URBI: Towards a Universal Robotic Low-Level Programming Language

Jean-Christophe Baillie
Laboratory of Electrical and Computer Engineering
ENSTA
32 Bd Victor 75015 Paris France
jean-christophe.baillie@ensta.fr

***Abstract*— The growing diversity and complexity of existing robotic devices like humanoids, animal-like robots or wheeled robots, lead to the development of several incompatible software interfaces to control these robots. We believe that there is a need for a standard which could be universal, powerful and easy to use. The open source project URBI, a Universal Robotic Body Interface, aims at providing the ground for such a standard. It is based on a client/server architecture where the server is running on the robot and accessed by the client, on the robot or remotely via TCP/IP. The URBI language is a scripted language used by the client and capable of controlling the joints of the robot or access its sensors, camera, speakers or any accessible part of the machine. We present in this article an introduction to URBI and we describe how URBI differs from currently existing solutions. As an example making use of URBI-specific features, we present a simple perturbative approach to walk pattern generation, with URBI running on a ERS7 Aibo robot.**

***Index Terms*— Control Architecture and Programming, Human-Robot Interfaces, Standard, Programming Language, Aibo**

I. Introduction

Humanoid and animal-like robots are more and more widespread and made available to researchers. However, each new robot is coming with its own programming language interface, most of the time forcing researchers to relearn what they already know. Besides, some of these interfaces, like the Sony OPEN-R SDK [4] or the low level control architecture of the HRP-2 humanoid robot, are notably difficult to master.

We have developed URBI, a Universal Robotic Body Interface, in an attempt to provide the research community with an open-source standard for robot control and interface, which would be both powerful and simple to use. URBI includes a scripted language with some high level capabilities, used from the client and capable of controlling the joints of the robot or access its sensors, camera, speakers or any accessible part of the machine. The main characteristics of URBI, which make it different from other existing solutions are:

- The purpose of URBI is to control the low level layer of the robot. Motors and sensors are directly read and set. Although complex high level commands can be written with URBI, the kernel of the system is low level by essence.
- URBI includes powerful time oriented control mechanisms to chain commands, serialize them or build complex motor trajectories.
- Using a client/server architecture, URBI is designed to be independent from both the robot and the client system. It relies on TCP/IP or Inter-Process Communication if the client and the server are both running on the robot.
- URBI is designed with a constant care for simplicity. There is no ”philosophy” or ”complex architecture” to be familiar with. It is understandable in a few minutes and can be used immediately with a simple telnet client.

Some already existing solutions, like tekkotsu [5] or player/stage [3], are notably different from URBI. Tekkotsu is providing a more high level and complex view on robot control (currently specialized for Aibo robots). Player/stage shares the client/server architecture of URBI but it is also providing high level features that are less general than the URBI approach. Also, player/stage is currently specialized for wheeled robots.

We will shortly present the client/server architecture used by URBI and the main characteristics of the language. To illustrate the capabilities of the language, we will give various examples and we will show how we can generate a walk pattern on the Aibo robot, together with a simple perturbation-based method using the mixing blend mode of URBI to alter the pattern and have the robot turn. Although these walk patterns are not to be compared with state of the art results, they interestingly show some of the features and expressiveness of the URBI language and their quality and performances is very acceptable.

In [1] we have detailed several performance benchmarks on Aibo to conclude that URBI is capable of handling simultaneously real-time complex motor control commands (walk or turn sequences), multiple simultaneous image retrievals at 30fps, stereo sound retrieval, sound streaming and smooth action/perception feedback loop (head tracking of the ball in the vision field). This shows that URBI is capable of sustaining demanding needs and research applications.

URBI Server implementations and libraries are released under the GNU GPL license, and are currently freely available at `urbi.sourceforge.net`. The URBI language is released under a specific license, see `urbi.sourceforge.net` for more details.

II. 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 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. It has some mandatory methods and variables and a list of device-specific methods. 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++ and Java librairies are available). For each new robot type, a new server has to be written (the C++ source code is divided into a kernel and robot-specific part, making the task relatively easy). Once this server is running on the robot, it is straightforward to command the robot, 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 server 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, and will be described here, but understanding them remains easy and is an incremental process: it is not necessary to understand the complex features to use the robot at a basic level.

Current implementations of URBI include Aibo ERS7, ERS2xx, HRP-2 humanoid robot (prototype), Webots robot simulator (Cyberbotics). We plan to release a Pioneer version in the coming months.

III. URBI language

The working cycle of URBI is to send command from the client to the server and to receive messages from the server to the client. Commands can be written directly in a telnet client on port 54000, where the messages will also be displayed or using a program and a library (see *liburbi* on sourceforge).

A. 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 is composed of a first part between brackets displaying a timestamp in milliseconds (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 ':', preceding the command. With the 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 C++ 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. 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;
```

B. Modificators

Modificators 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 other movement parameters using *modificators*.

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` modificators 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, this controls the right wheel speed, in the "positive" direction:

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

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

One of the most interesting modificator is `sin`, followed by a time period and coupled with the `ampli` modificator, 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` modificator:

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

We will make a great use of this modificator to design walk patterns by superimposing several sinusoidal profiles. Note that a "sin-modified" assignment command never terminates, except if it is coupled with a `timeout` modificator: this timeout modificator ensures that the command will be terminated after a given time limit is reached:

```
> wheelR.val = 150 speed:120 timeout:2000;
```

This command means that the value 150 must be reached at speed 120. After 2000ms the command will stop, even if the targeted value has not been reached.

Interestingly, modificators can be variables whose value might change during the command execution, reflecting on the command execution. Complex interleaved assignments can be made by this mean.

An important point about modificators is that it is not only available to set devices but for any kind of variable (variables of a device or global variables). Let's consider the following example:

```
> myvariable = 0,
> myvariable = 50 time:10000,
> myvariable,
[001410040:notag] 2.45471445
> myvariable,
[001412020:notag] 12.35471445
> myvariable,
[001442120:notag] 50.00000000
```

The first affectation sets the variable to zero and the second one commands the variable to reach the value 50 in 10 seconds. When the value of `myvariable` is checked over time, it is evolving from 0 to 50 during this time interval. This is a unique and powerful feature of URBI compared to other existing languages and which makes it a fundamentally asynchronous and time-oriented language. It allows to create a dynamics for parameters, useful in many situations like, for example, in the design of a walk sequence for a legged robot.

C. Serial and parallel commands

One key feature of URBI 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
```

The operator priority is the following: ; , & |

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 can be found on `urbi.sourceforge.net`.

These time sequencing capabilities are another specificity of URBI and are very important features to design and chain complex motor commands or behaviors.

D. Loops, conditions, event catching

Several control structures are available, like the classical "`for`", "`while`" and "`if then 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 starts every iteration at the same time. "`for |`", "`while |`" and "`at &`" are also available. See III-E for an example.

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. It is possible to set a hysteresis threshold associated to the test so that the test has to be true n times before it can trigger the command. This is done with the tilde separator in the test and it is called a *soft test*. The following example, for Aibo, let the head move in diagonals with smooth movements, except when an object is detected in the 25cm short range:

```
period = 2500;
```

```
at (distanceNear.val > 25 ~ 3)
 scanning: loop {
   { headTilt.val = 90 smooth:period |
     headTilt.val = -90 smooth:period }
   &
   { headPan.val = 90 smooth:period |
     headPan.val = -90 smooth:period }
 }
else
 stop scanning;
```

In this example, the hysteresis threshold is set to 3, which means that the test must be true 3 times before it can trigger the `loop` command. The meaning of the `else` part is symmetrically identical. The `stop` command, followed by a tag name, means that any command with this tag will be stopped. This is used here to stop the head sweeping. Note how the serial and parallel operators are used to specify the head movement.

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.

E. Multiplexing

Another key feature of URBI is its capability to perform multiplexing of commands. 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 clients. For example, what should be done if one client wants the *neck* device to be set to 20 degrees while the other one requests a value of -20?

Six strategies are available in URBI:

- $[normal]$: The last command received is executed on top of others (default)
- $[discard]$: Ignore and erase any conflicting command
- $[cancel]$: A conflicting command replaces any 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

Each strategy can be selected via the parameterized `blend` instruction. 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`:

```
blend[mix] m;
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 below. 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.

F. 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 and files. We will not present those elements here, but extensive details can be found in [1], [2].

IV. Code examples on Aibo

URBI, which is a command script language, is normally supposed to be used together with a client program written in 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. To illustrate this point and some of the capabilities of the language, we present here a set of simple examples performing interesting action/perception loops on Aibo robots.

A. 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 by constantly setting the variables `camera.ballx` and `camera.bally` to the ball position in the image (between 0 and 1), and -1 otherwise. 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 (camera.ballx != -1) {
  headPan.val  = headPan.val +
    camera.xfov * (0.5 - camera.ballx) &
  headTilt.val = headTilt.val +
    camera.yfov * (0.5 - camera.bally)
};

at & (camera.ballx == -1 ~ 500ms)
 scan : {
  headPan.valn = 0.5 sin:4000 ampli:0.5 &
  headTilt.valn= 0.5 cos:4000 ampli:0.5
 };

at (camera.ballx != -1) stop scan;
```

The `valn` field is a normalized equivalent to `val`, based on the device *min* and *max* range.

”Ball Tracking Head” is a typical example given with the Sony OPENR SDK. The URBI version is comparatively much simpler to understand and requires only ten lines of code in URBI, compared to 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.

B. 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`.

C. Stand-up sequence

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

```
{ 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.

V. Walk sequences and perturbation-based turning

Walk sequences are good examples of simple applications of URBI for Aibo. We present here two types of walks and a method based on perturbations to make the robot turn.

A. Simple walk sequence

The simplest way of doing a walk sequence with URBI is to use basic sinusoidal movements on all joints of the legs:

```
s = 600;          // speed, as a period in ms
leg3.val = 130;   // knees up
legs.PGain = 24;  // P Gain for this walk

legLF1.val = 0   sin:s ampli:30 &
legLF2.valn= 0.1 sin:s ampli:0.2 phase:pi/2 &
legRH1.val = 0   sin:s ampli:30 phase:pi &
legRH2.valn= 0.1 sin:s ampli:0.2 phase:pi/2 &
legLH1.val = 0   sin:s ampli:30 &
legLH2.valn= 0.1 sin:s ampli:0.2 phase:3*pi/2&
legRF1.val = 0   sin:s ampli:30 phase:pi &
legRF2.valn= 0.1 sin:s ampli:0.2 phase:3*pi/2,
```

All sinusoidal oscillations are identical from one leg to the other, only the phase shift differs and makes the walk possible. A slight oscillation on joint 2 is used to reduce the leg contact on the floor when it is coming back in position and not pushing.

The effective speed of this simple walk is 16.6cm/s, which is not fast but still usable in simple cases.

B. Walk sequence based on Fourier decomposition

When motors `val` fields are set to the `add` blend mode, conflicting commands are summed. This feature can be used to superimpose several sinusoidal motion profiles, possibly resulting from a Fourier decomposition of a desired joint trajectory. We have done this analysis on the LRV robocup walk sequence [6], with only two of the main fourier coefficients. The command to start the corresponding URBI walk is using the "`for &`" construct and a set of pre-calculated parameters, given by the fourier analysis:

```
for &(x=1;x<=2;x++)
  for &(y=1;y<=2;y++)
    for &(j=1;j<=3;j++)
      for &(d=1;d<=2;d++)
       walk: robot.leg[x][y][j]=
                lrv.mean[:x][:j]
          sin:  lrv.s*lrv.coef[:d]
          ampli:lrv.amp[:x][:j][:d]
          phase:lrv.phase[:x][:y][:j][:d],
```

The colon before the index variables makes them *static*, which is necessary in the case of a "`for &`" construct applied to non terminating commands like the sinusoidal assignment because the values of the modificators are constantly reevaluated and the index value would change. The variables `robot.leg[x][y][j]` are arrays of aliases used to conveniently refer to the leg joints in the loops. See [2] for more details about static variables and aliases.

We have measured, on our floor type, a speed of 26cm/s for the original LRV walk. Our implementation, using only two Fourier coefficients, gives a comparable speed of 26cm/s and no visually noticeable difference in the way the walk is performed.

One interesting feature of URBI is that, as said before, the values of the different modificators are constantly reevaluated during the execution of the sinusoidal assignment. This makes it possible very simply to modify parameters online while the walk is executing, like changing the speed or amplitude of the movements. Those parameters can themselves be controlled by a sinusoidal or linear assignment, allowing walk acceleration and deceleration.

Of course, to generate a walk sequence, it is always possible to do some reverse kinematics calculations and send the corresponding motion profiles instant by instant to URBI, but the purpose here is to see how acceptable a walk sequence can be with native URBI sinusoidal commands and compare it to existing pre-calculated approaches in terms of performances and simplicity of the code. Our speed measures shows that the difference in performance is acceptable and the coding effort is limited.

C. Turning as a result of a perturbation

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.

We have designed a sort of "swing pattern" for the right and left turning, which makes the robot periodically swing on the right or on the left while staying immobile, which is exactly the movement corresponding to a rotation of the body while the legs are in contact with the ground:

```
// Turning Right
for &(x=1;x<=2;x++)
  for &(y=1;y<=2;y++)
   robot.leg[:x][:y][2] = 0
          sin:lrv.s
        ampli:lrv.turn,

// Turning Left
for &(x=1;x<=2;x++)
  for &(y=1;y<=2;y++)
   robot.leg[:x][:y][2] = 0
          sin:lrv.s
        ampli:lrv.turn
        phase:pi,
```

This "swing pattern" makes the robot turn as expected when it is started on top of a regular walk pattern. The perturbation affects only the second level joint (controlling the spacing of the legs from the body). The intensity of the movement, and the intensity of the turning, can be controlled with the `lrv.turn` amplitude. The correct values that work stand between 0 and 20. This online parameter modification provided by URBI allow for smooth and easy to implement online update of the rotation radius. Note that the only difference between a left and right turning is the relative phase of the perturbation.

The perturbation will work only if its phase is synchronized with the regular walk phase. This can be achieved by starting the two turning set of commands in parallel of the walk command and setting them to a sleeping state by having the amplitude equal to zero (for this, it is necessary to distinguish between `lrv.turnleft` and `lrv.turnright`). The other option is to synchronize *a posteriori* by prefixing the turn commands by a `wait` command on the value of one of the joints:

```
wait (robot.leg[1][1][2].val ~= 0) |
// the turning perturbation command...
```

The synchronization in that case will be only approximative since the test is not exactly triggered at the time the value is reached. The $\sim =$ test operator is a fuzzy equality test available in URBI.

The last option to do the synchronization is to use the `getphase` modificator, which specifies a given variable to constantly store the current value of the phase in a sinusoidal assignment. This variable can be used later to synchronize other sinusoidal assignments.

This simple perturbation method is not to be compared to state of the art reverse kinematics based methods, but shows some of the functionalities of URBI and illustrates the `add` blending mode in a practical situation, with the idea of superimposing motion profiles to get desired results. It will be further investigated to see other applications and compare the expressiveness and simplicity of the approach to its performances.

VI. Conclusion

We have presented the URBI language and some examples to illustrate its capabilities. The primary interest of URBI is to be a candidate to become a standard for robotic low level control. 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 way.

Practical applications and use of the current version of URBI have shown the validity of our approach both in terms of performance and ease of use. 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. URBI is simple to learn and simple to use, it is a low level language but includes scripting and procedural features that makes it extendable.

URBI 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 the robotic research community will find it useful and that this work will help and contribute to the rapid development of this domain.

Acknowledgments

We would like to thank Matthieu Nottale for his contribution to the walk sequence generation and his help on liburbi design.

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 language specification. *http://urbi.sourceforge.net*, 2005.

[3] Brian Gerkey 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.

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

[5] David S. Touretzky and Ethan J. Tira-Thompson. Tekkotsu: a sony aibo application development framework. *The Neuromorphic Engineer*, 1.2, 2004.

[6] O. Stasse V. Hugel, P. Blazevic and P. Bonnin. Trot gait design details for quadrupeds. *Technical report, http://www.lrv.uvsq.fr/research/ legged/papers/ tech_reports/2003/ 2003_symposium_paper.pdf*, 2003.