# Choregraphe: a Graphical Tool for Humanoid Robot Programming

E.Pot, J.Monceaux, R.Gelin, B.Maisonnier, *Aldebaran Robotics*

***Abstract*— In this paper, we present Choregraphe: a graphical environment developed by Aldebaran Robotics for programming its humanoid robot, Nao. Choregraphe is a very powerful tool that allows macroscopic connection of high level behaviors to easily develop complex software for this 25 degrees of freedom robot. But it offers as well the ability to perform fine tuning of complex joint or Cartesian motions. At the lowest level, Choregraphe allows programming in Python.**

## I. Introduction

Humanoid robots are fascinating machines. Since the beginning of robotics, they have been the subject of much research, fiction and dreams. During the last 15 years, amazing prototypes have been developed mainly by Japanese companies or labs. Progressively they demonstrate to a large audience that humanoids are not a dream but a tangible result of scientific and technological developments. ASIMO, from Honda, and HRP2, from AIST, are among the most popular humanoid robots.

In 2005, Aldebaran-Robotics a French SME, decided to develop and market humanoid robots. Not as prototypes but as products that could be bought for a reasonable price, providing multiple features and usable in a simple but efficient way. The first version of the product called Nao, is dedicated to research institutes and should be considered as a platform for software development. Nao is a robot for robotic labs who do not want to develop robotics. As SONY realized, 10 years ago with the AIBO robotic dog, there is a market for people working on high level robotics (man-machine interface, navigation, localization…) needing robotic platforms for their own development. A good example of this concept is the need of the standard league of the RoboCup. The RoboCup is an international soccer contest in which university labs compete with teams of robots. Robots are split by type into several leagues (nano robots, giant robots, wheeled robots…) in which each team develops its own robot and its own embedded software. One of the leagues differentiates itself by having as its focus is on programming expertise and therefore, the committee requires all teams to compete with the same robot: the Standard Platform League. From 2000 to 2007, the common platform for the standard league was the Sony AIBO. After SONY stopped its production of robotic dogs, the RoboCup committee made a call for a new platform. In Summer 2007, Nao the humanoid of Aldebaran Robotics, was chosen to succeed the Aibo and Nao played its first games in 2008. Using the same robot, teams have to demonstrate their value, only by programming. The programming environment of Nao should be simple and efficient.

Though today's users of Nao are mainly research labs, over time Aldebaran wishes to spread its product to the wider audience of tech enthusiasts. . As lovers of ZX81 or Commodore did in the eighties with the first personal computers, geeks of 21st century will love to implement their own software on humanoid robots creating personalized behaviors. But if you ever had experimented with assembler programming on first personal computer, you will do your best to avoid inflicting such an experience on today's users. That is the reason why it was necessary to supply Nao with a powerful programming environment. Choregraphe was conceived to allow rapid development of complex applications by generalist developers while providing the ability to have a fine control of motions for demanding programmers.

In this paper, after a short presentation of Nao, we will explain the reasons that lead us to develop our own development platform despite there being similar tools already available. Then we present the basic principles of Choregraphe and the associated tools. The fourth chapter introduces the graphical user interface allowing description of event driven behaviors. The fifth chapter introduces NaoQi, the framework on which Choregraphe is built up. Chapter six explains the interface between NaoQi and Choregraphe. Chapter seven is a description of the timeline, a strong concept for fine tuning of complex motions. Before the conclusion, we propose actual applications of Choregraphe and future improvements of our programming environment.

## II. Description of Nao

Nao is a 57cm high humanoid robot with 25 degrees of freedom. Each joint is equipped with position sensors. An

Manuscript received March 25, 2009.

E.Pot is with Aldebaran Robotics, Paris, 75014, France +33 1 77 37 17 79 ; fax: +33 1 77 35 22 68; e-mail: pot@ aldebaran-robotics.com).

J.Monceaux is with Aldebaran-Robotics (e-mail: monceaux@aldebaran-robotics.com).

R.Gelin is with Aldebaran-Robotics (e-mail: gelin@aldebaran-robotics.com).

B.Maisonnier is with Aldebaran-Robotics (e-mail: maisonnier@aldebaran-robotics.com).

inertial unit including 2 gyrometers and 3 accelerometers, and 4 Force Sensitive Resistors under each foot give Nao the ability to estimate his current state. Sonars give a measurement of the distance between the robot and its environment. Bumpers on the feet detect collisions with obstacles on the ground. The head tactile device gives a way to communicate with the robot by, for instance, caressing Nao as a reward gesture. Other input devices are two VGA CMOS cameras (640x480) and four microphones. Microphones are very important sensors because we consider that voice should be the most natural interface between Nao and its users. A voice recognition module is part of the fundamental software functions provided with Nao. As output devices, Nao offers two loudspeakers and programmable LEDS around the eyes. The CPU is located in the head of the robot: it is a GEODE 500MHz board with 512Mo of flash memory and possible extension via a USB bus. A Wi-Fi connection links the robot to any local network and to other Naos if needed. Nao is powered by Lithium Polymer batteries offering between 45mn and 4 hours autonomy according to its activity.

When delivered, Nao runs NaoQi, on Linux, a framework giving access to all the features of the robot: sending commands to actuators, reading sensors, managing Wi-Fi connections… NaoQi is able to execute functions in parallel, sequentialy or driven by events. NaoQi's functions can be called in C++, in Python and even in Urbi.

Choregraphe running on a PC gives access, thanks to a graphical user interface, to all the functions provided by NaoQi.

Fig. 1. Nao

A complete description of Nao is available in [1].

## III. State of the Art

The concept of Nao is to be an open platform programmable by users who want to implement behaviors for the robot. Because we believe that each household will have at least one robot, it is important that anybody can program his robot in an easy and seamless manner. But because robots are elaborate devices able to perform complex tasks, it is necessary that the programming tools give access to all the features of the robot in a very efficient way.

Each robot manufacturer provides its own programming language. Sony Aibo came with Tekkotsu, a very specialized environment requiring complex software architecture. Programming a simple task required the user to understand complex structures. Lego proposes the Lego Mindstorm NXT software based on NI Labview. HRP2, the Japanese humanoid robot from AIST, can be programmed by a dedicated open software OPEN-HRP. Based on this observation, some companies propose universal programming languages for robots. We can mention here Microsoft Robotic Studio that offers a wide community of developers for plenty of existing robots; and Gostaï that proposes with Urbi, the Universal Robotic Body Interface.

All the languages we studied have their qualities but none of them answered to all of our criteria: the programming tool should run on any platform (PC, MAC, Linux) because we consider that our customers should not have to modify their computer environment when buying a robot; the programming environment should offer a level of abstraction allowing simple behaviors developments and ease for complex embedded software. Another important missing feature of many development tools is the ability to have both event control programming and time based programming. Last but not least, only a dedicated programming tool could exploit all the capabilities of Nao.

Considering qualities and limitations of existing environments, Aldebaran Robotics developed NaoQi and Choregraphe, its own programming tools.

## IV. Basic Principles

Choregraphe is a very intuitive graphical environment that allows simple programming of Nao. When the software is launched (version are available on Windows XP, Linux Ubuntu and Mac OS X Leopard), the graphical user interface displayed in figure 2 appears on the screen. The application window is divided into three zones: Zone 1, the Box Library that regroups the list of available behaviors. Zone 2, the Flow Diagram that allows the user to graphically lay out behaviors composed of library boxes and links between them. In Zone 3 is a graphical representation of Nao able to execute the implemented behavior.

A behavior is a piece of software controlling the robot. Programming Nao consists in implementing behaviors, themselves made of several behaviors. In the Box Library a set of classical pre-programmed behaviors are proposed from high level functions (walk, dance, turn, lie down, standup, reading sensors, speech synthesis, speech recognition) to very low level ones (reading sensors, turning LEDS on and off). By assembling these basic behaviors, it is possible to create an original behavior. Of course, anybody can create their own boxes that can be added to the existing library.

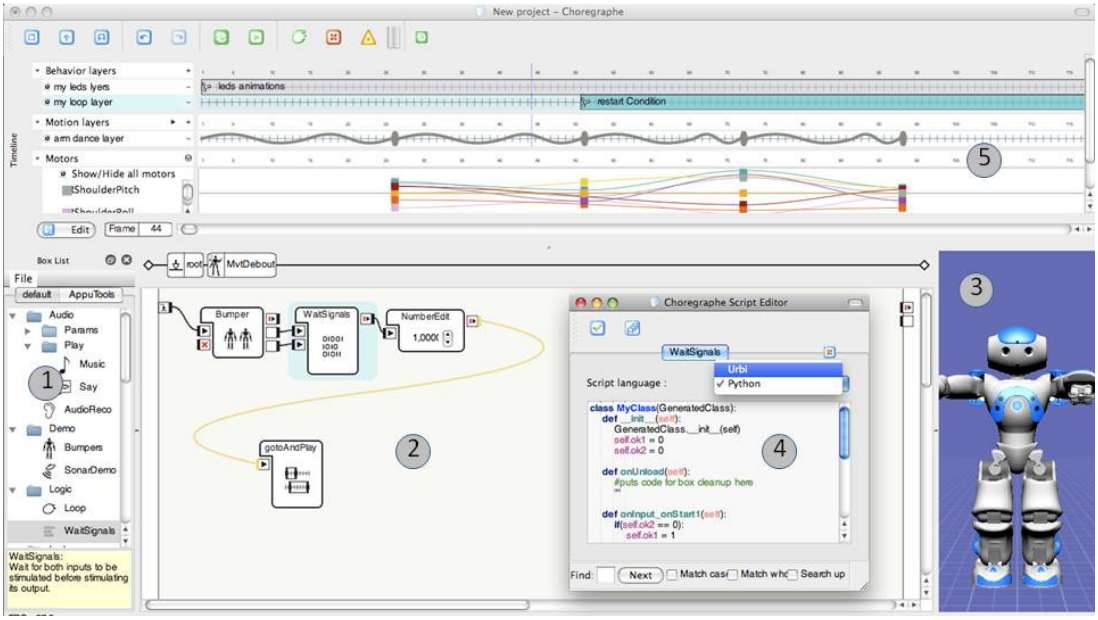


Fig. 2. The Choregraphe Graphic Interface

Assembly of behaviors is performed in the Flow Diagram. By "dragging and dropping" icons of behaviors from zone 1 to zone 2, it is possible to implement the behaviors on Nao's onboard computer. In zone 2, a behavior is represented as an icon equipped with small squares on the left (entry boxes) and small squares on the right (output boxes). Connecting the output box of one behavior to the input box of another one means that the second one will be executed one the first will be finished (event programming). On the top left of zone 2, the global "entry box" represents the starting point of the behavior. On the top right, the global "output box" represents the end of the behavior. The principle of programming Nao is to connect sequential, or parallel, behaviors between the "entry box" and the "output box".

When behaviors are chained from the entry box to the output box, they can be run on Nao by clicking on the "play" button on the top of the interface. If a real Nao is connected, via Wi-Fi to the PC running Choregraphe, the real Nao will execute the global behavior. If no real Nao is available, or if the behavior has to be tested before actual use, Choregraphe is connected to the 3D graphical Nao of Zone 3. The virtual Nao executes the programmed behavior.

In order to give a global overview of Choregraphe, two other zones are presented on figure 2. Zone 4 is a script editor for Python instructions to develop new boxes in a more classical way (computation, conditional branches…). Zone 5 is the timeline that allows time based programming. That is presented in section VII.

The first goal of Choregraphe is reached: programming simple behaviors for Nao is fast and intuitive. In the next chapters, we explain how it is possible to develop complex behaviors while using the same basic principles.

## V. NAOQI

### A. *Distributed Framework*

The reason for the efficiency and the power of Choregraphe is mainly to be found in NaoQi, the framework running behind Choregraphe. NaoQi is a distributed environment which allows several distributed binaries, each containing several software modules to communicate together. NaoQi is cross-platform. It runs on Linux (on the Nao's CPU) but also on Windows and Mac OS. Actually, Choregraphe is a specific module of NaoQi running on the development computer.

The use of NaoQi on the development computer brings

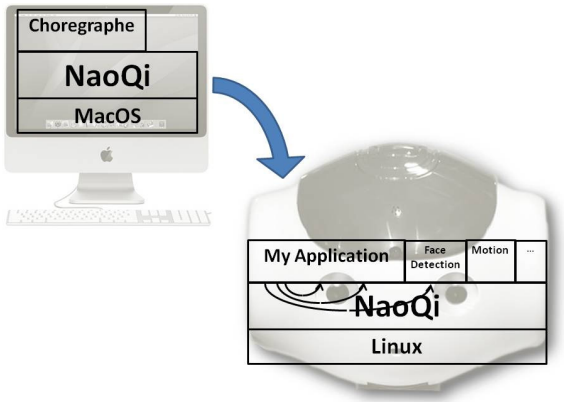


Fig. 3. NaoQi in Nao's environment. Choregraphe produces a file describing the application. The file is downloaded to Nao's memory. Embedded NaoQi interprets this file, calling the required modules.

naturally the ability to run the application developed for Nao on the development computer itself. If any Nao is connected to the computer, the application is running on the computer, controlling the graphical Nao.

NaoQi is cross-language. As basic modules, it provides Python and Urbi interpreters. C++ applications can use NaoQi's features through a dedicated SDK.

### B. Multi-tasking capabilities

Thanks to specific NaoQi commands it is possible to execute methods sequentially, in parallel or on event.

Instructions:

*Motion.Walk(0.1)*
*Speech.Say("I am arrived")*

will be executed sequentially as in any programming language. When the robot has walked 0.1m, he says "I am arrived".

Following instructions:

*Motion.post.Walk(0.1)*
*Speech.Say("I am walking")*

will execute parallel orders: the robot will walk 0.1m while saying "I am walking".

This ability is absolutely necessary to take into account the specificity of robotic applications: planned tasks executed sequentially, are running in parallel of perception tasks and are interrupted when an unexpected event happens. Events are managed via a shared memory module called AL_Memory

### C. AL_Memory

ALMemory is a class that implements an event-based storage base. Users can store data inside ALMemory, they can store groups, or relationships. Modules can access data via its name with write and read functions. But it is possible to subscribe to data, with the instruction:

*AL_Memory.subscribeOnDataChange(x,y,z,t)*

When subscribing, the application will receive notifications when data is changed with callback.

The data can be the binary value of a simple sensor (feet bumper for instance) or an analog value of a more sophisticated sensor (sonar). In this case, the value will change permanently and, at each sampling period, an event will be triggered. In this case, it is more relevant to have a procedure checking the value of the sensor data. When the data goes through a threshold, a dedicated data is set to 1 in AL_Memory. This data will be used as trigger by the application. The same kind of mechanism is used by vision processing modules delivering semantic information extracted from the pictures rather than 640x480 values of pixels.

If a developer wants to access directly to sensor values and actuator commands, a dedicated module called Device Communication Manager (DCM) is available. DCM is in charge of interfacing AL_Memory with sensors and actuators. Thanks to a Hardware Abstraction Layer, the DCM can give access in a same way to values on real robots or values on simulated ones. It could even address other robots than Nao.

### D. Distributed Functions

One of the most efficient capabilities of NaoQi is the distribution of functionality tree (Fig 4). As soon as NaoQis running on different platforms are connected via wired or wireless network, they communicate and they can exchange data and resources. For instance Nao#1 can use a module running on Nao#2 or a connected computer. From the application point of view, this is completely transparent. For the development of complex software, like vision processing, the application can be developed and debugged on the development computer when the video signal comes from the camera embedded on Nao. This functionality facilitates the development of impressive group demonstrations when several Naos are dancing together. Actually, all the dancing Naos are receiving the same orders from on single dance module running in the head of only one of the Naos.

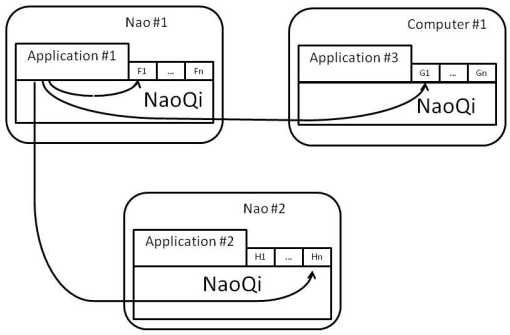


Fig. 4. Distributed Tree of Functionalities: If an application needs a module that is not available locally, NaoQi will look for it on other connected NaoQis

Compared to CORBA, which is dedicated to access to remote modules, NaoQi give the ability to access in a very transparent way to remote modules as well as local ones.

## VI. FROM NAOQI TO CHOREGRAPHE

Choregraphe gives access in a very intuitive way to all the functions provided by NaoQi. To describe sequential execution, behavior boxes are chained (Fig 2). To describe parallel execution, the output of the previous box is connected, with two links, to the inputs of the behavior boxes expected to run in parallel (Fig 5). Specific graphical representation is proposed to represent information coming from AL_Memory. With few clicks, it is possible to describe the name of the triggering variable.

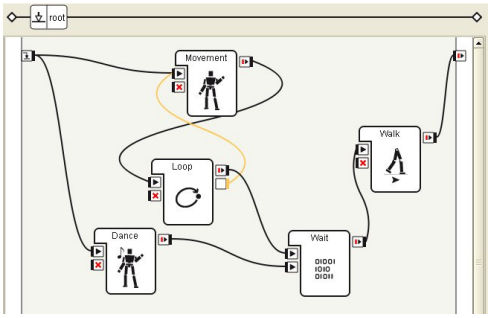


Fig. 5. Example of complete application with parallel and sequential behaviors. The robot is dancing while moving its head. When the dance is finished, the robot stops nodding and walks.

The user can build new boxes. Inside a box, it is possible to draw new flow diagrams or describe a procedure in Python or in Urbi. The next paragraph will present another kind of box.

From the NaoQi's point of view, Choregraphe is a module dealing with a graphical representation for the user. The Choregraphe module produces an XML file describing the application, the boxes, the connections between them, the included Python scripts etc... For execution, the file is interpreted by the XML module of NaoQi. If a robot is connected to the development computer, the embedded NaoQi interprets the file. If there is no Nao connected, the local NaoQi interprets the file.

Technically Choregraphe is just a graphical representation of NaoQi's functions but practically it is much more. It is a way for non-expert developers to avoid the complexity of "*post*" subtlety or "*subscribeOnData"* functions. But it is also a way to organize, in a very convenient manner, complex applications. Without the graphical representation, the last feature presented in the next paragraph, would be almost impossible to implement in such an intuitive and efficient way.

## VII. The Timeline

The timeline is the solution proposed by Choregraphe to implement time control in the programming process to describe the duration of each behavior. Creating a timeline box gives access to a new window similar to the high level interface of Choregraphe (Fig 1) at the top of each a horizontal frame is added (Fig 6).

In this frame, the horizontal axis represents the time along which three zones are represented. Zone 1 contains one or several behavior layers. Zone 2 contains one or several motion layers. Zone 3 contains a graphical representation of motor motions.

In a behavior layer of zone 1, it is possible to put several behaviors side by side. Each behavior is represented by a rectangle, called a keyframe. The length of the rectangle represents the duration of the behavior. If the behavior is represented by a 10 second long rectangle, the behavior will be stopped after 10 seconds even if the behavior is not finished. Then, the following behavior on the same layer is triggered. Behaviors on the same level are triggered sequentially when their duration is over. The event that causes the exit of a behavior is not a signal coming from another behavior (as described before) but should be considered as an interruption coming from a timer. To change the duration of a behavior, the user has just to change the length of the rectangle on the timeline.

If another behavior layer has been created in zone 1, the behaviors of this layer will be executed in parallel with those of the first layer. For musicians, this representation is very similar to the different staves of a musical score.

In Zone 2, the same principle of layers is implemented but the successive items on a layer are positions of the robot. Each vertical ellipse represented on a motion layer stands for a position of the robot. The snake-like line between two

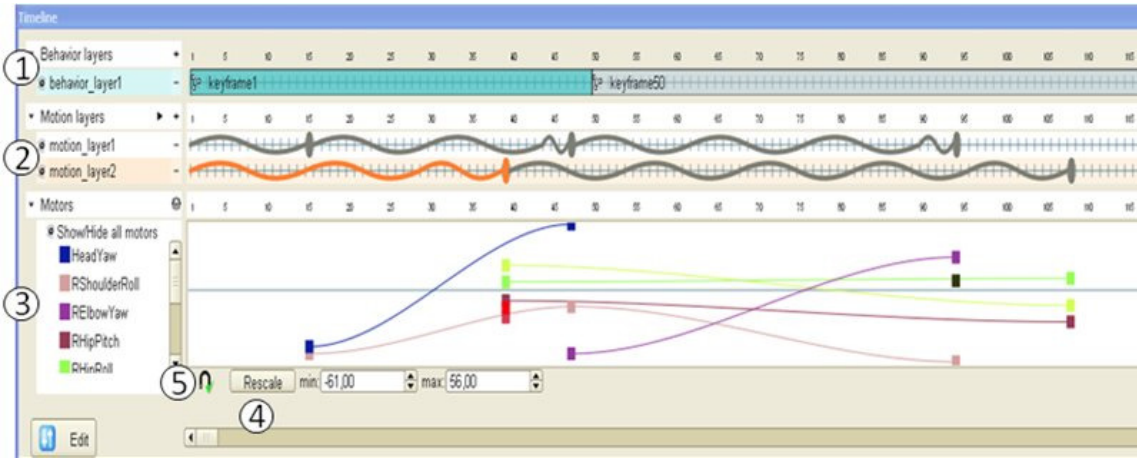


Fig. 6. The Timeline. Behavior layers and motion layers contain actions to be executed in parallel. Horizontal position indicates the starting time and duration of each action.

ellipses is a symbolic representation of the motion from a position to another. The position on the horizontal axis represents the time at which the position should be reached. Positions on the same layers will be reached one after the other according to their position on the horizontal axis.

In the same way as in the previous zone, another motion layer can be created in parallel to move another limb of the robot.

Zone 3 gives a way to control the position of each motor to correct the position registered. Each joint can be controlled independently by moving up and down the colored point representing the final joint position.

For example, the first layer of behavior can contain a behavior changing the color of the leds for 60 seconds. The second layer of behavior can contain a sequence of speech behavior saying "I am doing my gym", then, at second 10, a behavior playing music for 40 seconds, then a last speech behavior saying "I am tired". The first motion layer contains the successive positions of the left arm. The second motion layer could contain successive positions of the right arm. Assuring that all the layers ending after 60 seconds, all the motions and behaviors would have been synchronized in a very simple an intuitive way.

On figure 6, one can see that during the 15 first frames, no joint motion is represented. This delay is available to go from the current position of the robot to the starting position of the keyframe.

The figure 2 shows how events in flow diagram can influence the keyframes: the box "gotoAndPlay" allows jumping to a precise frame on the timeline. In this example, when the two bumpers are activated, the Frame Manger goes back to frame 1 and "leds animations" and "arm dance layer" start again.

## VIII. Actual Applications

Today, Nao is a development platform for research institutes. The applications are as wide as the imagination of researchers. Among the numerous implemented applications, we can mention two of them.

The standard league of the RoboCup competition gives an opportunity to see how 25 teams coming from 18 countries can use a 4 Nao soccer team in such different ways. Even if very specialized functions are implemented in C++, high level programming is based on Choregraphe.

Nao is often asked for special events (opening sessions of workshops, science festivals…). Voxler, business partner of Aldebaran, developed a special dance for Nao for the "Futurs en Seine" event organized by the Competitiveness Cluster Cap Digital in June 2009 in Paris. Voxler is a middleware development team and technology focused on vocal interaction. Thanks to Choregraphe, they implement very efficiently an interactive dance where the robot was taking into account music and audience reactions.

## IX. Future Development

For the time being, one of the most frustrating issues when developing with Choregraphe is to not have a real Nao. The 3D graphical representation gives an idea of the developed behavior but, without mentioning that a real robot is more interesting than a graphic one, it is not a full simulator. The gravity is not taken into account, which means the onscreen robot never falls. Input from sensors cannot be emulated from a simulated environment in which the simulated robot would move. A realistic dynamic simulator should be integrated to Choregraphe to increase the efficiency of the programming tool. Even if simulations of Nao are already available with Webots [5] the simulator developed by Cyberbotics for instance, or within MSRS and that they allow satisfying simulated situations, none provide the level of integration we want to have within Choregraphe.

Originally built as a graphical interface for NaoQi framework, Choregraphe became a tool for software design. Aldebaran Robotics will develop this aspect of the tool to propose a standard for graphical programming of robots. Applying Choregraphe on other robot would require a graphical model of the new robot and the development of the middleware between NaoQi and its hardware.

## X. Conclusion

In this paper, we wanted to present the main principles that made the efficiency of the Choregraphe programming tool. We have stressed the interest of the graphical interface that allows anyone to easily implement complex behaviors controlling the numerous features of Nao. The flow diagram representation gives the ability to describe sequential and parallel behaviors triggered by events, while the timeline gives access to a time scheduled programming. The simultaneous use of both sides opens huge possibilities to play with a humanoid robot without compiling a single line of C++. Thanks to the Wi-Fi connection, as soon as a behavior has been developed in Choregraphe, a simple click on the Play button gives the opportunity to watch Nao executing the (generally) expected motions.

## References


[1] D. Gouaillier, V. Hugel et al. *"Mechatronic design of NAO humanoid"*. IEEE Int. Conf. on Robotics and Automation, Kobe, 2009.

[2] http://msdn.microsoft.com/robotics.

[3] JC.Baillie, *"URBI: A Universal Language for Robotic Control"*, International journal of Humanoid Robotics, Oct. 2004

[4] International Electrotechnical Commission: International Standard IEC 61131-3

[5] *"Webots: Professional Mobile Robot Simulation"*, International Journal of Advanced Robotic Systems, Vol. 1, Num. 1, pages 39-42,2004.