Universal Programming Interfaces for Robotic Devices

Jean-Christophe Baillie

Ecole Nationale Supérieure de Techniques Avancées (ENSTA)
Laboratory of Electrical and Computer Engineering
32, Bd. Victor 75015 Paris, France
jean-christophe.baillie@ensta.fr

Abstract

Robotic devices, whether service robots designed to help people or entertainment robots, are more and more widespread, and their number is increasing. All these robots currently have a different programming interface, more or less complex and more or less powerful. The situation is reminding of the 80's and the personal computer revolution: many vendors, models and as many programming languages and interfaces. We have developed URBI, a Universal Robotic Body Interface in an attempt to address this issue and provide a standard and simple way to control robots, while still providing powerful high-level capabilities expected from a modern programming language. To achieve this, URBI is based on a client/server architecture where the server is running on the robot and accessed by the client, typically via TCP/IP. The client can be any system, thus giving a great deal of flexibility. The URBI language is a high level scripted interface language used by the client and capable of controlling the joints of the robot or accessing its sensors, camera, speakers or any accessible part of the machine. We present in this article a short introduction to URBI and show application examples with Aibo. We finally explore how URBI could impact the development of everyday robotics and facilitate the interaction between robots, computers and smart objects in general.

1. Introduction

Robotic devices, whether service robots designed to help people or entertainment robots, are more and more widespread, more and more complex and their number is increasing, opening doors to what is already called "everyday robotics" [8]. In this paper, we will discuss the question of how to interact with these robots, from a programmer standpoint. We will mainly focus on entertainment robots, which often present the most complex interfaces and capabilities.

The various robots that have been recently created (Aibo, QRIO, Asimo, HRP2…) currently have a different programming interface, more or less complex and more or less powerful. One well known example is the Aibo robot and the Sony OPEN-R SDK [4]. This general situation is reminding of the 80's and the personal computer revolution: many vendors, models and as many programming languages and interfaces. These new robotic devices share another trait with the early computers of the 80's: they offer an important potential for applications and they trigger people's imagination. One of the key issues to develop this potential is the possibility to program the robots in a simple and standard way, which, to a certain extent, should not depend on the robot type. The programming language, or interface, should be adapted to robotics and not a simple extension of classical computer programming, and it should be easy enough so that any hobbyist with a basic background in computer science could start programming its own robot, and not only IT professionals. But more advanced features should also be available for more demanding applications. This could open doors in terms of markets for education and entertainment as well as industrial applications.

With these constraints in mind, we have developed URBI, a Universal Robotic Body Interface, in an attempt to provide a standard and simple way to control any robots on a low level scale, while still providing powerful high-level capabilities expected from a modern programming language [1], [2], [3].

URBI is based on a client/server architecture where the server is running on the robot and accessed by the client, typically via TCP/IP. The client can be a Linux PC, a robot, a Mac or any kind of computer, thus giving a great deal of flexibility. The URBI language is a high level scripted interface language used by the client and capable of controlling the joints of the robot or accessing its sensors, camera, speakers or any accessible part of the machine. We present in this article some views on robot interaction and a short introduction to URBI with application examples for the Aibo robot. We finally explore how URBI could impact the development of everyday robotics and facilitate the interaction between robots, computers and smart objects in general.

2. Interacting with robots

There are many ways to interact with a robot. We will focus on the basic programmer's point of view, which is: how to make the robot do things? We can see this question from two opposite perspectives:

The most natural way of having the robot do things is to speak and tell it or teach it, in the same way we would do with humans or, more simply, with a animal like a dog. In this perspective, we don't actually program the robot; we educate it or talk it into doing things. We could call this "*natural programming*". The result is not necessarily accurate or dependable but the flexibility is important. This is the aim of several ongoing research activities in cognitive robotics (see [6], [7] for example) and is still a far goal to reach.

On the other hand, the most "unnatural" but also precise and accurate way is to have the robot controlled by a program. We can use complex interaction protocols (CORBA, GENOM…) and high level languages (C++, java, python…) to give specific instructions to the robot hardware and the robot will do exactly what it is told. Of course, it is necessary to know how to explicitly state the task requirements step by step, or design a converging dynamic system to do so, which is most of the time an open research topic. More importantly, the skills necessary to master the programming tools, like CORBA for example, or OPEN-R, can be important and will most of the time confuse non-professional users. Even for professional programmers, the task is usually time consuming and error prone.

URBI could play a role as an intermediary solution, simple enough to be understandable and usable by non-specialists, while being still powerful and extensible with all the benefits of a modern programming language. Also, since it has been designed with robotic applications in mind, it is more directly suitable for this type of programming, providing useful features and extensions. It can make things easier and could be perceived as being closer to the idea of "natural programming" than C++, for example.

Other programming interfaces, like Tekkotsu [9] or Player/Stage [10], have been compared to URBI in [1]. They differ mainly in terms of expressivity, scope and ease of use. Also, none of them offers a dedicated robotic oriented language: they are libraries on top of previously existing languages, which has clear advantages but also limitations. Many features trivially available with URBI would require a lot of programming with these other approaches. On the other hand, they provide some high level functions that are not natively available in URBI.

To have an idea of the URBI language, we will now describe some of its key features and show examples.

3. URBI Architecture

URBI is based on a client/server architecture. A URBI server is running on the robot and a client is sending commands to the server in order to interact with the robot. The communication channel between the client and the server can be a TCP/IP connection or direct Inter Process Communication (IPC) if the client and the server are both running on the robot.

The robot is described by its *devices*. Each element of the robot that can be controlled or each sensor is a device and has a device name. From a programmer's point of view, a device is an object, with methods and variables. Everything that can be done on the robot is done via the devices and the available methods and variables associated to them.

The main advantage of using the client/server architecture is the flexibility it allows. The client can be a simple telnet client or a complex program sending commands over TCP/IP. This client can run on Linux, Windows or Mac OSX and it can be programmed in C++, java, python or any language capable of handling TCP sockets (currently, C++, Java and Matlab libraries are available).

For each new robot type, a new server has to be written, but this is done once. Once this server is running on the robot, it is straightforward to command the robot for any user, whatever the robot is or how complex it is, as soon as one knows the list of devices and their associated methods. This list is supposed to be made available in the documentation of the robot and it is the only robot-specific piece of information required to know how to control a previously unknown robot.

The syntax used to access the devices is designed with simplicity in mind. More complex features of the URBI language are available but understanding them remains easy and it is an incremental process: it is not necessary to understand the complex features to use the robot at a basic level, which was one of the requested properties listed in the introduction, making URBI suitable for specialists and hobbyists as well.

A detailed study of the performances of URBI can be found in [1] and shows that the client/server architecture using a wireless 802.11G connection is fast enough for real-time reactivity and demanding applications. URBI on Aibo can handle up to 600 motor commands per seconds, with a 1.5ms latency and delivers images at 30fps. The reactivity time is even higher (600µs) when the client is directly embedded in the robot, using IPC communication.

4. URBI language

The working cycle of URBI is to send commands from the clients to the server and to receive messages from the server to the clients.

Commands can be written directly in a telnet client on port 54000, where the messages will also be displayed, or interfaced in a program including the URBI library (see *liburbi* on www.urbiforge.com).

4.1. Getting and setting a device value

As we said in the architecture description, each element of the robot is called a device and has a device name. For example, in the case of Aibo, here is a short list of devices: `legFL1`, `neck`, `camera`, `speaker`, `micro`, `headsensor`, `accelX`, `pawLF`, `ledF12`... To read the value of a device, the `val` field is available:

```
> neck.val;
[036901543:notag] 15.1030265089
```

The message returned (second line) is composed of a first part between brackets displaying a timestamp in milliseconds (counted from the start of the robot) and a command tag. In this case, the command tag is `notag`, since no tag has been specified with the command. The tag can be specified before a colon preceding the command. With a command tag, it is possible to retrieve the associated message later, possibly in a flow of other messages from the server:

```
> mytag: neck.val;
[041307845:mytag] 15.0040114317
```

This tagging feature is an essential part of URBI and the URBI library, where callback functions can be associated to any tag enabling asynchronous message handling.

The second part of the message is the response of the server. In the case of our example, it gives the value of the Aibo `neck` device `val` field, which is the position of the neck motor in degrees. One important fact for standardization is that the `val` field is available with any device. The type of data returned depends on the device: for example, camera devices return binary data (see [1] for more details on binary transfers in URBI).

Symmetrically, the `val` field can also be used to set a particular device value. If the device is a motor, it is going to move to the specified value. In the case of a LED, this will switch it to the corresponding illumination (between 0 and 1):

```
> motoron; // to activate the motors
> headPan.val = 15;ledF1.val = 0.6;
```

4.2. Modifiers

Modifiers are a particularity of URBI. The value specified by a `val` field assignment command is normally reached as quickly as the hardware of the robot allows it. It is however possible to control the speed and many other movement parameters using *modifiers*. The following example commands the robot to reach the value 80 degrees for the motor device `headPan` in 4500ms and the value 40 degrees for `headTilt` with a speed of 12.5 degrees per seconds:

```
> headPan.val = 80 time:4500;
```

```
> headTilt.val = 40 speed:12.5;
```

The `speed` or `time` modifiers are always positive numbers. It is possible to specify a speed without giving a targeted final value by setting the desired value to infinity (`inf`) or minus infinity (`-inf`). For example, in the case of a wheeled robot with control in position, this command sets the right wheel speed, in the "positive" direction:

```
> wheelR.val = inf speed:120;
```

Another interesting modifier is `accel` whose meaning is to control the acceleration.

One of the most interesting modifier is `sin`, followed by a time period and coupled with the `ampli` modifier, which makes the assigned variable oscillate around the value with the specified period and in a sinusoidal way with the given amplitude. Additionally, the phase can be controlled by the `phase` modifier:

```
> neck.val=45 sin:400 ampli:20 phase:pi/2;
```

This modifier can be used to design complex periodic patterns by superimposing several sinusoidal profiles, as explained in the "multiplexing" section below (4.5).

Modifiers are a unique and powerful feature of URBI compared to other existing languages and which makes it a fundamentally asynchronous and time-oriented language. Variables are not only containers but can store dynamic profiles evaluated in real-time. Besides, since modifiers are constantly reevaluated online, it is also possible to create a dynamics for the parameters themselves. This is useful in many situations, like, for example, in the design of a walk sequence for a legged robot with a variable speed expressed as the period of the `sin` modifier.

4.3. Serial and parallel commands

One key feature of URBI which make it fundamentally different from C++, Java and traditional procedural languages is the ability to process commands in a serial or parallel way. When two commands are separated by the "&" operator, they will be executed in *parallel*. In addition, they will start at exactly the same time:

```
> headPan.val = 15 & headTilt.val = 30;
```

This will move the head pan and tilt together, with both motors starting at the same time. In the same way, it is possible to *serialize* commands by separating them with a pipe. In that case, the second command will start just after the first one is finished, with no time gap.

```
> headPan.val = 15 | headTilt.val = 30;
```

This will move the head pan to 15 degrees and only when this value has been reached, and just after, it will start to move the headTilt motor.

Two commands separated by a semicolon have almost the same time semantics as the serial "|": the second will start after the end of the first, but the time gap between the end of the first and the beginning of the second is not specified. This is close to the standard semantics of C or C++. Most of the time URBI commands will be separated by semicolons.

Finally, two commands can be separated by a colon. In that case, the time semantics is close to the parallel operator "&", except that the two commands will not necessarily start at the same time. The meaning of a colon terminated command is simply to start the command as soon as possible. In particular, as soon as the command is in the receiving buffer of the server, it will be executed, whereas with "&", the chain of commands must be integrally received before execution. The following relationships represent those time dependencies:

```
a;b : b.start >= a.end
a,b : b.start >= a.start
a&b : b.start == a.start
a|b : b.start == a.end
```

Technically speaking, the consequence of those different operators is that commands are not stored in a pile in the URBI internal structures, but in a tree. More details on the practical implementation of the URBI kernel will be published on *www.urbiforge.com*.

These time sequencing capabilities are very important features to design and chain complex motor commands or behaviors. They are particularly well suited for robotics applications.

4.4. Loops, conditions, event catching

Several control structures are available, like the classical "`for`", "`while`" and "`if ... else`". Some new control structures like `loop`, which is equivalent to `while (true)`, or `loopn (n)` equivalent to `for(i=0; i<n;i++)` are also provided for convenience. The syntax of `for`, `while` and `if` is the same as in C. "`for &`" is a parallel implementation of "`for`" which will start every iteration at the same time. "`for |`", "`while |`" and "`at &`" are also available.

As a specificity of URBI, event catching control structures like `whenever`, `at` and `wait` are also available:

The instruction "`at (test) command`" will execute the command only once at the moment when the test becomes true. The instruction "`whenever (test) command`" will execute the command as long as the test is true. When the test becomes false, the command is not restarted once it is finished and the `whenever` instruction silently waits for the test to become true again. The semantics is close to `while`, except that the instruction never terminates: both "`at`" and "`whenever`" are run in the background, they return but they do not terminate.

The instruction "`wait (test)`" is blocking until the test becomes true. Another usage of this instruction is "`wait (tps)`", where `tps` is a number. In that case, the instruction will do nothing but lasts during `tps` milliseconds.

4.5. Multiplexing

Another key feature of URBI, is its capability to perform multiplexing of assignment commands, which can be seen as a sort of integrated mutex facility. The URBI server running on the robot is a multi client server, this means that it is always possible that two contradictory commands are sent to the server from two different client (or contradictory commands can be executed in parallel from a single client). For example, what should be done if one command requests the `neck` device to be set to 20 degrees while the other one requests a value of -30? Six strategies are available in URBI:

- `[normal]`: The last command received is executed on top of others, but the others are still running "silently" in the background (default)

- `[discard]`: The last command is ignored and erased if there is already one command running which is conflicting.
- `[cancel]`: The last command replaces any previously existing command
- `[queue]`: Queue the commands and execute them one after the other
- `[mix]`: Mix conflicting commands by averaging the instantaneous values
- `[add]`: Mix conflicting commands by adding the instantaneous values

For each URBI variable, those strategies can be selected via the `blend` property. For example, the following code calculates the average value of an array `tab` by setting the receiving variable `m` to the `mix` mode and performing a parallel affectation of all the array elements to `m`:

```
m->blend = mix;
for & (i=0;i<10;i++) m = tab[i];
```

Of course, one of the main interests of the `mix` and `add` modes is to aggregate several conflicting motor commands, as we will see in the examples. In the case of a sound playing device, setting the blending strategy to `mix` or `add` enables the robot to play several sounds at the same time, instead of queuing them.

4.6. Other language elements

Several other elements of the language are available, like the capacity to group devices into virtual devices and propagate commands along the device hierarchy, function definition, binary types, flags, static variables, advanced event processing and behavior definitions. We will not present those elements here, but extensive details can be found in [1], [2].

5. Code examples on Aibo

URBI, which is a command script language, is normally supposed to be used together with a client program written in a language like C++ or Java, which will handle all the image processing and cognitive part of the robot behavior. However, it is possible to write quite complex and useful programs fully in URBI, without the use of an external client. A simple telnet is enough, or an elaborated telnet version like *urbilab* (see www.urbiforge.com).

To illustrate this point and some of the capabilities of the language, we present here a set of simple examples in URBI, performing interesting action/perception loops on Aibo robots. It is interesting to compare the compactness and simplicity of the URBI code to the equivalent code in OPENR (see [4]), the Sony Aibo programming SDK. On average, URBI programs are 100 times smaller than equivalent OPENR programs.

5.1. Ball Tracking Head

The perception part in this example is limited to detecting a red ball in the image. This is done in the Aibo version of the URBI server via a *soft device* called `ball`, which is constantly setting the variables `ball.x` and `ball.y` to the ball position in the image (between 0 and 1), and -1 otherwise (`ball.size` is also available). The "action" part of the program is to follow the ball when the robot sees it, and search for it with circular head movements otherwise:

```
whenever (ball.x != -1) {
   headPan.val = headPan.val +
       camera.xfov * (0.5 - ball.x) &
   headTilt.val = headTilt.val +
       camera.yfov * (0.5 - ball.y)
};

at & (ball.x == -1 ~ 500ms)
   scan : {
       headPan.val\n = 0.5 sin:4000
       ampli:0.5 &
       headTilt.val\n = 0.5 cos:4000
       ampli:0.5
};

at (ball.x != -1) stop scan;
```

The `val\n` field is a normalized equivalent to `val`, based on the device *min* and *max* range (accessible via the `rangemin` and `rangemax` properties). `xfov` and `yfov` are constants giving the ratio between pixels and angles in degree for the Aibo camera. "Ball Tracking Head" is a typical example given with the Sony OPEN-R SDK. The URBI version is comparatively much simpler to understand and requires only ten lines of code in URBI, compared to about 600 lines for the OPENR version. The performances are comparable to the native OPENR version. The structure of the program is easy to grasp by reading the code and we expect URBI programs to be much easier to maintain than OPENR versions.

5.2. Mirroring

This simple program mirrors the right-front leg (RF) to the left-front leg (LF):

```
legRF.load = 0;
mirrortag: loop {
     legLF1.val = legRF1.val &
     legLF2.val = legRF2.val &
     legLF3.val = legRF3.val
},
```

The `load` field is available for all motor devices and controls how tensed the motors are. By setting it to zero, the motor becomes loose. The `loop` command is constantly doing the mirroring for the three joints of the Aibo leg and can be stopped with `stop mirrortag`. It is a good programming habit to prefix with a tag every non-terminating command, like `loop`, in order to be able to stop them later.

5.3. Stand-up sequence

The following code is performing a simple sequence of leg movements to have the robot stand up. This is a relatively complex sequence to program in OPENR, but it is done very simply here by using the serializing and parallelizing capabilities of URBI, together with *time modifiers*:

```
{ leg2.val = 90 time:2000 &
  leg3.val = 0 time:2000 } |
leg1.val = 90 time:1000 |
leg2.val = 10 time:1000 |
{ leg1.val = -10 time:2000 &
  leg3.val = 90 time:2000 }
```

`leg1`, `leg2` and `leg3` are virtual devices grouping all the level 1, 2 and 3 leg joints. See [1] or [3] for more details on virtual devices and grouping.

5.4. Walk sequences and perturbation-based turning

Walk sequences are good examples of simple applications of URBI for Aibo. The simplest way of doing a walk sequence with URBI is to use basic sinusoidal movements on all joints of the legs.

To go one step further, using the `add` blending mode, it is possible to superimpose several sinusoidal motion profiles and build any kind of motor trajectory by using the main coefficients from the Fourier decomposition of the trajectory.

One interesting idea is to try to make use of the "add" blending mode to generate a turning behavior while a walk is performed, using only a perturbative approach. In this approach, conflicting assignments are sent to the joints on top of the running walk commands and, therefore, are added to the current motion profile. This method is particularly suited for ZMP-based walk control [11].

Since URBI is capable of handling a large amount of motor commands per seconds (30 motor commands per motor and per second), it is also possible to store a walk sequence and replay it by sending the series of motor positions or, simply, to calculate the series of motor positions using reverse kinematics in a separate program (in C++, for example) and send them to the robot. This illustrates some of the flexibility of the URBI approach.

6. Possible role of URBI in the development of everyday robotics

We have presented a short technical description of URBI. The level of technical knowledge required to use URBI is not comparable to "natural programming" but is comparatively much lower than the level requested to use OPENR or CORBA-based approaches.

Considering its specificities, we want to investigate what role could URBI play in the development of everyday robotics. What are the strong and weak points of the language in that regard and what makes it fundamentally different from already existing solutions?

6.1. URBI as a standard

If URBI succeeds in becoming a *de facto* standard in robot programming, it will help making robotic applications easier to develop and programs more portable. This could help to create a momentum for the robotic software industry by allowing companies to develop robot programs working with a large variety of robotic hardware, without major incompatibility issues or development time and costs. In fact URBI as a standard is not limited to strict robotic applications but could be used with a wide range of remotely controllable devices, including smart objects in general.

However, it is clear that URBI alone is not sufficient to develop demanding robot applications. It needs some fast and compiled language to rely on for visual processing, sound processing or complex AI programming. This is why URBI is what we call an *interface* language. So, one important step towards a standard is to create powerful bridges with fast languages, like C++. This is currently done via the *liburbi* library but the integration with URBI could be tighter. Integration with python, which is already closely coupled to C++, might be another option.

One of the main weak points of URBI at the moment in the perspective of a standard development is its lack of industrial support, since it is a recent creation. We expect that the free distribution model of URBI will help to create a community of users which could drive industrial interest and support. Currently, we have URBI servers for Aibo, HRP-2, Webots simulation environment, and a private company companion robot. We plan to develop URBI servers for pioneer robots and simulation tools like ODE.

6.2. URBI as an educational tool

Another important role to play for URBI is in the education domain. The relative complexity of today's object oriented computer languages and GUI programming makes it difficult for children to develop their own programs, as they did in the 80's with the language "Basic" or "Logo" and the personal computers. In general, programming is no more a hobby for today's children. One of the focuses of kids in the 80's was in game programming (impossible today with modern games), competing with existing software and pushing the limits of the machine. They were getting "fun" out of programming. To some extent, the computer revolution has entered our homes through this door and the everyday robotics revolution might follow the same path: through entertaining and fun robot programming, using a simple and universal language, together with powerful and yet affordable robotic devices.

The benefit in terms of education is probably difficult to evaluate but it is commonly accepted that programming facilitates mathematical reasoning and logical thinking, and develops creativity in a positive way. Numerous social studies have already been conducted to study the impact of technology on education [5].

6.3. Home shared computing

Finally, URBI could play a role as a standard interface between computers and robots in general, including entertainment robots but also more generic devices or smart objects. The benefit of the URBI protocol of communication is that it is simple and works with a standard TCP/IP client/server model.

Another benefit of the client/server architecture of URBI is to allow remote processing of some of the robot software components in a straightforward manner. Distributed computing is of course a well known domain, but the application to robotics at home would be a relatively new idea, what could be called "home shared computing": when the robot needs more computing power, it scans the surrounding local wifi network for personal computers which are running a CPU-sharing application, and uses these computers if needed. The key point is that it is very likely that such a secondary computer will be available in today's home environment, especially for the kind of people willing to have a robot at home. From this perspective, the limitations of onboard CPU power could be overlooked to a certain extent in the application development and URBI could provide the glue required to assemble this extra CPU power available and coordinate the different modules.

7. Conclusion

We have presented URBI as a candidate for a medium solution between "natural programming" and complex high-end programming. We have given a short description of the language itself and examples to illustrate its capabilities. The five key features of URBI which make it different from existing solutions are: integrated parallel/serial command processing, multiplexing of conflicting commands, complex assignments via modifiers, integrated event-based programming, and tagged commands. The client/server architecture adds flexibility to the approach.

Obviously, nothing of what is done with URBI could not be done with other approaches but we claim that it can be done more quickly and more efficiently with URBI, and in a portable and simple way. In this context, one of the interests of URBI is to become a potential standard for robotics control. However, even if satisfying solutions already exist (*liburbi*), there is still work to be done to integrate URBI more tightly with existing compiled languages necessary for heavy algorithmic computation.

According to the first user feedback that we have, URBI is indeed reported as simple to learn and simple to use, especially if compared to existing solutions like OPEN-R. This simplicity makes it also a good candidate for an educational tool. The programs that we have presented here are just a few lines long and are easy to maintain, whereas this would certainly require a more important effort to develop with SDKs like OPEN-R, obviously incompatible with short term educational goals.

URBI is a low level language but includes scripting and procedural features that makes it extendable. It is still in an early stage of development but is already used on a daily basis in our lab, and other labs working with Aibo have started to use it. We hope that URBI will be a useful element in the development of the field of robotics.

References

[1] J.C. Baillie. *Urbi: Towards a Universal Robotic Body Interface*. in Proceedings of the 4th International Conference on Humanoids Robotics, 2004.

[2] J.C. Baillie. *URBI: Towards a Universal Robotic Low-Level Programming Language*. in Proceedings of IROS'05 (International Conference on Intelligent Robots and Systems).

[3] J.C. Baillie. *Urbi language specification*. www.urbiforge.com, urbi.sourceforge.net, 2005.

[4] Sony. *Open-r sdk for aibo robots*, www.openr.aibo.com 2005.

[5] John Schacter. *The Impact of Education Technology on Student Achievement, Milken Exchange on Education Technology* 1999, www.mff.org/pubs/ME161.pdf

[6] Kaplan, F., Oudeyer, P-Y., Kubinyi, E. and Miklosi, A. *Robotic clicker training*. Robotics and Autonomous Systems, 38(3-4):197-206 2002

[7] Ehrenmann, M. Rogalla, O. Zöllner, R. and Dillmann, R. *Teaching Service Robots Complex Tasks: Programming by Demonstation for Workshop and Household Environments.* In Proc. of the IEEE Int. Conf. on Field and Service Robotics 2001 (FRS), Finnland 2001

[8] *World Robotics 2004*, UNECE United Nations Economic Commission for Europe, 2004, www.unece.org

[9] *Tekkotsu Development Framework for AIBO Robots*: www.tekkotsu.org, Carnegie Mellon University.

[10] Richard T. Vaughan and Andrew Howard. *On device abstractions for portable, resuable robot code*. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems, pages 2121–2427, October 2003.

[11] Huang, Q. Yokoi, K., Kajita, S., Kaneko, K., Arai, H., Koyachi, N., Tanie, K. (2001) *Planning walking patterns for a biped robot.* IEEE Trans. on Robotics and Automation, 17(3), 280—289.