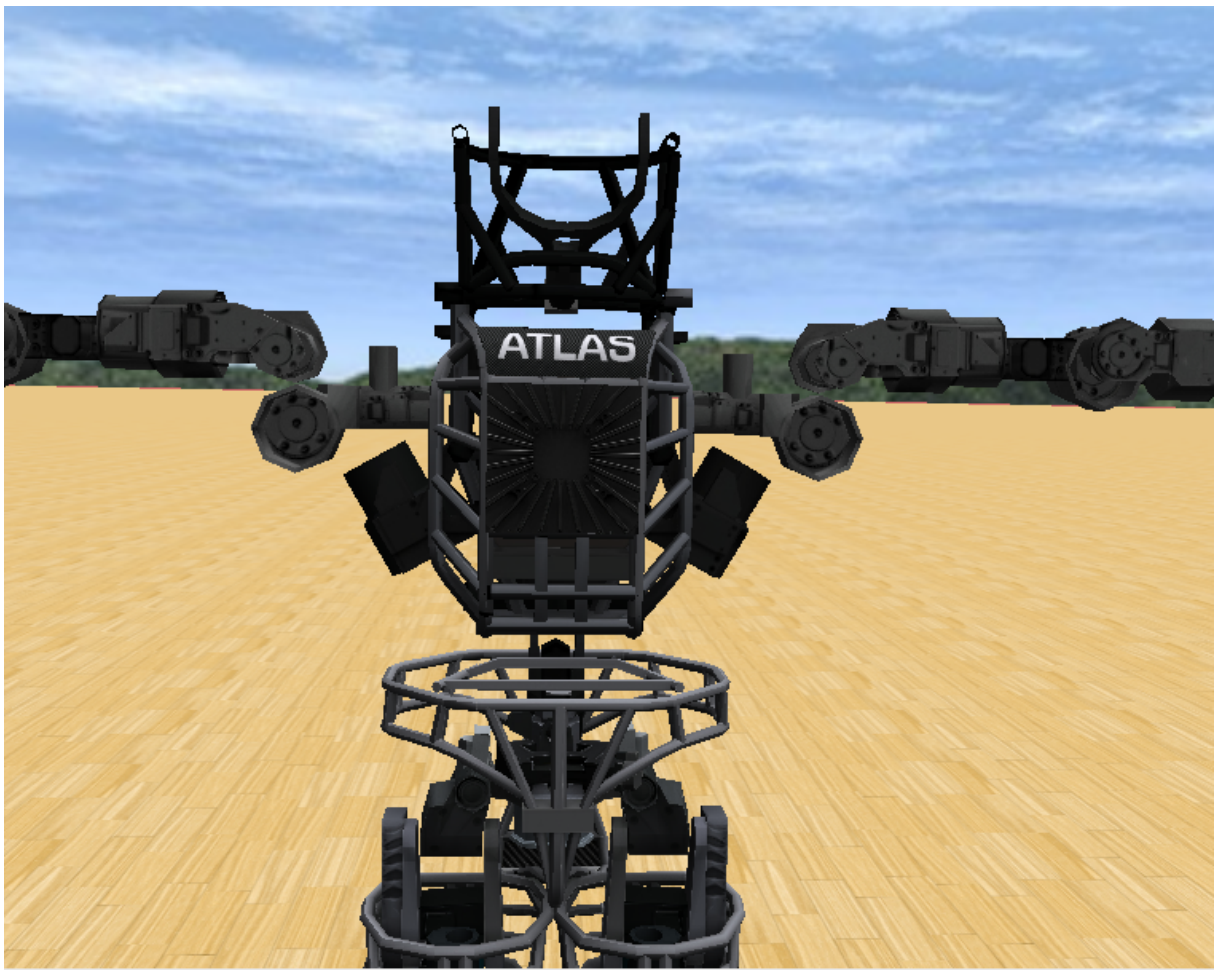


FIGURE 2 – Atlas robot with shoulders with wrong orientation

The Collada file for the shoulders of the Atlas have an identity matrix for the spatial transformation so the error doesn't comes from the parsing of the coordinates of the geometry. I couldn't identify from where does the error come from. It may be an error in the file available on the urdf database or a hidden feature not described in Collada documentation.

Finally, the robots generated have a really good shape and an accurate collision body but the complexity of the BoundingObject requires a lot of computation during collisions. To observe the slow it generates, I compared a version of the PR2 robot with a very accurate collision body based on trimesh and another version with a more approximated collision body based on simple geometries during collisions :

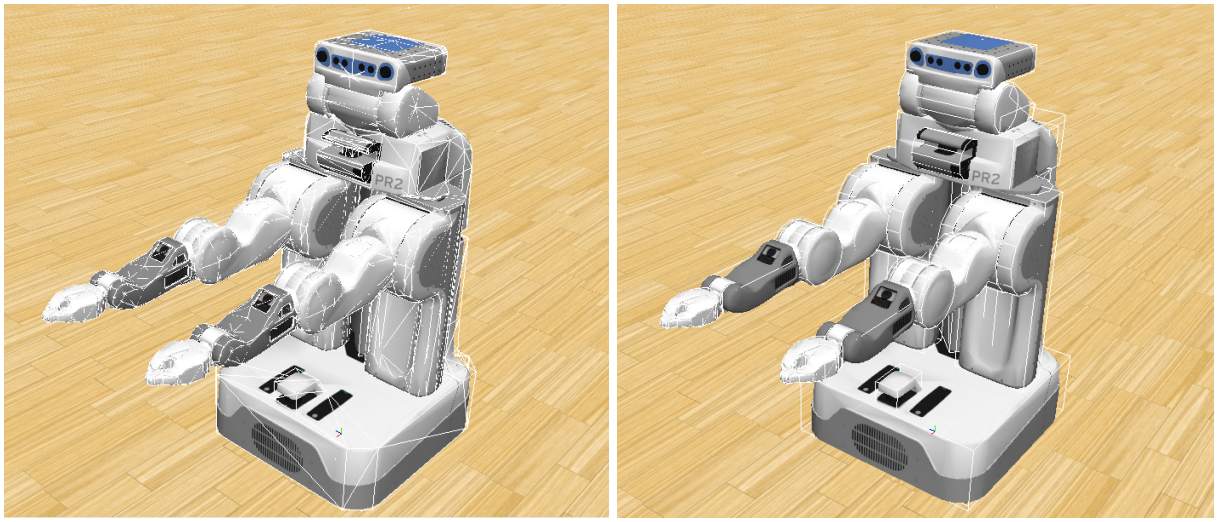


FIGURE 3 – PR2 with different collision body. The BoundingObjects are represented by white edge around each face of the solid. On the left, the accurate collision body. On the right, the approximated collision body

model of the PR2	**simulation speed with standard parameters**	**simulation speed with specific parameters**
complex collision body	x0.12	x1.90
simple collision body	x0.22	x2.01

TABLE 2 – Impact of the complexity of the collision body on simulation computation speed

The PR2 is very heavy and have great inertial moment and the default physical parameters used by Webots to handle collisions and cinematic leads to very bad performances. The simplified body helps a lot, here it double the speed, but this is still not enough. I had to tune the different parameters manually to optimize the simulation for the robot and finally get good results where the complexity of the collision body didn't much impact anymore.
URDF file format try to be as accurate as possible but for simulation purpose having the most accurate model is not always the best solution. In order to improve this drawback of the script, a good solution would be to introduced some options that could be given as arguments to the script. With these options, we could decide to approximate the geometries by box or cylinder proportional to the size of the triangulated geometry. These options could also allow the tuning of some parameter of the joints by a coefficient, enable/disable internal collision or simply set the initial position and rotation of the robot.

4.3 Examples

In order to illustrate my work and to test the different features I implemented, I developed several examples that will be available in Webots.
For the ros controller I made 5 worlds in Webots :

- e-puck_line.wbt : An e-puck uses its infra-red sensors to detect a black line and the ground and follow it. It can also use these sensors to detect and avoid obstacle.
- catch_the_bird : The robot controls a camera that can rotate around the vertical axis. It will process the image to detect if it can see an object and take a picture of it
- keyboard_teleop : A Pioneer 2 can be moved inside an arena containing obstacle by using the arrow keys.
- panoramic_view_recorder : A Nao robot stand still on a ground. A supervisor robot record a video and change the viewpoint and the light in order to rotate 360 ° around the Nao.
- complete_test : This is not a realistic world. It contains a robot that is a supervisor and have one copy of each kind of device and a few basic solids. Any function of the ros controller can be called and tested on this robot except the differential wheels device.

Each world uses the ros controller and also have a ros node with the same name as the world. The nodes for the examples needs to be compiled with ros framework like any other ros node and doesn't need to have Webots installed for that. They will subscribe to the ros network and connect to a ros controller running in an instance of Webots. The different nodes aim at providing to the users different classic applications of the ros controller.

The first world I worked on is the e-puck_line. The e-puck is widely used in Webots and it speaks to a lot to people. The code used for this example is highly similar to the original version shown in the guided tour of Webots. The node will connect to the robot, activate all the sensors and start a finite-state machine. The node will receive the value of the sensors trhough ros messages, detect obstacle and/or the black line on the ground a compute the command to the two wheels on the ROS network to the e-puck and then call the next step of the simulation. On top of showing a very simple application, it also shows how to translate a previous code written for a c++ controller in Webots into a ros controller that can be run on a remote computer.

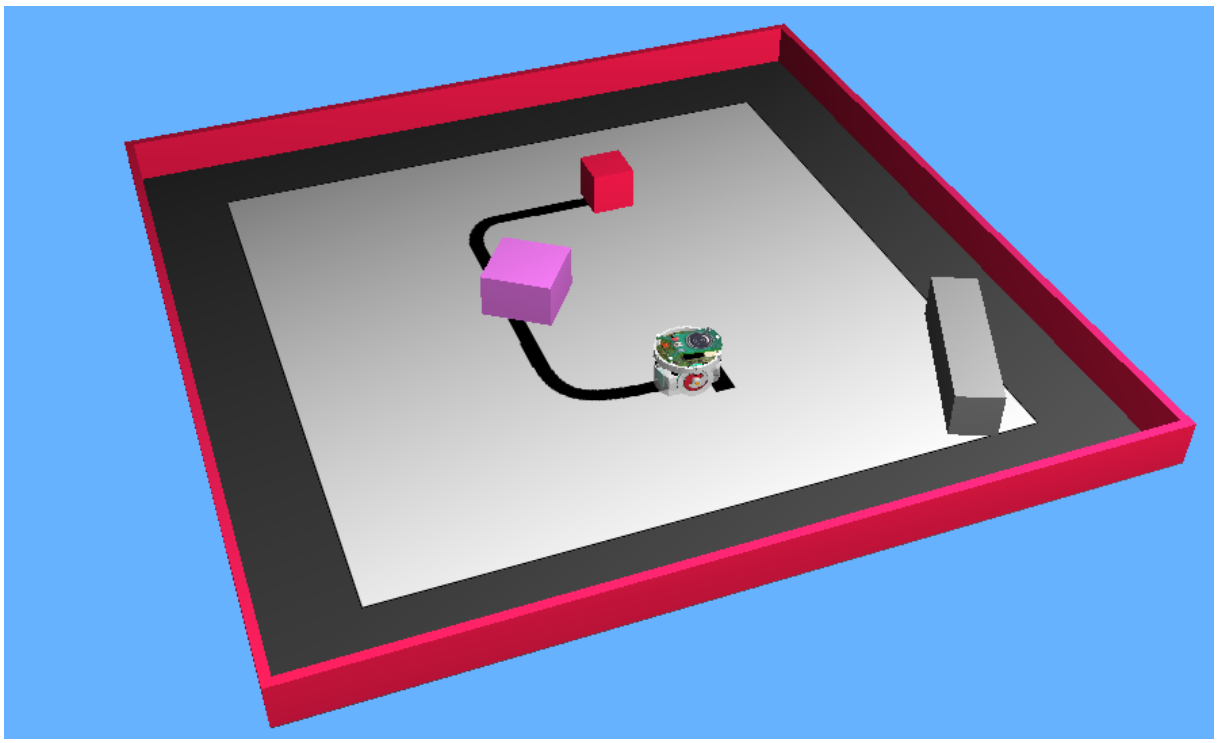


FIGURE 4 – e-puck_line world. The e-puck is on a black line inside an arena. A pink box is blocking the path to the other extremity of the line and the e-puck will have to avoid the obstacle to reach the end.

The world catch_the_bird contains a turret on which a rotating cylinder is mounted. The cylinder can rotate over 359° and contains a camera. The world also contains a bird standing still on a pole. The node will rotate the camera both ways and read the image. If it finds something unusual inside, it will stop the turret and save the image into a PNG file and print a message to state it has find something. The file will be saved in a path relative to the controller directory instead of the directory of the node as it could be on an another computer where Webots can't save the file. The world is very simple but it uses large data messages with the camera and a very easy way to process it.

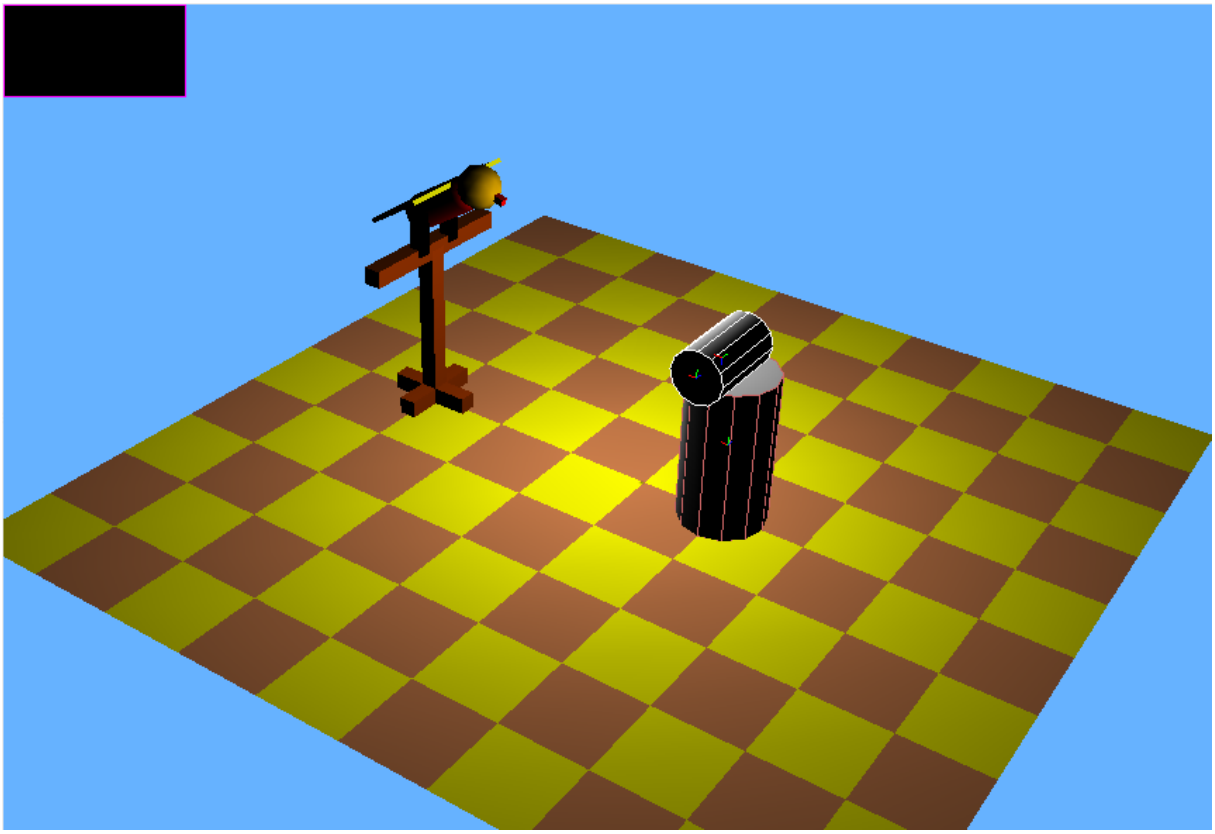


FIGURE 5 – catch_the_bird world. The cylinder with white lines contains the caemera and can rotate around the axis of the bigger cylinder. The yellow bird is standing on its wooden pole

Keyboard_teleop world have a Pioneer 2 in the center of a square arena of 4x4m. the arena is quite small and congested with a few box obstacles. The node enable the keyboard in the window of the simulated world and read the keys pressed on the keyboard. the user can use the arrow keys to drive the robot around the arena and try to move it without hitting any wall or object. With this world, the user can see there is no noticeable delay between the moment the key is pressed and the moment the command of the robot is updated. It also commands the wheels very differently from the e-puck : the robot have two RotationalMotor instead of the DifferentialWheels of the e-puck and it is control in position instead of speed.

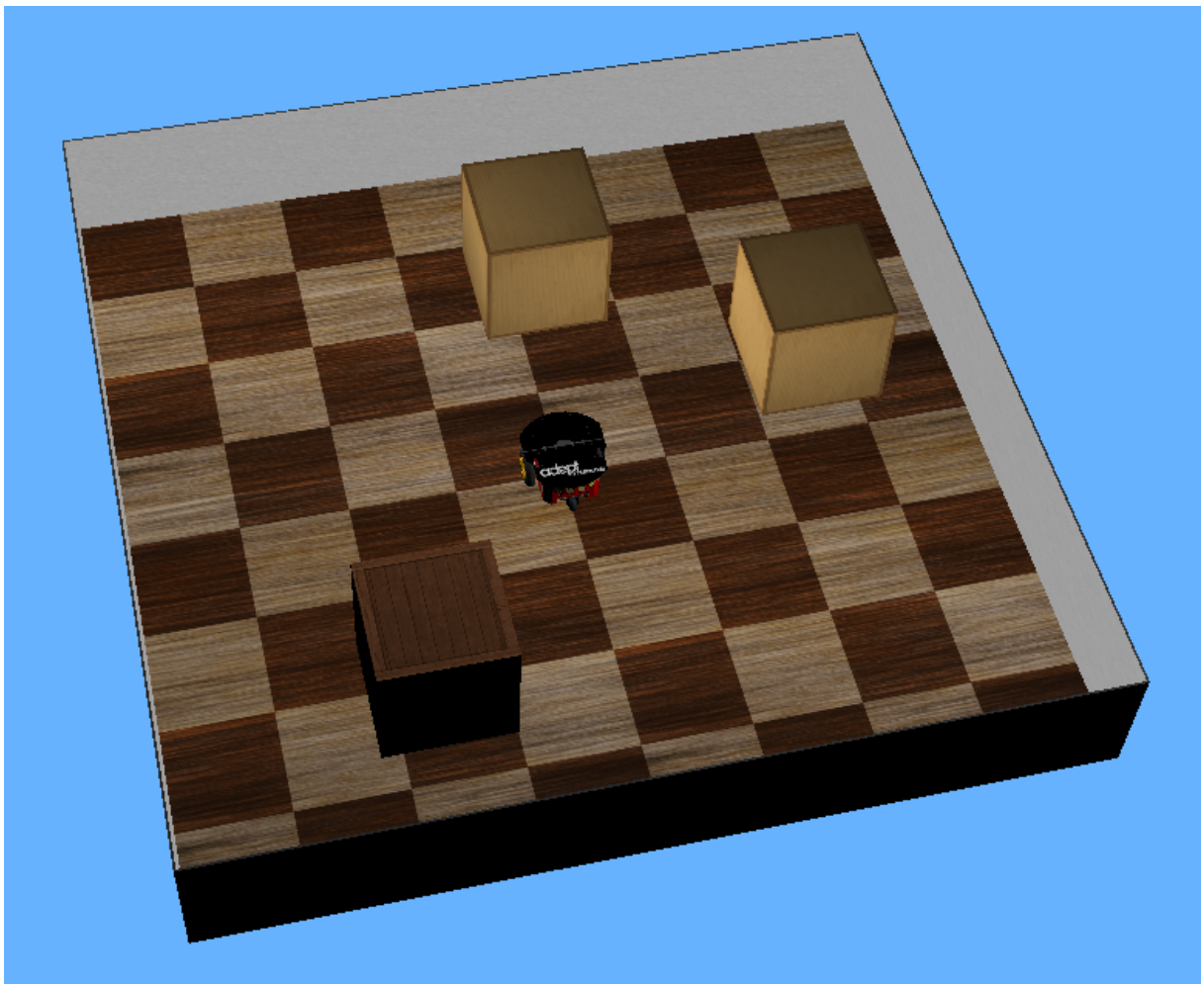


FIGURE 6 – keyboard_teleop world. The arena contains the Pioneer robot and 3 obstacles. There is just enough place for the robot to go between the boxes and walls

The panoramic_view_recorder world has a beautiful orange nao standing still on the ground. The ros controller is not running on the nao but on a robot with supervisor rights. It will record a movie of the view of the simulated world and edit the viewpoint and the spotlight to show the Nao every angle over 360 ° with a bright light. The goal of this example is to show how to use some functions of a Supervisor. As it can edit the world and other robots fields, it must be used with care by more advanced users.

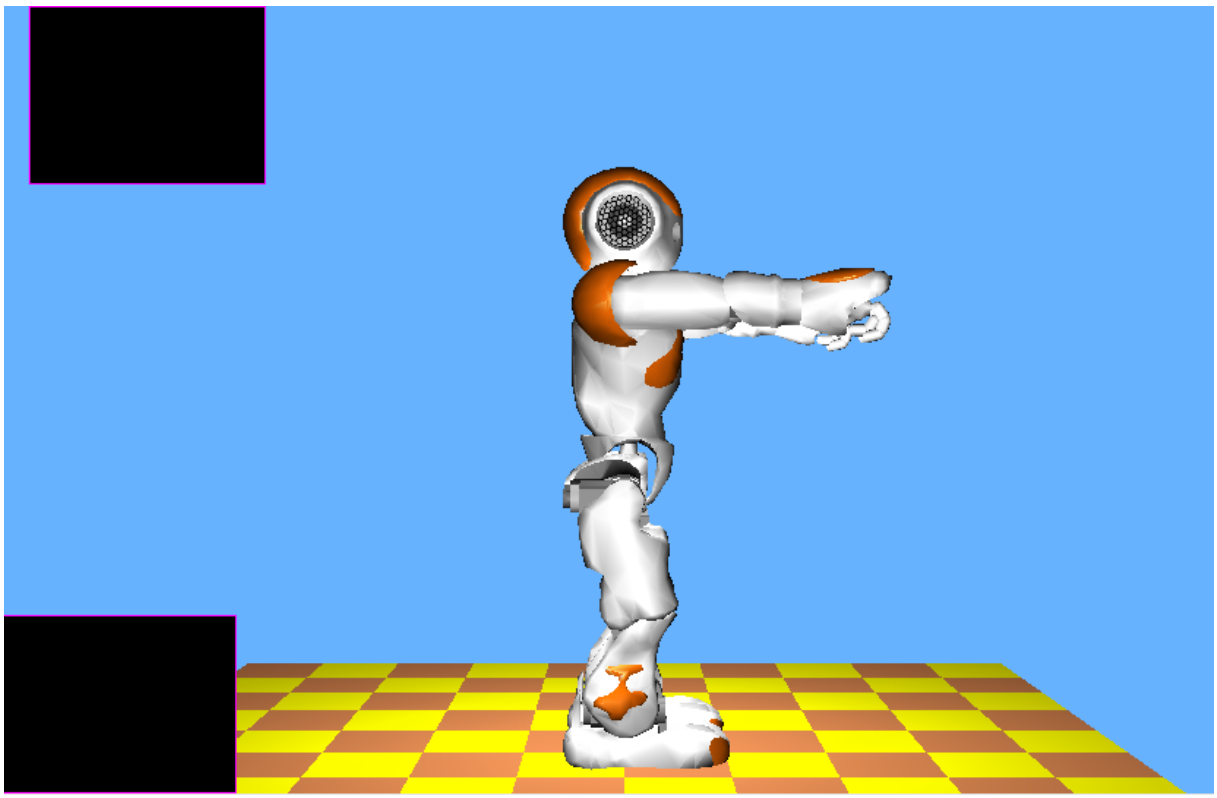


FIGURE 7 – panoramice_view_recoder world. The point of view is at half the height of the Nao. The spotLight has the same coordinates to avoid any shadow while recording the movie.

The last world is the complete_test one. This world isn't an possible application. It is build with a supervisor robot on which one copy of each device has been placed and some objects for collision and imaging purpose. the main goal of this node is to check that the controller has nothing broken by testing every different service and topic of the ros controller one by one. The only functions that aren't tested are the functions of a DifferentialWheels robot, but there aren't much of them and they are already tested in the e-puck_line world. this example can also serves as a database : when the user is not sure how to use a certain function, he can look at its implementation in this node.

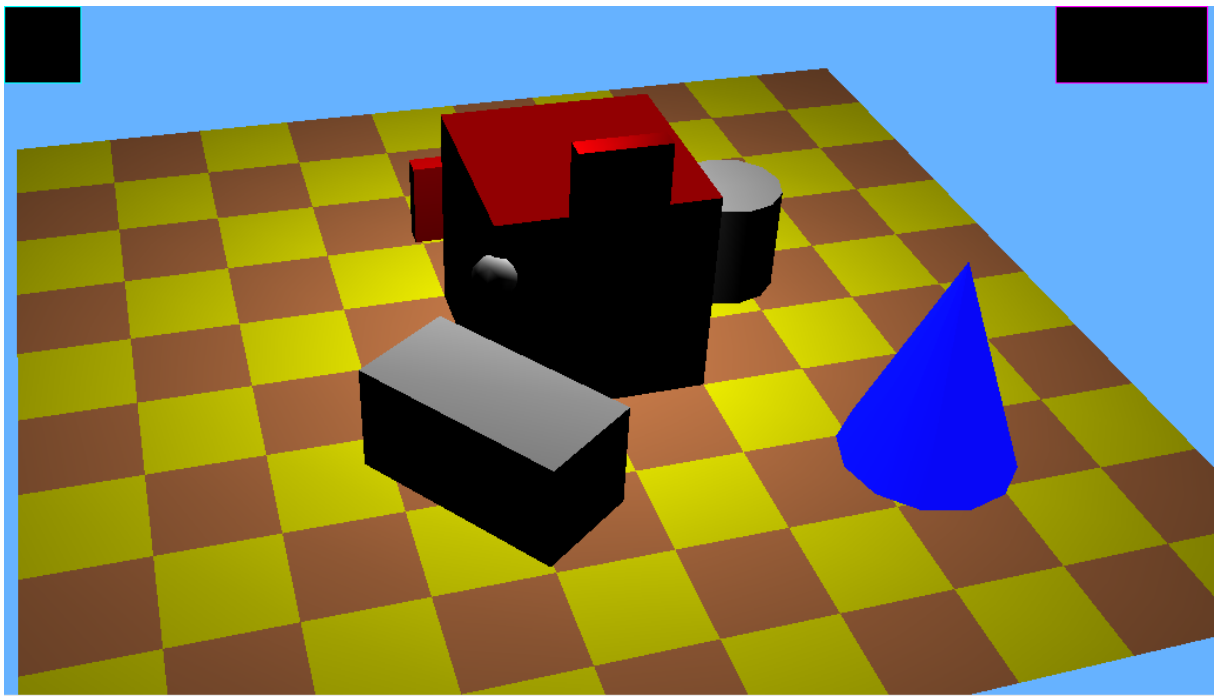


FIGURE 8 – complete_test world. The robot is the big red box in the center. We can see an arm in the front linked by a rotationalMotor while the arm on the back is linked by a linearMotor

There is a last example available in the directory called robot_information_parser. It is very short and doesn't have any world associated with because it can be used with any robot that has a ros controller. This node is a sort of tutorial for people starting with ros nodes and Webots. It contains the very core that should be in any ros node that connects to the ros controller and only print the list of the name of the devices existing on the robot. It has a lot of commentaries to guide the user step by step inside the code.

When working on the URDF translating script, I worked mainly on two existing robots used in research : the PR2 and the Atlas.

The PR2 is a robot developed by Willow Garage. It is an heavy humanoid robot with a large wheeled base. It weights over 300kg and has a really complex structure with a lot of sensors. It is quite exhaustive as it uses basic and triangulated geometries, textures, rotational and linear motors, lasers and cameras. The Solid geometries have been perfectly retrieves as well as the joints and only a few textures are missing due to a bad format in the TIFF files. The huge weight of the robot has been really hard to handle for Webots. I had to tune many parameters in the world to avoid buggy collisions and make all motors to behave correctly. The PR2 also lies on 8 wheels and leading to many mechanical loops that slow down the simulation a lot.

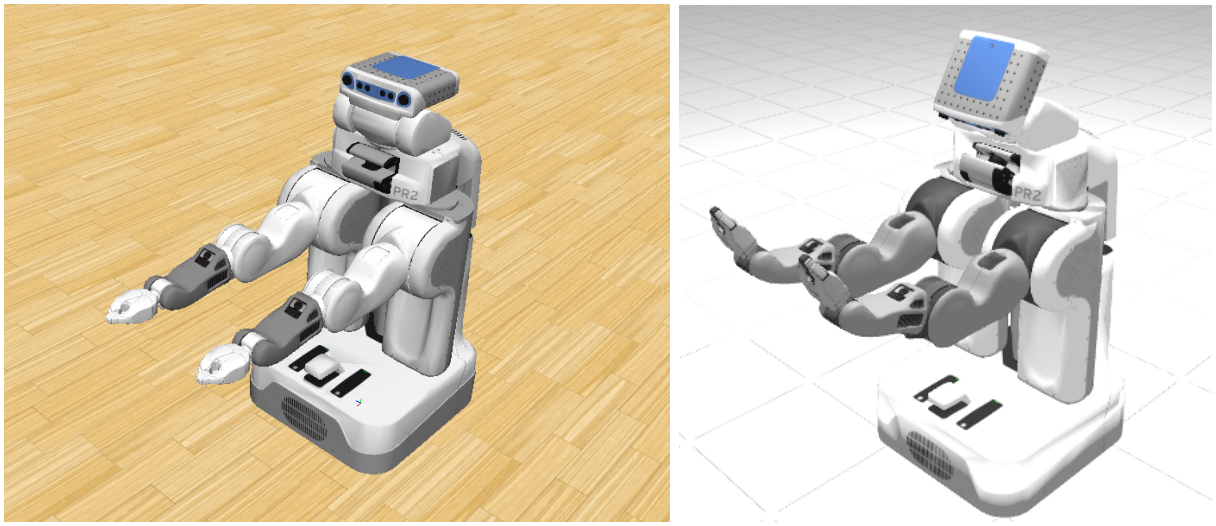


FIGURE 9 – Comparison between the PR2 model of the Gazebo simulator and the model produced by the script on the URDF file

The atlas robot has been developed by the Boston Dynamics group and is being used for projects of the DARPA challenges. It is a humanoid robot of human size with a wire frame skeleton. The tree of its links and joints is quite simple but all the solids described in the URDF file uses trimesh files. It uses STL files for the collision bodies and Collada files for the visual shapes.

This model example is really interesting for its Collada files. They contains any type of data that could be interesting to get into a model in Webots. It contains triangulated geometries, all kind of geometric transformations, textures, multiple geometries in the same file, etc. It has been a bit of a challenge to transform the URDF file into a well built proto for Webots.

While the result looks nice it is still not perfect and have some issue. One of the most annoying one is that it doesn't have a head. The head link is mentioned but there is no description of it inside the URDF (see Figure 2). The output Atlas also have some issue on geometric transformations and visual shapes of the arms are not exactly at the right position. The model in the URDF corresponds to the third version of the atlas while the actual model in Webots is based on the second version. The new version is much more accurate and looks more realistic but with all the triangulated geometries for the BoundingObjects its simulation cannot run as fast as the previous one.

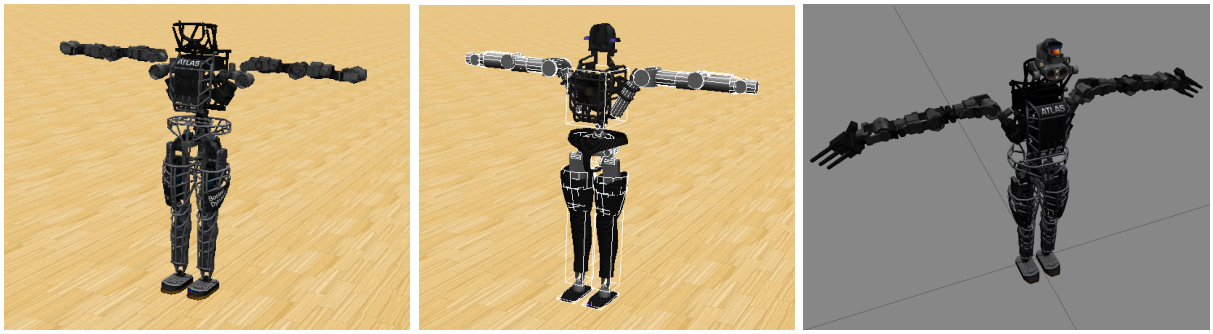


FIGURE 10 – Comparison between the new Atlas on the left, the previous Atlas of Webots in the middle and the model of Gazebo on the right

5 Conclusion

During this Master project at Cyberbotics I have been able to complete the conception of a new controller for robots in the Webots software that can interface with the ROS framework. The controller is really complete as it can run on any kind of simulated robot and works on every platform. It has been optimized as much as possible to be easy to maintain and upgrade in the future and to be user-friendly, even for people not used to program with Webots or ROS. It showed really good performances during the different test we perform and the conception seemed really robust to bug. There may or may not be some issues that we couldn't detect yet but they will only be find once it will be more and more used in Webots community and it got enough feedback. This controller should help a lot of people to simulate their research with Webots.
After developing this controller, I worked on a tool to convert a robot description from URDF files into a complete model of the robot inside Webots. The script has been able to create very accurate models of robots like the PR2 of Willow Garage but the result is not always ready for simulations and some improvement have yet to be done to have a final solution for this tool. This tool had promising results and a future similar tools could be developed to convert other kind of files like the SDF format used by the Gazebo simulator.

This project has also been an incredible experience for me. Working in a company is really different than studies. I learned to work in a team of professional developers and take many things into account to implement my work in a real product. The most unexpected thing I learned is how important the review process is to detect any flow in your work and make sure nothing has been forgotten before releasing your product.
I would like to thanks Olivier for his patience and his really good advices thanks to his clear vision on this project. I also wants to thanks Fabien and Luc for their expertise on critical points of the development.

6 References

– Gazebo documentation : http://gazebosim.org/wiki/Main_Page
– ROS tutorials : http://wiki.ros.org/ROS/Tutorials
– Boost : http://www.boost.org/
– Log4cxx : https://logging.apache.org/log4cxx/
– MinGW : http://www.mingw.org/
– Python tutorials : http://docs.python.org/2/tutorial/
– Specifications for URDF file format : http://wiki.ros.org/urdf/XML/model
– URDF model database : http://wiki.ros.org/urdf/Examples
– VRML specification : http://www.w3.org/MarkUp/VRML/
– Webots documentation : http://www.cyberbotics.com/reference/
– Python Image library : http://www.pythonware.com/products/pil/
– Collada : https://collada.org/
– Xml minidom documentation : http://docs.python.org/2/library/xml.dom.minidom.html
– many many solutions for programming : http://stackoverflow.com/

7 Annexes

7.1 Ros class

Ros.cpp :

```
// You may need to add webots include files such as
// <webots/DistanceSensor.hpp>, <webots/LED.hpp>, etc.
// and/or to add some other includes
#include "Ros.hpp"
#include <webots/Node.hpp>
#include <webots/DifferentialWheels.hpp>
#include <webots/Supervisor.hpp>

#include "RosAccelerometer.hpp"
#include "RosBatterySensor.hpp"
#include "RosCamera.hpp"
#include "RosCompass.hpp"
#include "RosConnector.hpp"
#include "RosDisplay.hpp"
#include "RosDistanceSensor.hpp"
#include "RosDifferentialWheels.hpp"
#include "RosEmitter.hpp"
#include "RosGPS.hpp"
#include "RosGyro.hpp"
#include "RosInertialUnit.hpp"
#include "RosLED.hpp"
#include "RosLightSensor.hpp"
#include "RosMotor.hpp"
#include "RosPen.hpp"
#include "RosPositionSensor.hpp"
#include "RosReceiver.hpp"
#include "RosSensor.hpp"
#include "RosSupervisor.hpp"
#include "RosTouchSensor.hpp"

#include "std_msgs/String.h"
#include <ctime>
#include "ros/master.h"

// IP resolution includes
#ifdef WIN32
#include <TlHelp32.h>
#else
#include <netdb.h>
#include<arpa/inet.h>
#endif

// This controller is really unsusual as it can be used on either a Robot, a Supervisor or a DifferentialWheels.
// Unfortunately, the C++ API has no way to know the type of a robot before creating on instance of it.
// Therefore we need to use the wb_robot_get_type function of the C API that can be called just after a wb_robot_init.
// This way, we get the type of the robot and we can call the right constructor for the Robot instance.
#define WB_NODE_ROBOT                   40
#define WB_NODE_SUPERVISOR              41
#define WB_NODE_DIFFERENTIAL_WHEELS     42
typedef int WbNodeType;
extern "C" {
  int        wb_robot_init();
  WbNodeType wb_robot_get_type();
}
// end of fix for robot's type.

using namespace std;

Ros::Ros(int argc, char **argv)
{
  wb_robot_init();
  WbNodeType type = wb_robot_get_type();
  if (type == WB_NODE_DIFFERENTIAL_WHEELS) {
    mRobot = new DifferentialWheels();
  } else if (type == WB_NODE_SUPERVISOR) {
    mRobot = new Supervisor();
  } else {
    mRobot = new Robot();
  }


  std_msgs::String robotName;

  mEnd = false;
  mStep = 0;
  mStepSize = 1;
  fixName();

  //check ROS_MASTER_URI address and use default one if none already exist
  if (argc == 1)
  {
    if (getenv("ROS_MASTER_URI") == NULL)
    {
      #ifdef WIN32
        _putenv_s("ROS_MASTER_URI", "http://localhost:11311");
      #else
        setenv("ROS_MASTER_URI", "http://localhost:11311", 0);
      #endif
    }
  }
  else if (argc == 2)
  {
    #ifdef WIN32
      _putenv_s("ROS_MASTER_URI", argv[1]);
    #else
```

```
      setenv("ROS_MASTER_URI", argv[1], 0);
    #endif
  }
  else
  {
    printf("ERROR : Too many arguments\n");
    return;
  }

  ros::init(argc, argv, mRobotName);

  if (!ros::master::check()) {
    printf("Failed to contact master at %s. Please start ROS master and restart this controller\n",getenv("ROS_MASTER_URI"));
    exit(EXIT_SUCCESS);
  }

  mNodeHandle = new ros::NodeHandle;
  printf(" The controller is now connected to the ros master\n");

  mNamePublisher = mNodeHandle->advertise<std_msgs::String>("model_name", 1, true);
  robotName.data = mRobotName;
  mNamePublisher.publish(robotName);

  mTimeStepService = mNodeHandle->advertiseService(mRobotName+"/Robot/time_step", &Ros::timeStepCallback, this);
  mSetKeyboardService = mNodeHandle->advertiseService(mRobotName+"/Robot/set_keyboard", &Ros::setKeyboardCallback, this);
  mGetControllerNameService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_controller_name", &Ros::getControllerNameCallback, this);
  mGetControllerArgumentsService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_controller_arguments", &Ros::getControllerArgumentsCallback, this);
  mGetTimeService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_time", &Ros::getTimeCallback, this);
  mGetModelService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_model", &Ros::getModelCallback, this);
  mGetModeService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_mode", &Ros::getModeCallback, this);
  mGetSynchronizationService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_synchronization", &Ros::getSynchronizationCallback, this);
  mGetProjectPathService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_project_path", &Ros::getProjectPathCallback, this);
  mGetBasicTimeStepService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_basic_time_step", &Ros::getBasicTimeStepCallback, this);
  mGetNumberOfDevicesService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_number_of_devices", &Ros::getNumberOfDevicesCallback, this);
  mGetTypeService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_type", &Ros::getTypeCallback, this);
  mSetModeService = mNodeHandle->advertiseService(mRobotName+"/Robot/set_mode", &Ros::setModeCallback, this);

  if (mRobot->getType() == Node::DIFFERENTIAL_WHEELS)
    mRosDifferentialWheels = new RosDifferentialWheels(this, static_cast<DifferentialWheels *>(mRobot));
  else if (mRobot->getType() == Node::SUPERVISOR)
    mRosSupervisor = new RosSupervisor(this, static_cast<Supervisor *>(mRobot));

  setRosDevices();
}

Ros::~Ros()
{
  ros::shutdown();
  delete mRobot;
  for (unsigned int i = 0; i < mDeviceList.size(); i++)
    delete mDeviceList[i];
}

// create the unique name identifier of the controller that can be seen on ros network
// and used by other nodes to communicate with him
void Ros::fixName()
{
  string webotsPID;
  string webotsHostname;
  ostringstream s;

  // retrieve Webots' PID
  #ifdef WIN32
  HANDLE h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  PROCESSENTRY32 pe = { 0 };
  pe.dwSize = sizeof(PROCESSENTRY32);
  if (Process32First(h, &pe)) {
while (Process32Next(h, &pe) && s.str() == "") {
  if (!strcmp(pe.szExeFile, "ros.exe")) {
s << pe.th32ParentProcessID;
  }
}
  }
  CloseHandle(h);
  #else
  s << getppid();
  #endif
  webotsPID = s.str();

  // retrieve local hostname
  char hostname[256];
  gethostname(hostname, 256);
  webotsHostname = hostname;

  mRobotName = mRobot->getName();
  mRobotName += '_' + webotsPID + '_' + webotsHostname;
  // remove unhautorized symbols ('-' and '.') for ROS
  for (size_t i = 0; i < mRobotName.size(); i++)
  {
    if (mRobotName[i] == '-' || mRobotName[i] == '.')
      mRobotName[i] = '_';
  }
  printf("Robot's unique name is %s \n", mRobotName.c_str());
}

// runs accros the list of devices availables and creates the corresponding RosDevices.
// also stores pointers to sensors to be able to call their publishValues function at each step
void Ros::setRosDevices()
{
  int nDevices = mRobot->getNumberOfDevices();
  for (int i = 0; i < nDevices; i++)
  {
    Device *tempDevice = mRobot->getDeviceByIndex(i);
```

```
    switch (tempDevice->getType())
    {
      case Node::ACCELEROMETER :
        mSensorList.push_back(static_cast<RosSensor *>(new RosAccelerometer(dynamic_cast<Accelerometer *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::CAMERA :
        mSensorList.push_back(static_cast<RosSensor *>(new RosCamera(dynamic_cast<Camera *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::COMPASS :
        mSensorList.push_back(static_cast<RosSensor *>(new RosCompass(dynamic_cast<Compass *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::CONNECTOR :
        mDeviceList.push_back(static_cast<RosDevice *>(new RosConnector(dynamic_cast<Connector *>(tempDevice), this)));
        break;
      case Node::DISPLAY :
        mDeviceList.push_back(static_cast<RosDevice *>(new RosDisplay(dynamic_cast<Display *>(tempDevice), this)));
        break;
      case Node::DISTANCE_SENSOR :
        mSensorList.push_back(static_cast<RosSensor *>(new RosDistanceSensor(dynamic_cast<DistanceSensor *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::EMITTER :
        mDeviceList.push_back(static_cast<RosDevice *>(new RosEmitter(dynamic_cast<Emitter *>(tempDevice), this)));
        break;
      case Node::GPS :
        mSensorList.push_back(static_cast<RosSensor *>(new RosGPS(dynamic_cast<GPS *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::GYRO :
        mSensorList.push_back(static_cast<RosSensor *>(new RosGyro(dynamic_cast<Gyro *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::INERTIAL_UNIT :
        mSensorList.push_back(static_cast<RosSensor *>(new RosInertialUnit(dynamic_cast<InertialUnit *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::LED :
        mDeviceList.push_back(static_cast<RosDevice *>(new RosLED(dynamic_cast<LED *>(tempDevice), this)));
        break;
      case Node::LIGHT_SENSOR :
        mSensorList.push_back(static_cast<RosSensor *>(new RosLightSensor(dynamic_cast<LightSensor *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::LINEAR_MOTOR :
      {
        RosMotor * tempLinearMotor = new RosMotor(dynamic_cast<Motor *>(tempDevice), this);
        mDeviceList.push_back(static_cast<RosDevice *>(tempLinearMotor));
        mSensorList.push_back(static_cast<RosSensor *>(tempLinearMotor->mForceFeedbackSensor));
        mSensorList.push_back(static_cast<RosSensor *>(tempLinearMotor->mPositionSensor));
        break;
      }
      case Node::ROTATIONAL_MOTOR :
      {
        RosMotor * tempRotationalMotor = new RosMotor(dynamic_cast<Motor *>(tempDevice), this);
        mDeviceList.push_back(static_cast<RosDevice *>(tempRotationalMotor));
        mSensorList.push_back(static_cast<RosSensor *>(tempRotationalMotor->mTorqueFeedbackSensor));
        mSensorList.push_back(static_cast<RosSensor *>(tempRotationalMotor->mPositionSensor));
        break;
      }
      case Node::PEN :
        mDeviceList.push_back(static_cast<RosDevice *>(new RosPen(dynamic_cast<Pen *>(tempDevice), this)));
        break;
      case Node::POSITION_SENSOR :
        mSensorList.push_back(static_cast<RosSensor *>(new RosPositionSensor(dynamic_cast<PositionSensor *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
      case Node::RECEIVER :
        mDeviceList.push_back(static_cast<RosDevice *>(new RosReceiver(dynamic_cast<Receiver *>(tempDevice), this)));
        break;
      case Node::TOUCH_SENSOR :
        mSensorList.push_back(static_cast<RosSensor *>(new RosTouchSensor(dynamic_cast<TouchSensor *>(tempDevice), this)));
        mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
        break;
    }
  }
  mSensorList.push_back(static_cast<RosSensor *>(new RosBatterySensor(mRobot, this)));
  mDeviceList.push_back(static_cast<RosDevice *>(mSensorList.back()));
  // Once the list has been created, makes it available to other rosnodes
  mDeviceListService = mNodeHandle->advertiseService(mRobotName+'/'+"Robot"+'/'+"device_list", &Ros::deviceListCallback, this);
}

// timestep callback allowing a ros node to run the simulation step by step
bool Ros::timeStepCallback(webots_ros_controller::robot_set_time_step::Request &req, webots_ros_controller::robot_set_time_step::Response &res)
{
  if (req.step >= 1 && (req.step % static_cast<int>(mRobot->getBasicTimeStep()) == 0))
  {
    mStep++;
    if (mRobot->step(req.step) == -1)
    {
      mEnd = true;
      mStep = 0;
      res.success = -1;
    }
    else
      res.success = 1;
    if (req.step != mStepSize)
      mStepSize = req.step;

  }
  else if (req.step == 0)
```

```
  //the rosnode has stopped and won't control the simulation runtime anymore
  {
    mRobot->step(32);
    mStep = 0;
    res.success = 2;
  }
  else
    res.success = 0;
  return true;
}

// send the list of the name of the devices by index order
bool Ros::deviceListCallback(webots_ros_controller::robot_device_list::Request &req, webots_ros_controller::robot_device_list::Response &res)
{
  int nDevices = mRobot->getNumberOfDevices();
  std::string deviceNameFixed;
  for (int j = 0; j < nDevices; j++)
  {
deviceNameFixed = mRobot->getDeviceByIndex(j)->getName();
for (size_t i = 0; i < deviceNameFixed.size(); i++)
{
  if (deviceNameFixed[i] == '-' || deviceNameFixed[i] == ' ')
  deviceNameFixed[i] = '_';
}
    res.list.push_back(deviceNameFixed);
  }
  return true;
}

void Ros::run()
{
  int oldStep = 1;
  int tic = 0;
  int toc = 0;
  while (!mEnd && ros::ok())
  {
    if (!ros::master::check()) {
      printf("ros master has stopped or is not responding anymore\n");
      mEnd = true;
    }
    ros::spinOnce();
    for (unsigned int i = 0; i < mSensorList.size(); i++)
      mSensorList[i]->publishValues(mStep * mStepSize);

    if (mStep != 0)
    {
      oldStep = mStep;
      tic = time(0);
      while (mStep == oldStep && !mEnd)
      {
        ros::spinOnce();
        toc = time(0);
        if (difftime(toc,tic) >= 3.0)
        {
          int test = mRobot->step(0);
          tic = toc;
          if (test == -1)
            mEnd = true;
        }
      }
    }
    else if (mRobot->step(32) == -1)
      mEnd = true;
  }
}

bool Ros::setKeyboardCallback(webots_ros_controller::robot_set_keyboard::Request &req, webots_ros_controller::robot_set_keyboard::Response &res)
{
  if (req.period == 0)
  {
    mRobot->keyboardDisable();
    mGetKeyService.shutdown();
    res.success = 0;
  }
  else if( req.period % mStepSize == 0)
  {
    mRobot->keyboardEnable(req.period);
    mGetKeyService = mNodeHandle->advertiseService(mRobotName+"/Robot/get_key", &Ros::getKeyCallback, this);
    res.success = 1;
  }
  else
    res.success = -1;
  return true;
}

bool Ros::getKeyCallback(webots_ros_controller::robot_get_key::Request &req, webots_ros_controller::robot_get_key::Response &res)
{
  res.key = mRobot->keyboardGetKey();
  return true;
}
bool Ros::getControllerNameCallback(webots_ros_controller::robot_get_controller_name::Request &req,
                                    webots_ros_controller::robot_get_controller_name::Response &res)
{
  res.name = mRobot->getControllerName();
  return true;
}

bool Ros::getControllerArgumentsCallback(webots_ros_controller::robot_get_controller_arguments::Request &req,
                                         webots_ros_controller::robot_get_controller_arguments::Response &res)
{
  res.arguments = mRobot->getControllerArguments();
  return true;
}
```

```
bool Ros::getTimeCallback(webots_ros_controller::robot_get_time::Request &req, webots_ros_controller::robot_get_time::Response &res)
{
  res.time = mRobot->getTime();
  return true;
}

bool Ros::getModelCallback(webots_ros_controller::robot_get_model::Request &req, webots_ros_controller::robot_get_model::Response &res)
{
  res.model = mRobot->getModel();
  return true;
}

bool Ros::getModeCallback(webots_ros_controller::robot_get_mode::Request &req, webots_ros_controller::robot_get_mode::Response &res)
{
  res.mode = mRobot->getMode();
  return true;
}

bool Ros::getSynchronizationCallback(webots_ros_controller::robot_get_synchronization::Request &req,
                                     webots_ros_controller::robot_get_synchronization::Response &res)
{
  res.synchronization = mRobot->getSynchronization();
  return true;
}

bool Ros::getProjectPathCallback(webots_ros_controller::robot_get_project_path::Request &req,
                                 webots_ros_controller::robot_get_project_path::Response &res)
{
  res.path = mRobot->getProjectPath();
  return true;
}

bool Ros::getBasicTimeStepCallback(webots_ros_controller::robot_get_basic_time_step::Request &req,
                                   webots_ros_controller::robot_get_basic_time_step::Response &res)
{
  res.step = mRobot->getBasicTimeStep();
  return true;
}

bool Ros::getNumberOfDevicesCallback(webots_ros_controller::robot_get_number_of_devices::Request &req,
                                     webots_ros_controller::robot_get_number_of_devices::Response &res)
{
  res.number = mRobot->getNumberOfDevices();
  return true;
}

bool Ros::getTypeCallback(webots_ros_controller::robot_get_type::Request &req, webots_ros_controller::robot_get_type::Response &res)
{
  res.type = mRobot->getType();
  return true;
}

bool Ros::setModeCallback(webots_ros_controller::robot_set_mode::Request &req, webots_ros_controller::robot_set_mode::Response &res)
{
  void *arg;
  char buffer[req.arg.size()];
  for (unsigned int i = 0; i < req.arg.size(); i++)
    buffer[i] = req.arg[i];
  arg = buffer;
  mRobot->setMode(req.mode,arg);
  res.success = 1;
  return true;
}

// This is the main program of your controller.
// It creates an instance of your Robot subclass, launches its
// function(s) and destroys it at the end of the execution.
// Note that only one instance of Robot should be created in
// a controller program.
// The arguments of the main function can be specified by the
// "controllerArgs" field of the Robot node
int main(int argc, char **argv)
{

  Ros *controller = new Ros(argc, argv);
  controller->run();
  delete controller;
  return 0;
}
```

Ros.hpp :

```
/*****************************************************************************************************/
/* File:         Ros.hpp                                                                             */
/* Date:         Sept 2013                                                                           */
/* Description:  Header file for Ros controller                                                      */
/* Author:       Simon Puligny                                                                       */
/*****************************************************************************************************/

#ifndef ROS_HPP
#define ROS_HPP

#include <webots/Robot.hpp>
#include "ros/node_handle.h"
#include "services/robot_set_time_step.h"
#include "services/robot_device_list.h"
#include "services/robot_set_keyboard.h"
#include "services/robot_get_key.h"
#include "services/robot_get_controller_name.h"
#include "services/robot_get_controller_arguments.h"
```

```
#include "services/robot_get_time.h"
#include "services/robot_get_model.h"
#include "services/robot_get_mode.h"
#include "services/robot_set_mode.h"
#include "services/robot_get_synchronization.h"
#include "services/robot_get_project_path.h"
#include "services/robot_get_basic_time_step.h"
#include "services/robot_get_number_of_devices.h"
#include "services/robot_get_type.h"

using namespace webots;

class RosSensor;
class RosDevice;
class RosDifferentialWheels;
class RosSupervisor;

class Ros
{
  public:
    Ros(int argc,char **argv);
    virtual ~Ros();

    void             run();
    ros::NodeHandle *nodeHandle() {return mNodeHandle;}
    int              stepSize() {return mStepSize;}
    std::string      name() {return mRobotName;}

  private:
    void fixName();
    void setRosDevices();
    bool timeStepCallback(webots_ros_controller::robot_set_time_step::Request &req, webots_ros_controller::robot_set_time_step::Response &res);
    bool deviceListCallback(webots_ros_controller::robot_device_list::Request &req, webots_ros_controller::robot_device_list::Response &res);
    bool setKeyboardCallback(webots_ros_controller::robot_set_keyboard::Request &req, webots_ros_controller::robot_set_keyboard::Response &res);
    bool getKeyCallback(webots_ros_controller::robot_get_key::Request &req, webots_ros_controller::robot_get_key::Response &res);
    bool getControllerNameCallback(webots_ros_controller::robot_get_controller_name::Request &req,
                                   webots_ros_controller::robot_get_controller_name::Response &res);
    bool getControllerArgumentsCallback(webots_ros_controller::robot_get_controller_arguments::Request &req,
                                        webots_ros_controller::robot_get_controller_arguments::Response &res);
    bool getTimeCallback(webots_ros_controller::robot_get_time::Request &req, webots_ros_controller::robot_get_time::Response &res);
    bool getModelCallback(webots_ros_controller::robot_get_model::Request &req,
                          webots_ros_controller::robot_get_model::Response &res);
    bool getModeCallback(webots_ros_controller::robot_get_mode::Request &req, webots_ros_controller::robot_get_mode::Response &res);
    bool getSynchronizationCallback(webots_ros_controller::robot_get_synchronization::Request &req,
                                    webots_ros_controller::robot_get_synchronization::Response &res);
    bool getProjectPathCallback(webots_ros_controller::robot_get_project_path::Request &req,
                                webots_ros_controller::robot_get_project_path::Response &res);
    bool getBasicTimeStepCallback(webots_ros_controller::robot_get_basic_time_step::Request &req,
                                  webots_ros_controller::robot_get_basic_time_step::Response &res);
    bool getNumberOfDevicesCallback(webots_ros_controller::robot_get_number_of_devices::Request &req,
                                    webots_ros_controller::robot_get_number_of_devices::Response &res);
    bool getTypeCallback(webots_ros_controller::robot_get_type::Request &req, webots_ros_controller::robot_get_type::Response &res);
    bool setModeCallback(webots_ros_controller::robot_set_mode::Request &req, webots_ros_controller::robot_set_mode::Response &res);

    Robot                   *mRobot;
    std::string              mRobotName;
    std::vector<RosDevice *> mDeviceList;
    std::vector<RosSensor *> mSensorList;
    RosDifferentialWheels   *mRosDifferentialWheels;
    RosSupervisor           *mRosSupervisor;
    ros::NodeHandle         *mNodeHandle;
    ros::Publisher           mNamePublisher;
    ros::ServiceServer       mTimeStepService;
    ros::ServiceServer       mDeviceListService;
    ros::ServiceServer       mSetKeyboardService;
    ros::ServiceServer       mGetKeyService;
    ros::ServiceServer       mGetControllerNameService;
    ros::ServiceServer       mGetControllerArgumentsService;
    ros::ServiceServer       mGetTimeService;
    ros::ServiceServer       mGetModelService;
    ros::ServiceServer       mGetModeService;
    ros::ServiceServer       mGetSynchronizationService;
    ros::ServiceServer       mGetProjectPathService;
    ros::ServiceServer       mGetBasicTimeStepService;
    ros::ServiceServer       mGetNumberOfDevicesService;
    ros::ServiceServer       mGetTypeService;
    ros::ServiceServer       mSetModeService;
    unsigned int             mStepSize;
    int                      mStep;
    bool                     mEnd;
};

#endif //ROS_HPP
```

7.2 URDF to VRML script

URDFtoVRML.py :

```
#! /usr/bin/env python

import shutil
import os
import string
import sys
import xml
import parserURDF
import writeProto

from xml.dom import minidom


xmlFile = sys.argv[1]
domFile = minidom.parse(xmlFile)

for child in domFile.childNodes:
    if child.localName == 'robot':
        robotName = parserURDF.getRobotName(child)
        protoFile = os.path.splitext(xmlFile)[0]
        robot = child
        protoFile = open(protoFile+'.proto','w')
        writeProto.header(protoFile,xmlFile,robotName)
        writeProto.declaration(protoFile,robotName)
        linkElementList = []
        jointElementList = []
        for child in robot.childNodes:
            if child.localName == 'link':
                linkElementList.append(child)
            elif child.localName == 'joint':
                jointElementList.append(child)

        linkList = []
        jointList = []
        parentList = []
        childList = []
        rootLink = parserURDF.Link()

        for joint in jointElementList:
            jointList.append(parserURDF.getJoint(joint))
            parentList.append(jointList[-1].parent.encode("ascii"))
            childList.append(jointList[-1].child.encode("ascii"))
        parentList.sort()
        childList.sort()
        for link in linkElementList:
            linkList.append(parserURDF.getLink(link))
            if parserURDF.isRootLink(linkList[-1].name,childList):
                rootLink = linkList[-1]
                print 'root link is ' + rootLink.name
        pluginList = parserURDF.getPlugins(robot)
        print 'there is ' + str(len(linkList)) + ' links, ' + str(len(jointList)) + ' joints and ' + str(len(pluginList)) + ' plugins'

        writeProto.URDFLink(protoFile, rootLink, 3, parentList, childList, linkList, jointList)
        protoFile.write('    ]\n')
        writeProto.basicPhysics(protoFile)
        protoFile.write('  }\n')
        protoFile.write('}\n')
        protoFile.close()
        exit(1)
print 'could not read file'
```

ParserURDF.py :

```
#! /usr/bin/env python


import os
import string
import sys
import xml
import struct
import math
import numpy
import Image
from collada import *

counter = 0

class Trimesh():
    def __init__(self):
        self.coord = [] #list of coordinate points
        self.coordIndex = [] #list of index of points
        self.texCoord = [] #list of coordinate points for texture
        self.texCoordIndex = [] #list of index for texture

class Inertia():
    def __init__(self):
        self.position = [0.0, 0.0, 0.0]
        self.rotation = [1.0, 0.0, 0.0, 0.0]
        self.mass = 1.0
        self.ixx = 1.0
        self.ixy = 0.0
        self.ixz = 0.0
        self.iyy = 1.0
        self.iyz = 0.0
        self.izz = 1.0
```

```
class Box():
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
        self.z = 0.0

class Cylinder():
    def __init__(self):
        self.radius = 0.0
        self.length = 0.0

class Sphere():
    def __init__(self):
        self.radius = 0.0

class Geometry():
    def __init__(self):
        self.box = Box()
        self.cylinder = Cylinder()
        self.sphere = Sphere()
        self.trimesh = Trimesh()
        self.scale = [1.0, 1.0, 1.0]

class Color():
    def __init__(self):
        self.red = 0.0
        self.green = 0.0
        self.blue = 0.0
        self.alpha = 0.0

class Material():
    def __init__(self):
        self.color = Color()
        self.texture = ""

class Visual():
    def __init__(self):
        self.position = [0.0, 0.0, 0.0]
        self.rotation = [1.0, 0.0, 0.0, 0.0]
        self.geometry = Geometry()
        self.material = Material()

class Collision():
    def __init__(self):
        self.position = [0.0, 0.0, 0.0]
        self.rotation = [1.0, 0.0, 0.0, 0.0]
        self.geometry = Geometry()

class Calibration():
    def __init__(self):
        self.limit = 0.0
        self.rising = True

class Dynamics():
    def __init__(self):
        self.damping = 0.0
        self.friction = 0.0

class Limit():
    def __init__(self):
        self.lower = 0.0
        self.upper = 0.0
        self.effort = 0.0
        self.velocity = 0.0

class Safety():
    def __init__(self):
        self.lower = 0.0
        self.upper = 0.0
        self.kPosition = 0.0
        self.kVelocity = 0.0

class Link():
    def __init__(self):
        self.name = 'default'
        self.inertia = Inertia()
        self.visual = []
        self.collision = []

class Joint():
    def __init__(self):
        self.name = 'default'
        self.type = 'default'
        self.position = [0.0, 0.0, 0.0]
        self.rotation = [1.0, 0.0, 0.0, 0.0]
        self.parent = 'default'
        self.child = 'default'
        self.axis = []
        self.calibration = Calibration()
        self.dynamics = Dynamics()
        self.limit = Limit()
        self.safety = Safety()

def vector_norm(data, axis=None, out=None):
    data = numpy.array(data, dtype=numpy.float64, copy=True)
    if out is None:
        if data.ndim == 1:
            return math.sqrt(numpy.dot(data, data))
        data *= data
        out = numpy.atleast_1d(numpy.sum(data, axis=axis))
        numpy.sqrt(out, out)
        return out
    else:
```

```
        data *= data
        numpy.sum(data, axis=axis, out=out)
        numpy.sqrt(out, out)

# euler-axes-angle (vrml) to quaternion
def vrml_to_q(v):
    q = [0.0, 0.0, 0.0, 0.0]
    l = v[0] * v[0] + v[1] * v[1] + v[2] * v[2]
    if (l > 0.0):
        q[0] = math.cos(v[3] / 2)
        l = math.sin(v[3] / 2) / math.sqrt(l)
        q[1] = v[0] * l
        q[2] = v[1] * l
        q[3] = v[2] * l
    else:
        q[0] = 1
        q[1] = 0
        q[2] = 0
        q[3] = 0
    return q

# quaternion to euler-axes-angle (vrml)
def q_to_vrml(q):
    v = [0.0, 0.0, 0.0, 0.0]
    v[3] = 2.0 * math.acos(q[0])
    if (v[3] < 0.0001):
        # if v[3] close to zero then direction of axis not important
        v[0] = 0.0
        v[1] = 1.0
        v[2] = 0.0
    else:
        # normalise axes
        n = math.sqrt(q[1] * q[1] + q[2] * q[2] + q[3] * q[3])
        v[0] = q[1] / n
        v[1] = q[2] / n
        v[2] = q[3] / n
    return v


# quaternion multiplication (combining rotations)
def q_mult(qb, qc):
    qa = [0.0, 0.0, 0.0, 0.0]
    qa[0] = qb[0]*qc[0] - qb[1]*qc[1] - qb[2]*qc[2] - qb[3]*qc[3]
    qa[1] = qb[0]*qc[1] + qb[1]*qc[0] + qb[2]*qc[3] - qb[3]*qc[2]
    qa[2] = qb[0]*qc[2] + qb[2]*qc[0] + qb[3]*qc[1] - qb[1]*qc[3]
    qa[3] = qb[0]*qc[3] + qb[3]*qc[0] + qb[1]*qc[2] - qb[2]*qc[1]
    return qa

def convertRPYtoEulerAxis(rpy, cylinder=False):
    offset2 = 0.0
    if cylinder:
        offset2 = 1.57
    ea1 = [0.0, 1.0, 0.0, rpy[1]]
    ea2 = [1.0, 0.0, 0.0, rpy[0] + offset2]
    ea3 = [0.0, 0.0, 1.0, rpy[2]]
    # convert vrml to quaternion representation
    q1 = vrml_to_q(ea1)
    q2 = vrml_to_q(ea2)
    q3 = vrml_to_q(ea3)
    #combine 3 quaternions into 1
    qa = q_mult(q1, q2)
    qb = q_mult(qa, q3)
    # convert quaternion to vrml
    return q_to_vrml(qb)


def getRobotName(node):
    name = node.getAttribute('name')
    print 'the name of the robot is ' + name
    return name

def getPlugins(node):
    pluginList = []
    for child in node.childNodes:
        if child.localName != 'link' and child.localName != 'joint' and child.nodeType == xml.dom.minidom.Node.ELEMENT_NODE:
                pluginList.append(child)
    return pluginList

def hasElement(node,element):
    if node.getElementsByTagName(element).length > 0:
        return True
    else:
        return False


def getSTLMesh(filename, node):
    stlFile = open(filename, 'rb')
    stlFile.read(80)
    vertex1 = []
    vertex2 = []
    vertex3 = []
    numTriangles = struct.unpack("@i", stlFile.read(4))[0]
    struct.unpack("<3f", stlFile.read(12))
    a = struct.unpack("<3f", stlFile.read(12))
    vertex1.append(a)
    b = struct.unpack("<3f", stlFile.read(12))
    vertex2.append(b)
    c = struct.unpack("<3f", stlFile.read(12))
    vertex3.append(c)
    struct.unpack("H", stlFile.read(2))
    node.geometry.trimesh.coord.append(vertex1[0])
    node.geometry.trimesh.coord.append(vertex2[0])
    node.geometry.trimesh.coord.append(vertex3[0])
```

```
    for i in range(1, numTriangles):
        struct.unpack("<3f", stlFile.read(12))
        a = struct.unpack("<3f", stlFile.read(12))
        vertex1.append(a)
        if node.geometry.trimesh.coord.count(a) == 0:
            node.geometry.trimesh.coord.append(a)
        b = struct.unpack("<3f", stlFile.read(12))
        vertex2.append(b)
        if node.geometry.trimesh.coord.count(b) == 0:
            node.geometry.trimesh.coord.append(b)
        c = struct.unpack("<3f", stlFile.read(12))
        vertex3.append(c)
        if node.geometry.trimesh.coord.count(c) == 0:
            node.geometry.trimesh.coord.append(c)
        struct.unpack("H", stlFile.read(2))
        node.geometry.trimesh.coordIndex.append([node.geometry.trimesh.coord.index(vertex1[i]),
                                                 node.geometry.trimesh.coord.index(vertex2[i]),
                                                 node.geometry.trimesh.coord.index(vertex3[i])])
    stlFile.close()
    return node


def getColladaMesh(filename, node, link):
    colladaMesh = Collada(filename)
    if node.material:
        for geometry in list(colladaMesh.scene.objects('geometry')):
            visual = Visual()
            visual.position = node.position
            visual.rotation = node.rotation
            visual.material.color = node.material.color
            visual.material.texture = ""
            visual.geometry.scale = node.geometry.scale
            for value in list(geometry.primitives())[0].vertex:
                visual.geometry.trimesh.coord.append(numpy.array(value))
            for value in list(geometry.primitives())[0].vertex_index:
                visual.geometry.trimesh.coordIndex.append(value)
            for value in list(geometry.primitives())[0].texcoordset[0]:
                visual.geometry.trimesh.texCoord.append(value)
            for value in list(geometry.primitives())[0].texcoord_indexset[0]:
                visual.geometry.trimesh.texCoordIndex.append(value)
            if list(geometry.primitives())[0].material and list(geometry.primitives())[0].material.effect:
                visual.material.texture ='textures/' + list(geometry.primitives())[0].material.effect.diffuse.sampler.surface.image.path.split('/')[-1]
                if os.path.splitext(visual.material.texture)[1] == '.tiff' or os.path.splitext(visual.material.texture)[1] == '.tif':
                    for dirname, dirnames, filenames in os.walk('.'):
                        for filename in filenames:
                            if filename == str(visual.material.texture.split('/')[-1]):
                                try:
                                    tifImage = Image.open(os.path.join(dirname, filename))
                                    tifImage.save(os.path.splitext(os.path.join(dirname, filename))[0] + '.png')
                                    visual.material.texture = 'textures/' + os.path.splitext(filename)[0] + '.png'
                                    print 'translated image', visual.material.texture
                                except:
                                    visual.material.texture = ""
                                    print 'failed to open ' + os.path.join(dirname, filename)
            link.visual.append(visual)
    else:
        for geometry in list(colladaMesh.scene.objects('geometry')):
            collision = Collision()
            collision.position = node.position
            collision.rotation = node.rotation
            collision.geometry.scale = node.geometry.scale
            for value in list(geometry.primitives())[0].vertex:
                collision.geometry.trimesh.coord.append(numpy.array(value))
            for value in list(geometry.primitives())[0].vertex_index:
                collision.geometry.trimesh.coordIndex.append(value)
            link.collision.append(collision)

def getPosition(node):
    position = [0.0, 0.0, 0.0]
    positionString = node.getElementsByTagName('origin')[0].getAttribute('xyz').split()
    position[0] = float(positionString[0])
    position[1] = float(positionString[1])
    position[2] = float(positionString[2])
    return position

def getRotation(node, isCylinder = False):
    rotation = [0.0, 0.0, 0.0]
    if hasElement(node, 'origin'):
        orientationString = node.getElementsByTagName('origin')[0].getAttribute('rpy').split()
        rotation[0] = float(orientationString[0])
        rotation[1] = float(orientationString[1])
        rotation[2] = float(orientationString[2])
    if isCylinder:
        return convertRPYtoEulerAxis(rotation, True)
    else:
        return convertRPYtoEulerAxis(rotation, False)

def getInertia(node):
    inertia = Inertia()
    inertialElement = node.getElementsByTagName('inertial')[0]
    if hasElement(inertialElement,'origin'):
        if inertialElement.getElementsByTagName('origin')[0].getAttribute('xyz'):
            inertia.position = getPosition(inertialElement)
        if inertialElement.getElementsByTagName('origin')[0].getAttribute('rpy'):
            inertia.rotation = getRotation(inertialElement)
    if hasElement(inertialElement,'mass'):
        inertia.mass = float(inertialElement.getElementsByTagName('mass')[0].getAttribute('value'))
    if hasElement(inertialElement,'inertia'):
        matrixNode = inertialElement.getElementsByTagName('inertia')[0]
        inertia.ixx = float(matrixNode.getAttribute('ixx'))
        inertia.ixy = float(matrixNode.getAttribute('ixy'))
        inertia.ixz = float(matrixNode.getAttribute('ixz'))
        inertia.iyy = float(matrixNode.getAttribute('iyy'))
```

```
        inertia.iyz = float(matrixNode.getAttribute('iyz'))
        inertia.izz = float(matrixNode.getAttribute('izz'))
    return inertia

def getVisual(link, node):
    for index in range(0, len(node.getElementsByTagName('visual'))):
        visual = Visual()
        visualElement = node.getElementsByTagName('visual')[index]
        if hasElement(visualElement,'origin'):
            if visualElement.getElementsByTagName('origin')[0].getAttribute('xyz'):
                visual.position = getPosition(visualElement)
            if visualElement.getElementsByTagName('origin')[0].getAttribute('rpy'):
                if hasElement(visualElement.getElementsByTagName('geometry')[0], 'cylinder'):
                    visual.rotation = getRotation(visualElement, True)
                else:
                    visual.rotation = getRotation(visualElement)
        elif hasElement(visualElement.getElementsByTagName('geometry')[0], 'cylinder'):
                visual.rotation = getRotation(visualElement, True)

        geometryElement = visualElement.getElementsByTagName('geometry')[0]

        if hasElement(visualElement,'material'):
            if hasElement(visualElement.getElementsByTagName('material')[0],'color'):
                colorElement = visualElement.getElementsByTagName('material')[0].getElementsByTagName('color')[0].getAttribute('rgba').split()
                visual.material.color.red = float(colorElement[0])
                visual.material.color.green = float(colorElement[1])
                visual.material.color.blue = float(colorElement[2])
                visual.material.color.alpha = float(colorElement[3])
            if hasElement(visualElement.getElementsByTagName('material')[0],'texture'):
                visual.material.texture = visualElement.getElementsByTagName('material')[0].getElementsByTagName('texture')[0].getAttribute('filename')
                if os.path.splitext(visual.material.texture)[1] == '.tiff' or os.path.splitext(visual.material.texture)[1] == '.tif':
                    for dirname, dirnames, filenames in os.walk('.'):
                        for filename in filenames:
                            if filename == str(visual.material.texture.split('/')[-1]):
                                print 'try to translate image', filename
                                try:
                                    tifImage = Image.open(os.path.join(dirname, filename))
                                    tifImage.save(os.path.splitext(os.path.join(dirname, filename))[0] + '.png')
                                    visual.material.texture = 'textures/' + os.path.splitext(filename)[0] + '.png'
                                except:
                                    visual.material.texture = ""
                                    print 'failed to open ' + os.path.join(dirname, filename)

        if hasElement(geometryElement,'box'):
            visual.geometry.box.x = float(geometryElement.getElementsByTagName('box')[0].getAttribute('size').split()[0])
            visual.geometry.box.y = float(geometryElement.getElementsByTagName('box')[0].getAttribute('size').split()[1])
            visual.geometry.box.z = float(geometryElement.getElementsByTagName('box')[0].getAttribute('size').split()[2])
            link.visual.append(visual)
        elif hasElement(geometryElement,'cylinder'):
            visual.geometry.cylinder.radius = float(geometryElement.getElementsByTagName('cylinder')[0].getAttribute('radius'))
            visual.geometry.cylinder.length = float(geometryElement.getElementsByTagName('cylinder')[0].getAttribute('length'))
            link.visual.append(visual)
        elif hasElement(geometryElement,'sphere'):
            visual.geometry.sphere.radius = float(geometryElement.getElementsByTagName('sphere')[0].getAttribute('radius'))
            link.visual.append(visual)
        elif hasElement(geometryElement,'mesh'):
            meshfile = geometryElement.getElementsByTagName('mesh')[0].getAttribute('filename')
            #hack for gazebo mesh database
            if meshfile.count('package'):
                meshfile = str(meshfile.split('/')[-2]) + '/' + str(meshfile.split('/')[-1])
            if geometryElement.getElementsByTagName('mesh')[0].getAttribute('scale'):
                meshScale = geometryElement.getElementsByTagName('mesh')[0].getAttribute('scale').split()
                visual.geometry.scale[0] = float(meshScale[0])
                visual.geometry.scale[1] = float(meshScale[1])
                visual.geometry.scale[2] = float(meshScale[2])
            if os.path.splitext(meshfile)[1] == '.dae':
                getColladaMesh(meshfile, visual, link)
            elif os.path.splitext(meshfile)[1] == '.stl':
                visual = getSTLMesh(meshfile, visual)
                link.visual.append(visual)

def getCollision(link, node):
    for index in range(0,len(node.getElementsByTagName('collision'))):
        collision = Collision()
        collisionElement = node.getElementsByTagName('collision')[index]
        if hasElement(collisionElement,'origin'):
            if collisionElement.getElementsByTagName('origin')[0].getAttribute('xyz'):
                collision.position = getPosition(collisionElement)
            if collisionElement.getElementsByTagName('origin')[0].getAttribute('rpy'):
                if hasElement(collisionElement.getElementsByTagName('geometry')[0], 'cylinder'):
                    collision.rotation = getRotation(collisionElement, True)
                else:
                    collision.rotation = getRotation(collisionElement)
        elif hasElement(collisionElement.getElementsByTagName('geometry')[0], 'cylinder'):
                collision.rotation = getRotation(collisionElement, True)

        geometryElement = collisionElement.getElementsByTagName('geometry')[0]
        if hasElement(geometryElement,'box'):
            collision.geometry.box.x = float(geometryElement.getElementsByTagName('box')[0].getAttribute('size').split()[0])
            collision.geometry.box.y = float(geometryElement.getElementsByTagName('box')[0].getAttribute('size').split()[1])
            collision.geometry.box.z = float(geometryElement.getElementsByTagName('box')[0].getAttribute('size').split()[2])
            link.collision.append(collision)
        elif hasElement(geometryElement,'cylinder'):
            collision.geometry.cylinder.radius = float(geometryElement.getElementsByTagName('cylinder')[0].getAttribute('radius'))
            collision.geometry.cylinder.length = float(geometryElement.getElementsByTagName('cylinder')[0].getAttribute('length'))
            link.collision.append(collision)
        elif hasElement(geometryElement,'sphere'):
            collision.geometry.sphere.radius = float(geometryElement.getElementsByTagName('sphere')[0].getAttribute('radius'))
            link.collision.append(collision)
        elif hasElement(geometryElement,'mesh'):
            meshfile = geometryElement.getElementsByTagName('mesh')[0].getAttribute('filename')
            if geometryElement.getElementsByTagName('mesh')[0].getAttribute('scale'):
                meshScale = geometryElement.getElementsByTagName('mesh')[0].getAttribute('scale').split()
```

```
                collision.geometry.scale[0] = float(meshScale[0])
                collision.geometry.scale[1] = float(meshScale[1])
                collision.geometry.scale[2] = float(meshScale[2])
            # hack for gazebo mesh database
            if meshfile.count('package'):
                meshfile = str(meshfile.split('/')[-2]) + '/' + str(meshfile.split('/')[-1])
            if os.path.splitext(meshfile)[1] == '.dae':
                collision.geometry.collada = getColladaMesh(meshfile, collision, link)
            elif os.path.splitext(meshfile)[1] == '.stl':
                collision.geometry.stl = getSTLMesh(meshfile, collision)
                link.collision.append(collision)

def getAxis(node):
    axis = [0.0, 0.0, 0.0]
    axisElement = node.getElementsByTagName('axis')[0].getAttribute('xyz').split()
    axis[0] = float(axisElement[0])
    axis[1] = float(axisElement[1])
    axis[2] = float(axisElement[2])
    return axis

def getCalibration(node):
    calibration = Calibration()
    calibrationElement = node.getElementsByTagName('calibration')[0]
    if hasElement(calibrationElement,'rising'):
        calibration.limit = calibrationElement.getAttribute('rising')
        calibration.rising = True
    else:
        calibration.limit = calibrationElement.getAttribute('falling')
        calibration.rising = False
    return calibration

def getDynamics(node):
    dynamics = Dynamics()
    dynamicsElement = node.getElementsByTagName('dynamics')[0]
    if dynamicsElement.getAttribute('damping'):
        dynamics.damping = float(dynamicsElement.getAttribute('damping'))
    if dynamicsElement.getAttribute('friction'):
        dynamics.friction = float(dynamicsElement.getAttribute('friction'))
    return dynamics

def getLimit(node):
    limit = Limit()
    limitElement = node.getElementsByTagName('limit')[0]
    if limitElement.getAttribute('lower'):
        limit.lower = float(limitElement.getAttribute('lower'))
    if limitElement.getAttribute('upper'):
        limit.upper = float(limitElement.getAttribute('upper'))
    limit.effort = float(limitElement.getAttribute('effort'))
    limit.velocity = float(limitElement.getAttribute('velocity'))
    return limit

def getSafety(node):
    safety = Safety()
    if node.getElementsByTagName('safety_controller')[0].getAttribute('soft_lower_limit'):
        safety.lower = float(node.getElementsByTagName('safety_controller')[0].getAttribute('soft_lower_limit'))
    if node.getElementsByTagName('safety_controller')[0].getAttribute('soft_upper_limit'):
        safety.upper = float(node.getElementsByTagName('safety_controller')[0].getAttribute('soft_upper_limit'))
    if node.getElementsByTagName('safety_controller')[0].getAttribute('k_position'):
        safety.kPosition = float(node.getElementsByTagName('safety_controller')[0].getAttribute('k_position'))
    safety.kVelocity = float(node.getElementsByTagName('safety_controller')[0].getAttribute('k_velocity'))
    return safety

def getLink(node):
    link = Link()
    link.name = node.getAttribute('name')
    if hasElement(node,'inertial'):
        link.inertia = getInertia(node)
    if hasElement(node,'visual'):
        getVisual(link, node)
    if hasElement(node,'collision'):
        getCollision(link, node)
    return link

def getJoint(node):
    joint = Joint()
    joint.name = node.getAttribute('name')
    joint.type = node.getAttribute('type')
    if hasElement(node,'origin'):
        if node.getElementsByTagName('origin')[0].getAttribute('xyz'):
            joint.position = getPosition(node)
        if node.getElementsByTagName('origin')[0].getAttribute('rpy'):
            joint.rotation = getRotation(node)
    joint.parent = node.getElementsByTagName('parent')[0].getAttribute('link')
    joint.child = node.getElementsByTagName('child')[0].getAttribute('link')
    if hasElement(node,'axis'):
        joint.axis = getAxis(node)
    if hasElement(node,'calibration'):
        joint.calibration = getCalibration(node)
    if hasElement(node,'dynamics'):
        joint.dynamics = getDynamics(node)
    if hasElement(node,'limit'):
        joint.limit = getLimit(node)
    if hasElement(node,'safety_controller'):
        joint.safety = getSafety(node)
    return joint

def isRootLink(link,childList):
    for child in childList:
        if link == child:
            return False
    return True
```

writeProto.py :

```
#! /usr/bin/env python


import os
import string
import sys
import xml


def header(proto,srcFile, robotName):
    proto.write('#VRML_SIM V7.4.0 utf8\n')
    proto.write('# This is a proto file for Webots for the ' + robotName + '\n')
    proto.write('# Extracted from: ' + srcFile + '\n\n')

def declaration(proto, robotName):
    proto.write('PROTO '+robotName+' [\n')
    proto.write('   field SFVec3f translation 0 0 0\n')
    proto.write('   field SFRotation rotation 1 0 0 -1.57\n')
    proto.write('   field SFString controller "void"\n')
    proto.write(']\n')
    proto.write('{\n')
    proto.write('  Robot {\n')
    proto.write('    translation IS translation\n')
    proto.write('    rotation IS rotation\n')
    proto.write('    controller IS controller\n')
    proto.write('    children [\n')

def basicPhysics(proto):
    proto.write('    physics Physics {\n')
    proto.write('      density -1\n')
    proto.write('      mass 1\n')
    proto.write('      inertiaMatrix [1 1 1 0 0 0]\n')
    proto.write('      centerOfMass [0 0 0]\n')
    proto.write('    }\n')

def URDFLink(proto, link, level, parentList, childList, linkList, jointList, fixed = False, jointPosition = [0.0, 0.0, 0.0], jointRotation = [1.0, 0.0, 0.0, 0.0]):
    indent = '  '
    proto.write(level * indent  + 'DEF ' + link.name + ' Solid {\n')
    if fixed:
        proto.write((level + 1) * indent + 'translation ' + str(jointPosition[0]) + ' '
                    + str(jointPosition[1]) + ' ' + str(jointPosition[2]) + '\n')
        proto.write((level + 1) * indent + 'rotation ' + str(jointRotation[0]) + ' ' + str(jointRotation[1])
                    + ' ' + str(jointRotation[2]) + ' ' + str(jointRotation[3]) + '\n')
    else:
        proto.write((level + 1) * indent + 'translation 0 0 0\n')
        proto.write((level + 1) * indent + 'rotation 1 0 0 0\n')
    proto.write((level + 1) * indent + 'physics Physics {\n')
    proto.write((level + 2) * indent + 'density -1\n')
    proto.write((level + 2) * indent + 'mass ' + str(link.inertia.mass) + '\n')
    proto.write((level + 2) * indent + 'inertiaMatrix [' + str(link.inertia.ixx) + ' ' + str(link.inertia.iyy)
                + ' ' + str(link.inertia.izz) + ' ' + str(link.inertia.ixy)
                + ' ' + str(link.inertia.ixz) + ' ' + str(link.inertia.iyz) + ']\n')
    proto.write((level + 2) * indent + 'centerOfMass [0.0 0.0 0.0]\n')
    proto.write((level + 1) * indent + '}\n')

    if link.collision:
        URDFBoundingObject(proto, link, level + 1)

    proto.write((level + 1) * indent + 'children [\n')

    if link.visual:
        URDFShape(proto, link, level + 2)

    for joint in jointList:
        if joint.parent == link.name:
            URDFJoint(proto, joint, level + 2, parentList, childList, linkList, jointList)
    proto.write((level + 1) * indent + ']\n')
    proto.write(level * indent + '}\n')


def URDFBoundingObject(proto, link, level):
    indent = '  '
    boundingLevel = level
    proto.write(level * indent + 'boundingObject ')
    if len(link.collision) > 1:
        proto.write('Group {\n')
        proto.write((level + 1) * indent + 'children [\n')
        boundingLevel = level + 2

    for boundingObject in link.collision:
        if boundingObject.position != [0.0, 0.0, 0.0] or  boundingObject.rotation[3] != 0.0:
            if len(link.collision) > 1:
                proto.write((level + 2) * indent + 'Transform {\n')
            else:
                proto.write('Transform {\n')
            proto.write((boundingLevel + 1) * indent + 'translation ' + str(boundingObject.position[0])
                        + ' ' + str(boundingObject.position[1]) + ' ' + str(boundingObject.position[2]) + '\n')
            proto.write((boundingLevel + 1) * indent + 'rotation ' + str(boundingObject.rotation[0])
                        + ' ' + str(boundingObject.rotation[1]) + ' ' + str(boundingObject.rotation[2])
                        + ' ' + str(boundingObject.rotation[3]) + '\n')
            proto.write((boundingLevel + 1) * indent + 'children [\n')
            boundingLevel = boundingLevel + 2

        if boundingObject.geometry.box.x != 0:
            proto.write(boundingLevel * indent + 'Box {\n')
            proto.write((boundingLevel + 1) * indent + ' size ' + str(boundingObject.geometry.box.x)
                        + ' ' + str(boundingObject.geometry.box.y) + ' ' + str(boundingObject.geometry.box.z) + '\n')
            proto.write(boundingLevel * indent + '}\n')

        elif boundingObject.geometry.cylinder.radius != 0 and boundingObject.geometry.cylinder.length != 0:
            proto.write(boundingLevel * indent + 'Cylinder {\n')
            proto.write((boundingLevel + 1) * indent + 'radius ' + str(boundingObject.geometry.cylinder.radius) + '\n')
            proto.write((boundingLevel + 1) * indent + 'height ' + str(boundingObject.geometry.cylinder.length) + '\n')
```

```
            proto.write(boundingLevel * indent + '}\n')

        elif boundingObject.geometry.sphere.radius != 0:
            proto.write(boundingLevel * indent + 'Sphere {\n')
            proto.write((boundingLevel + 1) * indent + 'radius ' + str(boundingObject.geometry.sphere.radius) + '\n')
            proto.write(boundingLevel * indent + '}\n')

        elif boundingObject.geometry.trimesh.coord != []:
            proto.write(boundingLevel* indent + 'IndexedFaceSet {\n')

            proto.write((boundingLevel + 1) * indent + 'coord Coordinate {\n')
            proto.write((boundingLevel + 2) * indent + 'point [\n')
            for value in boundingObject.geometry.trimesh.coord:
                proto.write(boundingLevel * indent + str(value[0] * boundingObject.geometry.scale[0])
                            + ' ' + str(value[1] * boundingObject.geometry.scale[1]) + ' '
                            + str(value[2] * boundingObject.geometry.scale[2]) + '\n')
            proto.write((boundingLevel + 2) * indent + ']\n')
            proto.write((boundingLevel + 1) * indent + '}\n')

            proto.write((boundingLevel + 1) * indent + 'coordIndex [\n')
            for value in boundingObject.geometry.trimesh.coordIndex:
                proto.write(boundingLevel * indent + str(value[0]) + ' ' + str(value[1]) + ' ' + str(value[2]) + ' -1\n')
            proto.write((boundingLevel + 1) * indent + ']\n')

            proto.write((boundingLevel + 1) * indent + 'creaseAngle 1\n')
            proto.write(boundingLevel * indent + '}\n')

        else:
            proto.write((boundingLevel + 1) * indent + 'Box{\n')
            proto.write((boundingLevel + 1) * indent + ' size 0.01 0.01 0.01\n')
            proto.write(boundingLevel * indent + '}\n')

        if boundingLevel == level + 4:
            proto.write((level + 3) * indent + ']\n')
            proto.write((level + 2) * indent + '}\n')
            boundingLevel = level + 2
    if boundingLevel == level + 2:
        proto.write((level + 1) * indent + ']\n')
        proto.write(level * indent + '}\n')

def URDFShape(proto, link, level):
    indent = '  '
    shapeLevel = level
    transform = False
    group = False
    if len(link.visual) > 1:
        proto.write(level * indent + 'Group {\n')
        proto.write((level + 1) * indent + 'children [\n')
        shapeLevel = level + 2
        group = True

    for visualNode in link.visual:
        if visualNode.position != [0.0, 0.0, 0.0] or  visualNode.rotation[3] != 0:
            proto.write(shapeLevel * indent + 'Transform {\n')
            proto.write((shapeLevel + 1) * indent + 'translation ' + str(visualNode.position[0])
                        + ' ' + str(visualNode.position[1]) + ' ' + str(visualNode.position[2]) + '\n')
            proto.write((shapeLevel + 1) * indent + 'rotation ' + str(visualNode.rotation[0])
                        + ' ' + str(visualNode.rotation[1]) + ' ' + str(visualNode.rotation[2]) + ' ' + str(visualNode.rotation[3]) +'\n')
            proto.write((shapeLevel + 1) * indent + 'children [\n')
            shapeLevel = shapeLevel + 2
            transform = True

        proto.write(shapeLevel * indent + 'Shape {\n')
        proto.write((shapeLevel + 1) * indent + 'appearance Appearance {\n')
        if visualNode.material.color.red != 0 or visualNode.material.color.green != 0 or visualNode.material.color.blue != 0:
            proto.write((shapeLevel + 2) * indent + 'material Material {\n')
            proto.write((shapeLevel + 3) * indent + 'diffuseColor ' +
                        str(visualNode.material.color.alpha * visualNode.material.color.red + 1 - visualNode.material.color.alpha)
                        + ' ' + str(visualNode.material.color.alpha * visualNode.material.color.green + 1 - visualNode.material.color.alpha)
                        + ' ' + str(visualNode.material.color.alpha * visualNode.material.color.blue + 1 - visualNode.material.color.alpha) + '\n')
            proto.write((shapeLevel + 2) * indent + '}\n')
        if visualNode.material.texture != "":
            proto.write((shapeLevel + 2) * indent + 'texture ImageTexture {\n')
            proto.write((shapeLevel + 3) * indent + 'url ["' + visualNode.material.texture + '"]\n')
            proto.write((shapeLevel + 2) * indent + '}\n')
        proto.write((shapeLevel + 1) * indent + '}\n')

        if visualNode.geometry.box.x != 0:
            proto.write((shapeLevel + 1) * indent + 'geometry Box {\n')
            proto.write((shapeLevel + 2) * indent + ' size ' + str(visualNode.geometry.box.x)
                        + ' ' + str(visualNode.geometry.box.y) + ' ' + str(visualNode.geometry.box.z) + '\n')
            proto.write((shapeLevel + 1) * indent + '}\n')

        elif visualNode.geometry.cylinder.radius != 0:
            proto.write((shapeLevel + 1) * indent + 'geometry Cylinder {\n')
            proto.write((shapeLevel + 2) * indent + 'radius ' + str(visualNode.geometry.cylinder.radius) + '\n')
            proto.write((shapeLevel + 2) * indent + 'height ' + str(visualNode.geometry.cylinder.length) + '\n')
            proto.write((shapeLevel + 1) * indent + '}\n')

        elif visualNode.geometry.sphere.radius != 0:
            proto.write((shapeLevel + 1) * indent + 'geometry Sphere {\n')
            proto.write((shapeLevel + 2) * indent + 'radius ' + str(visualNode.geometry.sphere.radius) + '\n')
            proto.write((shapeLevel + 1) * indent + '}\n')

        elif visualNode.geometry.trimesh.coord != []:
            proto.write((shapeLevel + 1) * indent + 'geometry IndexedFaceSet {\n')
            proto.write((shapeLevel + 2) * indent + 'coord Coordinate {\n')
            proto.write((shapeLevel + 3) * indent + 'point [\n')
            for value in visualNode.geometry.trimesh.coord:
                proto.write(shapeLevel * indent + str(value[0] * visualNode.geometry.scale[0])
                            + ' ' + str(value[1] * visualNode.geometry.scale[1]) + ' ' + str(value[2] * visualNode.geometry.scale[2]) + '\n')
            proto.write((shapeLevel + 3) * indent + ']\n')
            proto.write((shapeLevel + 2) * indent + '}\n')
```

```
            proto.write((shapeLevel + 2) * indent + 'coordIndex [\n')
            for value in visualNode.geometry.trimesh.coordIndex:
                proto.write(shapeLevel * indent + str(value[0]) + ' ' + str(value[1]) + ' ' + str(value[2]) + ' -1\n')
            proto.write((shapeLevel + 2) * indent + ']\n')

            if visualNode.geometry.trimesh.texCoord != []:
                proto.write((shapeLevel + 2) * indent + 'texCoord TextureCoordinate {\n')
                proto.write((shapeLevel + 3) * indent + 'point [\n')
                for value in visualNode.geometry.trimesh.texCoord:
                    proto.write(shapeLevel * indent + str(value[0]) + ' ' + str(value[1]) + '\n')
                proto.write((shapeLevel + 3) * indent + ']\n')
                proto.write((shapeLevel + 2) * indent + '}\n')

                proto.write((shapeLevel + 2) * indent + 'texCoordIndex [\n')
                for value in visualNode.geometry.trimesh.texCoordIndex:
                    proto.write(shapeLevel * indent + str(value[0]) + ' ' + str(value[1]) + ' ' + str(value[2]) + ' -1\n')
                proto.write((shapeLevel + 2) * indent + ']\n')

            proto.write((shapeLevel + 2) * indent + 'creaseAngle 1\n')
            proto.write((shapeLevel + 1) * indent + '}\n')

        proto.write(shapeLevel * indent + '}\n')
        if transform:
            proto.write((shapeLevel - 1) * indent + ']\n')
            proto.write((shapeLevel - 2) * indent + '}\n')
            shapeLevel = shapeLevel - 2
    if group:
        proto.write((shapeLevel - 1) * indent + ']\n')
        proto.write((shapeLevel - 2) * indent + '}\n')


def URDFJoint(proto, joint, level, parentList, childList, linkList, jointList):
    indent = '  '
    if joint.type == 'revolute' or joint.type == 'continuous':
        proto.write(level * indent + 'DEF ' + joint.name + ' HingeJoint {\n')
        proto.write((level + 1) * indent + 'jointParameters HingeJointParameters {\n')
        proto.write((level + 2) * indent + 'axis ' + str(joint.axis[0]) + ' ' + str(joint.axis[1]) + ' ' + str(joint.axis[2]) + '\n')
        proto.write((level + 2) * indent + 'anchor ' + str(joint.position[0]) + ' ' + str(joint.position[1]) + ' ' + str(joint.position[2]) + '\n')
        proto.write((level + 2) * indent + 'dampingConstant ' + str(joint.dynamics.damping) + '\n')
        proto.write((level + 2) * indent + 'staticFriction ' + str(joint.dynamics.friction) + '\n')
        proto.write((level + 1) * indent + '}\n')
        proto.write((level + 1) * indent + 'device RotationalMotor {\n')
    elif joint.type == 'prismatic':
        proto.write(level * indent + 'DEF ' + joint.name + ' SliderJoint {\n')
        proto.write((level + 1) * indent + 'jointParameters JointParameters {\n')
        proto.write((level + 2) * indent + 'axis ' + str(joint.axis[0]) + ' ' + str(joint.axis[1]) + ' ' + str(joint.axis[2]) + '\n')
        proto.write((level + 2) * indent + 'dampingConstant ' + str(joint.dynamics.damping) + '\n')
        proto.write((level + 2) * indent + 'staticFriction ' + str(joint.dynamics.friction) + '\n')
        proto.write((level + 1) * indent + '}\n')
        proto.write((level + 1) * indent + 'device LinearMotor {\n')
    elif joint.type == 'fixed':
        for childLink in linkList:
            if childLink.name == joint.child:
                URDFLink(proto, childLink, level, parentList, childList, linkList, jointList, True, joint.position, joint.rotation)
        return

    elif joint.type == 'floating' or joint.type == 'planar':
        print joint.type + ' is not a supported joint type in Webots'
        return

    proto.write((level + 2) * indent + 'name "' + joint.name + '"\n')
    if joint.limit.velocity != 0.0:
        proto.write((level + 2) * indent + 'maxVelocity ' + str(joint.limit.velocity) + '\n')
    if joint.limit.lower != 0.0:
        proto.write((level + 2) * indent + 'minPosition ' + str(joint.limit.lower) + '\n')
    if joint.limit.upper != 0.0:
        proto.write((level + 2) * indent + 'maxPosition ' + str(joint.limit.upper) + '\n')
    if joint.limit.effort != 0.0:
        if joint.type == 'prismatic':
            proto.write((level + 2) * indent + 'maxForce ' + str(joint.limit.effort) + '\n')
        else:
            proto.write((level + 2) * indent + 'maxTorque ' + str(joint.limit.effort) + '\n')
    proto.write((level + 1) * indent + '}\n')

    proto.write((level + 1) * indent + 'endPoint Solid {\n')
    proto.write((level + 2) * indent + 'translation ' + str(joint.position[0]) + ' ' + str(joint.position[1]) + ' ' + str(joint.position[2]) + '\n')
    proto.write((level + 2) * indent + 'rotation ' + str(joint.rotation[0]) + ' ' +
                str(joint.rotation[1]) + ' ' + str(joint.rotation[2]) + ' ' + str(joint.rotation[3]) + '\n')
    proto.write((level + 2) * indent + 'physics Physics {\n')
    proto.write((level + 3) * indent + 'density -1\n')
    proto.write((level + 3) * indent + 'mass 0.001\n')
    proto.write((level + 3) * indent + 'inertiaMatrix [1 1 1 0 0 0]\n')
    proto.write((level + 3) * indent + 'centerOfMass [0.0 0.0 0.0]\n')
    proto.write((level + 2) * indent + '}\n')
    proto.write((level + 2) * indent + 'children [\n')
    for childLink in linkList:
        if childLink.name == joint.child:
            URDFLink(proto, childLink, level + 3, parentList, childList, linkList, jointList)
    proto.write((level + 2) * indent + ']\n')
    proto.write((level + 1) * indent + '}\n')
    proto.write(level * indent + '}\n')
```