Programming of a Nao humanoid in Gazebo using Hierarchical FSM

Borja Menéndez, Rubén Salamanqués, José María Cañas

***Abstract*—The interest in humanoid robots has increased significantly in the last years. International contests like RoboCup or DARPA Robotic Challenge foster the research in humanoids and an increasing number of conferences focus on them. The intelligence of the humanoid robot mainly lies on its software. There are several tools that help in the development of intelligent software for them. In this paper we present two tools inside our JdeRobot framework. First, the support for the Nao humanoid in the latest releases of Gazebo simulator. Second, a visual programming application, named VisualHFSM, which provides Hierarchical Finite State Machines to program the robot behaviors. Both have been experimentally validated using VisualHFSM to generate example behaviors in the simulated Nao.**

***Index Terms*—Humanoid, simulators, middleware, visual languages, finite state machines**

I. Introduction

Humanoids is a field of growing interest in robotics. Prototypes such as the Honda Asimo or the Fujitsu HOAP3 are the basis for many research efforts, some of them designed to replicate human intelligence and manoeuvrability. Their appearance, similar to people, facilitates their acceptance and natural interaction with humans as a personal assistant in the field of service robotics. As a representative sample, the functionality achieved in the Asimo humanoid has progressed significantly in recent years, allowing it to run, climb stairs, push carts and serve drinks. Moreover, the recent DARPA Robotics Challenge has chosen a humanoid platform (first in simulation, second with the real Atlas robot) for its contest testing robot usefulness for hazardous activities in disaster response operations.

Within this field the humanoid robot Nao, manufactured by Aldebaran Robotics, has burst recently. Since 2008 it is the official platform of the Standard Platform League (SPL) in the international robotics competition RoboCup. Due to its low price, compared to other humanoids, their prominence and its good features such as an SDK easy to work with, the number of users of this robot is growing in the robotics community, both for use in research and teaching.

Most of the robot intelligence lies on its software. Its functionality resides in its programming, in the software that manages hardware resources like sensors and actuators. There is no universally standardized methodology to program robots. In the last few years the robotics community has begun to apply software engineering methodologies to its field, making more emphasis in code reuse, distributed software design, etc. Several robot programming platforms that simplify the development of applications have emerged. These platforms (1) provide a more or less portable hardware access layer (HAL); (2) offer a concrete software architecture to the applications; (3) include tools and libraries with already ready-to-use functionality and building blocks. Many of the emerged platforms are component oriented, such as ROS, Orca, Microsoft Robotics Studio, RoboComp, JdeRobot, etc.

One significant tool for development of robot software are simulators. They provide virtual environments and emulate the sensor observations and the effects of the orders to the actuators. They allow testing and debugging before putting the software on the real robot, making easier and cheaper to work with teams of robots. They also provide ground truth data to test perception software. Current available simulators have gained in realism, incoporate noise in sensors and actuators, provide physics engine and 3D simulation, including cameras.

There are many simulators available, some on commercial bases like Webots or the one in Microsoft Robotics Developer Studio[1] [10], and others from the open source robotics community. One interesting recent simulator is MORSE[2] [6], aimed at educational environments. Perhaps the most widespread ones are `Stage` [7], a multirobot bi-dimensional simulator, and `Gazebo` [11]. In particular Gazebo[3] is a 3D simulator that provides a rich environment for developing and testing multirobot systems quickly and simulates realistically even cameras (Figure 1). It was born within the project Player/Stage/Gazebo [19], but recently WillowGarage and Open Source Robotics Foundation fostered its development, as a project on its own. In addition, DARPA chose Gazebo as the platform for the first stage of its Darpa Robotic Challenge.

Another interesting tool that can be used for robotic software development is the use of automatons to generate robot behaviors. Finite state machines (FSM) have been succesfully used to symbolize the robot behaviors, representing them in a compact and abstract form. With FSM the behavior is defined by a set of *states*, each of which performs a particular task. The system can then switch from one state to another through *transitions*, which are conditions of state change or stay depending on certain events or sensor conditions that may happen, both internal or external. FSMs provide one smart way to organize the control code and perception on-board a mobile robot. They have been

Universidad Rey Juan Carlos.

E-mails: b.menendez.moreno@gmail.com, jorge_bermejo0817@hotmail.com, jmplaza@gsyc.es

[1] `http://www.microsoft.com/robotics`
[2] `http://www.openrobots.org/wiki/morse`
[3] `http://gazebosim.org`

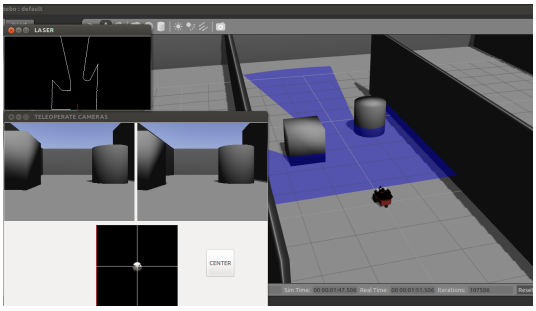

Fig. 1: Gazebo 1.8.1 showing the a Pioneer robot with laser and cameras

explored in research and also incorporated in recent robotic frameworks with tools that let the programmer to focus on the behavior logic more than in implementation details. Most of the code is then generated automatically by a software tool from the abstract description of the FSM. This diminishes the chance of bugs, reduces the development time and allows the programming of complex behaviors in a robust way.

In this paper we present two steps forward from our previous works: the support of the Nao humanoid in latest releases of Gazebo simulator [4] and its programming using a visual tool with finite state machines that we have developed inside the JdeRobot framework, extending it to manage hierarchical FSM.

The reminder of this paper is organized as follows. In the second section we review related works on frameworks for Nao programming and other tools to create hierarchical finite state machines in robots. The third section presents in detail our visual programming tool, VisualHFSM. The fourth section describes the developed support for the Nao robot in Gazebo, its sensors and actuators, and its use inside the JdeRobot framework through standard ICE interaces. The fifth section describes experiments with robot behaviors developed using the VisualHFSM for programming a simulated Nao. Finally, conclusions are summarized.

II. Related works

The reference simulator for Nao is Webots from Cyberbotics, which is compatible with the Nao SDK. Webots[5] is a proprietary software to simulate mobile robots mainly used for educational purposes. Like Gazebo, Webots uses a physics engine to reproduce realistic behaviors. With it, you can simulate in a realistic way several Nao robots, visualizing its cameras and having access to its sensors and actuators.

Another example of using simulators for Nao in the SPL Robocup is SimRobot. This is an open source simulator developed by the B-Human team, led by Thomas Röfer [12], developed at the Universität Bremen and the German Research Center for Artificial Intelligence. It is being used for further research on autonomous robots.

The Nao robot is also supported by Robot Operating System (ROS[6]). ROS provides an API that allow roboticists to have full control of the real robot, so you can run ROS inside the Nao using this API. ROS is supporting the simulated Nao by Rviz[7], the visualization tool for this platform, licensed under BSD and Creative Commons, in which you can create scenarios for the Nao robot.

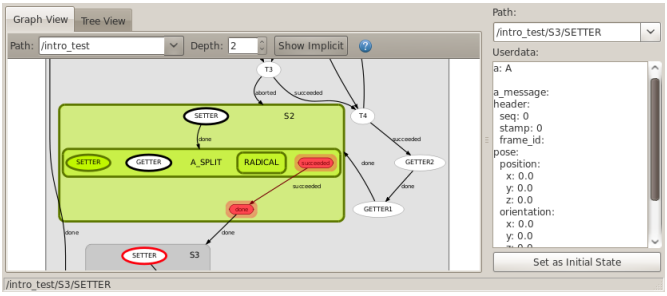

Fig. 2: An instance of SMACH-viewer

Regarding finite state machines, ROS is also gaining prestige with its platform-independent SMACH[8]. This tool is a task-level architecture for rapidly creating complex robot behaviors. At its core, SMACH is a ROS-independent Python library to build hierarchical finite state machines. To develop a hierarchical finite state machine you have to write the code needed to create and describe each state and transition, it is not a visual tool. The package also comes with the SMACH-viewer (Figure 2), a tool that shows the developed finite state machine at runtime. In that tool, we can see either a graph or a tree view of the current automaton, but not both at the same time. It also shows every state and transition, as well as the active state and a debug window where we can see data of the active state.

One of the most powerful frameworks that use HFSM in robotics is MissionLab, developed in Georgia Tech by the professor R. Arkin. This environment includes a graphical editor of *configurations* (CfgEdit) [15] as a tool, similar to automaton, that allows to specify missions with its states, transitions, etcetera. It allows to generate applications following the AuRA architecture, developed by the same group. In order to represent the automaton they developed their own language, the Configuration Description Language.

A more recent example of FSM is the automaton editor inside the component-oriented platform RoboComp, from Universidad de Extremadura. For instance, in [3], they used it to program the behavior of a forklift. Another sample is the behaviors graphical editor Gostai Studio [8], inside the Urbi platform. This graphical editor generates *urbiscript* code as its output, includes time-execution visualization of the state of the automaton, allows to stop and continue the execution of the generated automaton and offers the possibility of creating hierarchical finite state machines.

In the RoboCup scenario finite state machines are frequently used to generate behaviors for the standard SPL league humanoid robots. Several teams use the tool and language XABSL [14; 17] to specify behaviors, around the influential

[4] At the time of writing this paper the latest release is 1.8
[5] http://www.cyberbotics.com
[6] http://www.ros.org
[7] http://www.ros.org/wiki/rviz
[8] http://www.ros.org/wiki/smach

German team B-Human. In addition, the TeamChaos team [9] used an HFSM tool to handle hierarchical finite state machines for its architecture ThinkingCap, allowing even behavior hot-edition in each state. In the SPIteam[9] team a tool named Vicode is used to generate finite state machines for BICA architecture.

Another example of the automaton power is the use for programming the intelligence of automatic players in videogames. This scenario is simpler than real robotic because much of the player perception is obtained looking up some videogame variable. For instance, in the successful Halo 2 of Bungie automaton trees were used to deploy more than one hundred different behaviors.

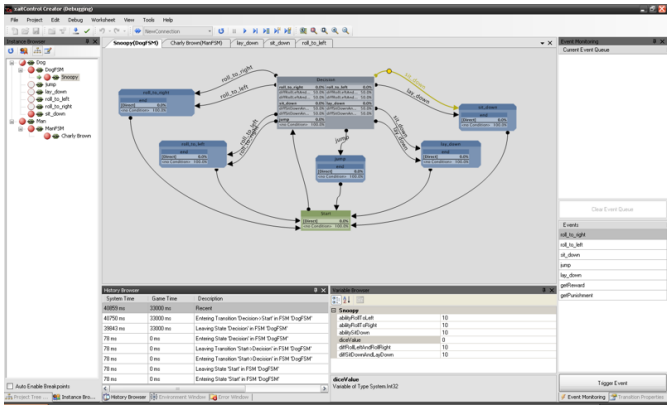

Fig. 3: An instance of xaitControl

Xait[10] enterprise commercializes tools that facilitates automaton programming to videogames developers, as its xaitControl (Figure 3). This tool allows the programmer to easily develop behaviors with hierarchical finite state machines. It has a main canvas in which the automaton is displayed, a tree view on the left shows the created structure and other panels show different information about auxiliary procedures, execution flow control, etc.

III. Hierarchical visual FSM tool

We have created the VisualHFSM tool inside JdeRobot[11] framework for the programming of robot behaviors using finite state machines with hierarchy (HFSM - Hierarchical Finite State Machines) to generate components for this component-oriented framework. It represents the robot's behavior graphically on a canvas composed of states and transitions. The source code to be executed at each state or when checking each transition can be also introduced. This tool decreases the development time for new applications, providing the developer with a more abstract visual language. It also increases the quality of these applications, automatically generating most of their code and optimizing the code to use it as a JdeRobot component. The tool allows the engineer to focus on specific parts of his application, automatically generating the rest, getting a more robust code, less prone to failure.

Current release is based on a previous work without hierarchy [20]. In that release, the functionality was almost the same, but it did not include more than one-level automaton, fact that is a matter of usability and readability.

VisualHFSM is divided into two parts: the graphical editor and the automatic code generator. First, the graphical editor displays the main window of the tool, where all the functionality is located to create the automaton structure. It also contains internal structures that provide functionality to the interface and saves in an XML file all the information related to the automaton structure and the application.

Second, inside the code generator there are two different parts: (1) the file generator, which is supported by the XML file generated by the editor and in a JdeRobot template.

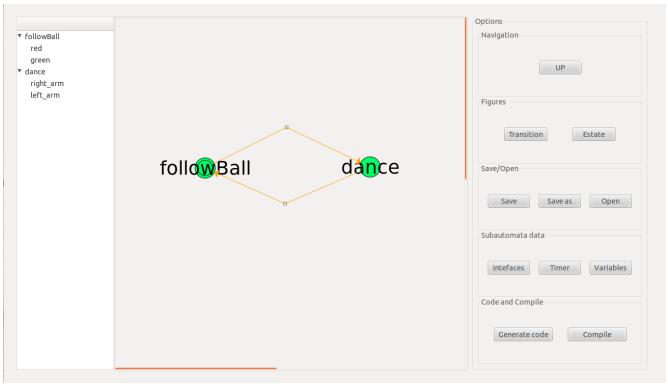

Fig. 4: Graphical editor of the VisualHFSM tool including the sub-automaton of one of its states

A. Graphical editor

The graphical editor allows the user to visually represent, edit and add states and transitions in a clear an simple way (Figure 4). The component is represented by a finite state machine in a canvas, where every element or state and the transitions that connect each other are visualized. It allows to manipulate states (create, remove, move, name...) and to define the behavior in every state. It is also possible to set the conditions of stay and the transitions between states.

The GUI is divided into three parts: the tree view at the left, the canvas at the center and the action buttons at the right. The graphical editor saves in an XML file all the features of the developed component: the structure of the automaton, the characteristics of each node and transition, etcetera.

The tree view is the area where you can see the hierarchical tree of the generated automaton. It has a drop down menu where you can navigate through the created hierarchical automaton. The tree view allows to double-click in a state and represent its underlying sub-automaton in the main canvas.

The canvas is the area where the automaton is drawn, showing the name of each element, either state or transition. States are represented as blue circles if the state has no sub-automatons and as green circles if the state has a sub-automaton. It also marks with a double circle if the state is

[9] http://www.spiteam.org
[10] http://www.xaitment.com
[11] http://jderobot.org

the initial of the showed automaton. For each state we can do a few actions: rename it; edit it, allowing you to change the code of the selected state; mark it as initial; copy it, allowing to paste the selected state into another or the same automaton; and delete it. Every state can be connected to another by an unlimited number of transitions. An state can also be connected to itself by an autotransition. Transitions are visualized as an arc and correspond to the conditions of stay or change. These conditions can be temporary or conditionals. States and transitions are associated with a name defining its specific functionality, entered from the graphical interface. The operations associated to transitions are rename, edit and delete it, with the same functionality as mentioned before.

The button area is the area where the buttons are showed. They are structured in five groups: navigation, where you can go up to see the parent automaton; object or figure buttons, to create states and transitions; save and open buttons, to save the automaton created or load an existing one; edit automaton data, where you define interfaces, timing, variables and functions of the showed automaton; and generate code and compile buttons.

B. XML

As mentioned before, the graphical editor allows to save an XML file in disk as nonvolatile support of all the application properties. The equivalent component in textual language will be generated from this file. The XML file allows to save the component, regardless it is finished or not, to its later modification, editing it in another computer or even loading it in later tool versions.

The XML file saves for each sub-automaton different features. First of all, it saves a number as the automaton id and another number as the automaton parent id. This unique identifier allows to distinguish and establish the dependency relationship between states. States are also saved with its own identifier, the point coordinates in the canvas, its name, its child id (if it does not have any, 0 is written), its own code and its outgoing transitions. For every transition the position in the canvas is saved, its name and its destiny as the state id where the transition ends. It also saves the automaton attributes as iteration time, sensor and actuator interfaces used, variables and functions.

C. Automaton template

In order to materialize and translate all of the states and transitions to textual code, a template is used. It has two parts: the control thread and the graphics thread. Each of them executes periodic iterations in concurrent threads. The tool we have built automatically fills the automaton template and embed the code of states and transitions in it (code, names, changes between states, etc.).

To implement the multilevel hierarchical FSM, the control is subdivided into different concurrent threads belonging to each automaton. For each automaton thread it executes different actions in a constant loop at a refresh rate of 10 Hz, that is, every 100 ms, as a design decision. First of all, it has an evaluation switch in which the transitions are evaluated. If the transition is met, the state is changed to the new state. Then, it has an action switch in which the code of the state is executed.

This is the basic template used for the first automaton, but for its children it has one modification. Before executing the code embedded of the sub-automaton, each thread checks whether the parent state is active or not. If not, it does not execute its own code.

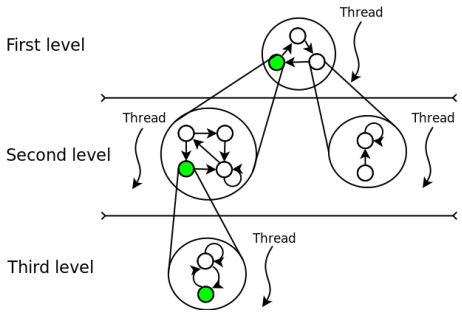

Fig. 5: Multi-thread automaton template

The automaton template will work as a multi-threaded execution in which every thread is an automaton waiting for being executed. This way, with a multi-threaded execution, each automaton is ready for execution and it takes no time to start running. We can see in Figure 5 an example of a multi-threaded automaton template; the active state is shown in green. As the tool builds hierarchical finite state machines, a sub-automaton will be executed only when its parent state is active. If the parent state transits to another state the tool saves the state in which its child sub-automaton has been stopped. To save that information, it creates another states called *ghost* states. If the parent transits to another state, the child changes to its corresponding *ghost* state. Later on, when the parent recovers the previous state, the child starts its execution where it stopped before, changing its state from *ghost* to regular. It makes complete sense with a three or more level automaton.

This behavior is summarized in Figure 6. In that figure, every state has its own ghost state except in the first level of the automaton. The active state is green coloured, either the regular state or the ghost state. At first, we have the active state running with its respective child and grandchild. In the second snapshot, the previous active state of the second level automaton loses the control, so there is a new state as active, and its child, the third level automaton, has the state as *ghost*. In the third snapshot, the active state of the first level automaton loses the control and then the second level automaton has the state as *ghost*, maintaining the *ghost* state in the third level.

The graphical thread helps the developer to debug its automaton. It iterates in an infinite loop, as the execution threads of each automaton, at a refresh rate of 10 Hz, too. It shows in a window which is the active state running on the hierarchical finite state machine, sensor data and internal structures at runtime. The automatic code generator general schema is showed in Figure 7.

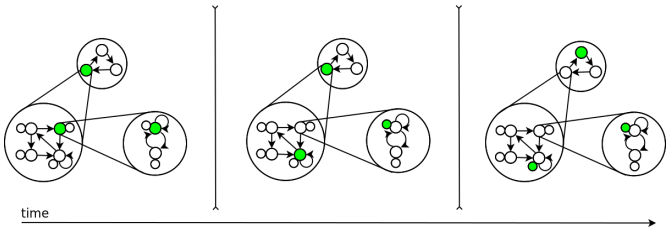

Fig. 6: Ghost states of the same automaton at three different times

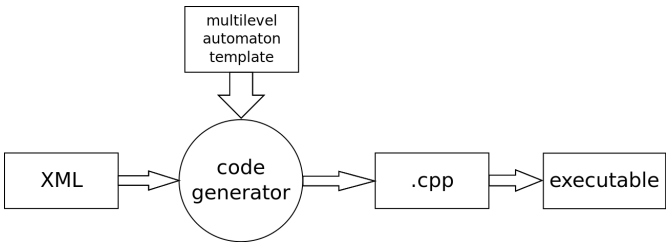

Fig. 7: Automatic code generator of VisualHFSM tool

IV. Nao support in Gazebo

There are two building blocks when programming the Nao support in Gazebo: create the model in Gazebo and plugins that allows external applications to access to its sensors and actuators. Every plugin work as a dynamic library (.so) that is loaded at the start of Gazebo. A previous work was developed before, but for Gazebo 0.9 [1]. Since that version, the simulator has deeply changed and it i-s quite different, providing a totally different API and methods to move a simulated robot.

A. The model

The first step to bring support to the Nao humanoid in Gazebo is to create a model inside the simulator with basic objects: links, joints and sensors. In addition to assembling all the pieces, the simulator offers Simulation Description Format (SDF), a way to write XML files defining the visual properties of our robot. The last version of this SDF (1.4) works with versions of Gazebo up to 1.6.

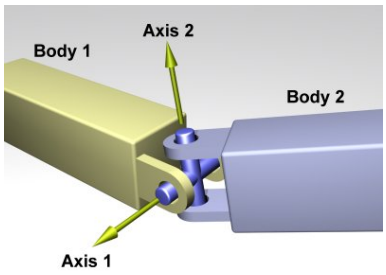

Fig. 8: Two blocks connected by a hinge

To build the robot we have to join some basic predefined blocks like hexahedrons, spheres, cylinders, etc (Figure 8). These blocks can be given geometrical size and mass necessary for the particular model, as well as its location in the world, its inertia matrix and many other features. These bodies are joined by hinges, which can give one or more directions of rotation. This provides our simulated model with different degrees of freedom. Every hinge in this model is built based on *revolute* joints, which commands the aperture of the hinge in the specified axis. The arm and the camera will be detailed as representative samples.

First of all, the humanoid's arm in Gazebo is composed by humerus, ulna, radius and hand. The humerus is modelled as a rectangular hexahedron with dimensions of 60x60x105mm and a mass of 157,77g; the part of ulna, radius and hand is joined in the same structure and is developed building a rectangular hexahedron with dimensions of 60x60x55,95mm and a mass of 77,61g (Figure 9).

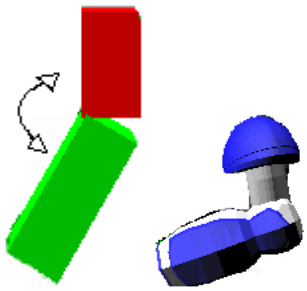

Fig. 9: An arm as connected blocks (left) and with skin (right)

These two bodies are joined by an elbow hinge. This hinge gives the regular elbow's degree of freedom, which allows to bend and stretch the arm. Besides, it allows another additional degree of freedom available in the real Nao, one that allows you to rotate the forearm with respect to the humerus. To achieve these two degrees of freedom at the elbow we used two hinge joints in the model. Gazebo can not join two bodies with more than one hinge at the same time, so we introduced another dummy body between the forearm and the upper arm. It is the only way to get that turning the forearm rotating with the other elbow hinge that allows us to stretch and bend the arm. This new body created has a size small enough to not be visually appreciated and also a mass small enough to be negligible in the overall calculation of the mass of our simulated robot.

To add a skin is as simple as specify the path to a Collada[12] file (.dae) in the `visual` tag. The Nao robot skins were built with Blender and added to this model.

The humanoid head is modelled as an sphere, with its skin, and incorporates two camera sensors, as in the real robot. To do that, it is necessary to add two sensors in the model. This sensor is created based on the SonyVID30 camera, sensor already supported by Gazebo. This camera provides images with a size of 320x240 pixels, with an horizontal field of view (HFOV) of 60 degrees and a refresh of 10 frames per second. Once incorporated to the head, we developed the basic programming interface which allows to get data from

[12] `https://collada.org`

the camera to see what the Nao robot in Gazebo is watching every moment.

In order to move the head, we have developed a neck hinge. The neck has two degrees of freedom: one for the pitch (tilt), that is, the vertical movement of the head, and one for the yaw (pan), that is, the horizontal movement of the head. As explained before, this is done joining the body with the head through a new little dummy body with small size and mass.

Following these steps we designed and built every part of the simulated humanoid Nao: the arms, legs, body and head. In Figure 10, we can see at the left the humanoid without textures, with mechanical blocks, and at the right the appearance of the same robot with skins incorporated, with a much more realistic appearance. As well as the mechanical assemble of the blocks, this model adds a set of basic programming interfaces for each block: actuators in its joints and sensors, including its cameras.

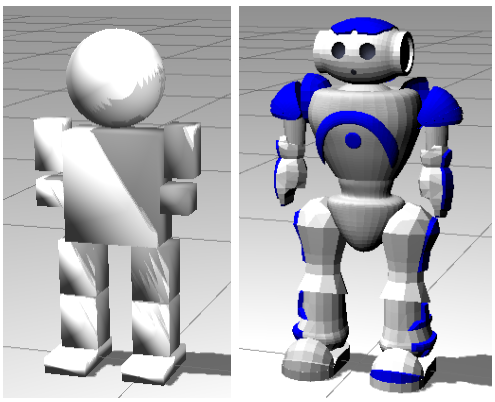

Fig. 10: Nao with the basic blocks without skin (left) and with skin (right)

Every part of the simulated Nao is designed following the Aldebaran Robotics documentation, including the center of masses, inertial matrices or the aperture range of the hinges, getting a realistic behavior at the simulation.

B. Gazebo plugin

Once the Gazebo humanoid model was built, developers can program applications that use its sensors and command movements to its actuators. The simulator provides an API with different modules in order to have a realistic simulation, such as methods to do operations with quaternions, get the position of a link, simulate a depth camera sensor or give an easy way to have a PID controller in the simulation, among others.

Gazebo provides a low level API in which a plugin can act moving bodies, while Gazebo plugins provides a high level API that can be connected to another JdeRobot components (Figure 11). At least, a Gazebo plugin may have at least three elements: the line in which you register the plugin, the *Load* method in which you load every element of the SDF file you want to take over control and the *OnUpdate* method that will run iteratively in a loop to do the work.The Gazebo plugin is developed following these lines:

First of all, as explained before, we register the model. Then, the model will be loaded. In this step, we have to find the links

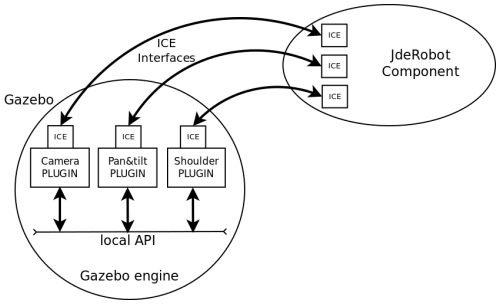

Fig. 11: Low level and high level API in a Gazebo plugin

and joints we want to provide movement. Finally, the code will enter in the *OnUpdate* method, where we have to check the position of the hinge with its encoders in order to move it to the position we ordered. Encoders and motors are offered as standard interfaces, updating them in every loop and offering them when the user chooses it.

The plugin provides access to sensors and actuators of the robot. To do this in an easy way, JdeRobot provides standard interfaces with the ICE middleware. This fact makes all JdeRobot components can easily communicate each other. There are ICE interfaces predefined in JdeRobot, so to provide this access we wrote methods that can connect other components to obtain the sensor values and give ability to move the actuators.

The simulated robot's neck is special: it is the only hinge that allows movement both in position and in speed. With this characteristic we can move the neck to an specific position or command it to move to an specific speed in the axis we command it. This feature allows the programmer to do experiments such as follow-objects.

V. Experiments

In order to validate the VisualHFSM tool and the structure to give support to Nao robot in Gazebo, we have performed three different experiments. The first one tests some of the simulated actuators, the robot dances using its arms. The second one tests the main Nao sensor, its camera, and performs a visual control to let the robot track a color ball with its head. The third one validates VisualHFSM integrating the previous behaviors in just one hierarchical finite state machine.

A. Dancing

In the very first experiment the robot dances with its arms. The experiment consists on moving the right and the left arm at the same time, commanding different positions (Figure 12).

To do this behavior, the Nao has different joint hinge controls. This control is a position based to each hinge and stops its movement when the position is reached. The developed component that moves the robot gets the position of the hinge through the odometric in the plugin and, when it reaches the desired position, commands the hinge to a new position with the position sequence the simulated robot receives in each step. The simulated robot is in a constant movement, dancing with its arms.

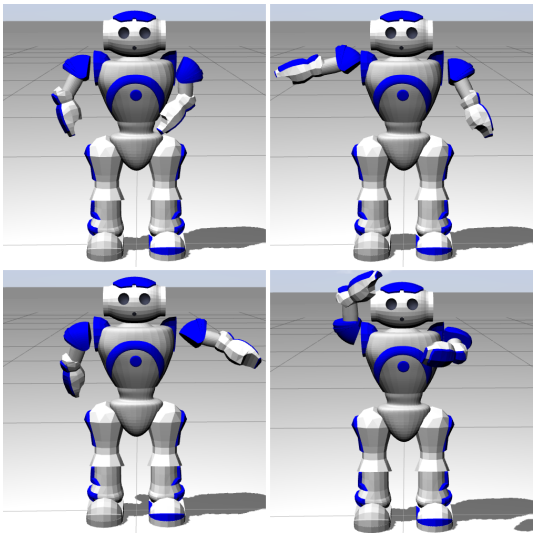

Fig. 12: The Nao robot dancing in Gazebo simulator

B. Follow-ball

The second experiment is as simple as following a ball with its upper camera. A green ball is moved following a 3D trajectory in Gazebo and the simulated Nao moves its head vertically and horizontally to keep the ball in the middle of the field of view (Figure 13).

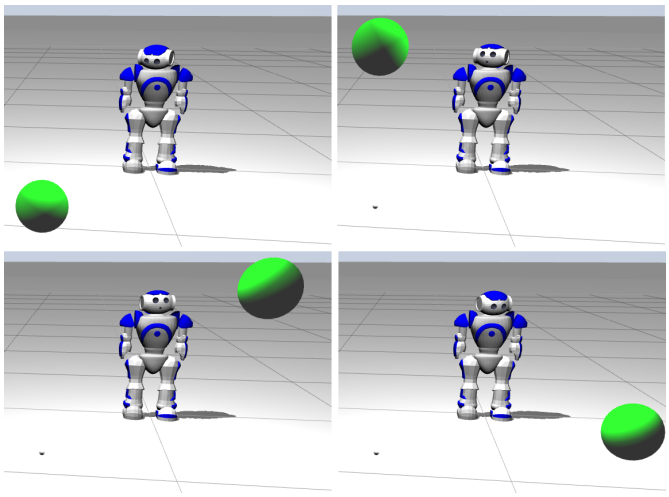

Fig. 13: The Nao robot following a ball in Gazebo simulator

The green ball movement is performed with a plugin in order to move it through the world. This plugin is developed similarly to the plugins of the robot, but it uses two different hinges. The first (*revolute*) gives horizontal movement and the second (*prismatic*) gives vertical movement. In order to move it, we developed an ad-hoc JdeRobot component which commands the position to the ball and waits until the position is reached; once reached, it sends a new position again.

To generate the tracking behavior, the simulated robot receives the image, does an RGB colour filter in order to know where the ball is and calculates the center of the ball. Second, it has a closed loop control with a proportional controller. It calculates the error between the center of the ball and the center of the image, commanding the speed movement to its neck. The Nao has a neck speed control. The software sends speed commands of how fast the tilt and pan hinges have to move its motors, so that it is not given a position, but a speed command. The more error it has, the more speed it commands in each direction (x and y axis). The robot moves its neck following the ball as fast as possible.

C. Follow-ball and dancing with VisualHFSM

The third experiment has been developed with VisualHFSM and it combines the previous dancing and tracking behaviors. The hierarchical finite state machine is showed in Figure 14. This is a toy example of hierarchy created with VisualHFSM. We can see at the left the tree view with the first level of the automaton and its children, showing in different *windows* the children of each state.

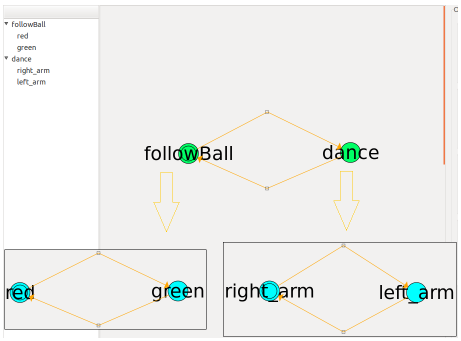

Fig. 14: Hierarchical automaton developed with VisualHFSM for the simulated Nao

In the first level of the automaton in which there are two states with one transition per state: the `followBall` state and the `dance` state. Each state does nothing by itself, it just gives the control to its respective child sub-automaton. The transitions are defined in a temporary way: every 10 seconds the working state is changing, starting on `followBall` state. This way, with a transition, the old state loses the control of the robot, giving this control to the new state and *switching off* its respective sub-automaton. When the control returns to that state, its child is *switched on*, recovering its previous state.

The sub-automaton child of `followBall` has two states: `red` and `green`. In each state, the robot follows the red or the green ball respectively. The transitions are defined in a temporary way too, but now the states are running for eight seconds.

The sub-automaton child of `dance` has two states: `right` and `left`. In each state, the robot moves its right or its left arm respectively. The transitions are defined in a conditional way: if the arm has reached 10 different positions, the control is passed to the other state.

The entire video with this combined behavior running is shown at JdeRobot[13] web page.

[13] `http://bit.ly/nao_automata`

VI. Conclusions

Nao from Aldebaran is an example of the growing interest in humanoid robots in the robotics community. It is the official platform of the Standard Platform League of the Robocup since 2008, and an increasing number of centres use it in their research or teaching. One typical simulator for this robot is Webots from Cyberbotics, but it is commercial and closed source. We have developed the support for the Nao humanoid in the open source Gazebo simulator. Gazebo is becoming a de facto standard since the ROS middleware chose it as its reference simulator. More recently the DARPA chose it as the platform for the first stage of the DARPA Robotics Challenge.

The developed support consists of two parts. First, creating a model of the Nao robot inside Gazebo, with its mechanical pieces properly connected and with right sizes and weights, its sensors, hinges and realistic appearance. Second, developing a set of software plugins that link the low-level local API for sensors and motors inside the simulator with a set of high-level ICE interfaces that provide such data to external applications. These ICE interfaces allow the connection of distributed objects to the sensors and actuators of the simulated Nao, may be running in different computers. They are exactly the same that the `NaoServer` component in JdeRobot offers to access to the resources of the real Nao. With this design the robotic application can run without changes both in the real and in the simulated humanoid.

In addition, we have extended our VisualHFSM application to deal with multilevel FSM. It provides a visual programming tool to develop robot behaviors with hierarchical finite state machines. This tool provides a graphical user interface to show and manage the automaton. It also automatically generates the C++ source code from an XML description of the automaton, filling in a multithreaded template that we have created for any FSM. Its output is a JdeRobot C++ component. This tool allows a reliable and quick development of new robot behaviors in the abstract and powerful terms of states and transitions.

As preliminar experimental validation we have programmed example behaviors for the Nao in Gazebo using the VisualHFSM tool. The robot is able to combine several behaviors like following a color ball with the head and dancing in a sequence that uses temporary transitions and sensor conditions to switch from one state to another.

We intend to follow the work presented here in several lines. First, to use both tools in robotics teaching. With this JdeRobot+Gazebo+VisualHFSM environment, students can program the behavior of the simulated Nao humanoid and learn the foundations of robotics, visual perception, etc. Their feedback will help to improve the tools' usability. Second, to extend the VisualHFSM to generate source code for Naoqi middleware, not only for the JdeRobot framework. A different automaton template should be developed for that, and so the generated code would run seamlessly on the real Nao robot, equipped with the original manufacturer Naoqi framework. Finally, we would like the VisualHFSM to be used with several different robot platforms and to explore the integration of different languages like LUA or Python.

References

[1] J. Bermejo and J. Cañas. Soporte del robot humanoide nao en el simulador 3d gazebo. In *Proceedings of XIII Workshop on Physical Agents, WAF 2012*, pages 73–80, Santiago de Compostela, September 2012.

[2] S. Carpin, M. Lewis, J. Wang, S. Balarkirsky, and C. Scrapper. Usarsim:a robot simulator for research and education. In *Proceedings of the IEEE 2007 International Conference on Robotics and Automation, pp. 1400-1405*, 2007.

[3] R. Cintas, L. Manso, L. Pinero, P. Bachiller, and P. Bustos. Robust behavior and perception using hierarchical state machines: A pallet manipulation experiment. *Journal of Physical Agents*, 5(1):35–44, 2011.

[4] P.T. Cox and T.J. Smedley. Visual programming for robot control. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 217–224, sep 1998.

[5] Jeff Craighead, Robin Murphy, Jenny Burke, and Brian Goldiez. A survey of commercial open source unmanned vehicle simulators. In *Proceedings of the IEEE 2007 International Conference on Robotics and Automation, pp. 852-857*, 2007.

[6] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. Modular openrobots simulation engine: Morse. In *Proceedings of the IEEE ICRA*, 2011.

[7] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics ICAR-2003, pp. 317-323*, Coimbra (Portugal), 2003.

[8] Gostai. Gostai studio suite. http://www.gostai.com/products/studio/gostai_studio/, 2012.

[9] D. Herrero-Perez, F. Bas-Esparza, H. Martinez-Barbera, F. Martin, C.E. Aguero, V.M. Gomez, V. Matellan, and M.A. Cazorla. Team chaos 200. In *Proceedings of the IEEE 3rd Latin American Robotics Symposium, 2006. LARS '06*, pages 208–213, 2006.

[10] J. Jackson. Microsoft robotics studio: a technical introduction. *IEEE Robotics & Automation Magazine*, 14(4):82–87, 2007.

[11] N. Koening and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004.

[12] T. Laue and T. Röfer. Simrobot - development and applications. In *The Universe of RoboCup Simulators - Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots*, 2008.

[13] T. Laue, K. Spiess, and T. Röfer. Simrobot - a general physical robot simulator and its application in robocup. In *In A. Bredenfeld, A. Jacoff, I. Noda, Y. Takahashi (Hrsg.), RoboCup 2005: Robot Soccer World Cup IX, Nr. 4020, S. 173–183, Lecture Notes in Artificial Intelligence. Springer*, 2006.

[14] M. Lötzsch, J. Bach, H.D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language xabsl. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*. Springer, 2004.

[15] D C MacKenzie and R C Arkin. Evaluating the usability of robot programming toolsets. *The International Journal of Robotics Research*, 17(4):381, 1998.

[16] F. Martín, C. Agüero, Cañas J.M., and E. Perdices. Humanoid soccer player design. In Vladan Papic, editor, *Robot Soccer*, pages 67–100. IN-TECH, 2010.

[17] Max Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. PhD thesis, Fachbereich Informatik, Technischen Universitat Darmstadt, 2009.

[18] F. Rivas, J. Cañas, and J. González. Aprendizaje automático de modos de caminar para un robot humanoide. In *Proceedings of Robot 2011, III Workshop de Robótica: Robótica experimental*, pages 120–127, Sevilla, November 2011.

[19] Richard T. Vaughan and Brian P. Gerkey. Reusable robot software and the player/stage project. In D. Brugali, editor, *Software Engineering for Experimental Robotics*, pages 267–289. Springer, Berlin / Heidelberg, 2007.

[20] D. Yunta and J. Cañas. Programación visual de autómatas para comportamientos en robots. In *Proceedings of XIII Workshop on Physical Agents, WAF 2012*, pages 65–71, Santiago de Compostela, September 2012.