

URBI Tutorial for Urbi 1.0

(book compiled from 682M)

Jean-Christophe Baillie
Mathieu Nottale
Benoit Pothier

URBI Tutorial for Urbi 1.0: (book compiled from 682M)

by Jean-Christophe Baillie, Mathieu Nottale, and Benoit Pothier

Publication date

Copyright © 2006-2007 Gostai™

This document is released under the Attribution-NonCommercial-NoDerivs 2.0 Creative Commons licence (<http://creativecommons.org/licenses/by-nc-nd/2.0/deed.en>).

Table of Contents

1. Introduction	1
2. Installing URBI	2
Installing the memorystick for Aibo	2
3. First moves	4
Setting and reading a motor value	4
Setting speed, time or sinusoidal movements	5
Discovering variables	5
General structure for variables	6
Device values and .val alias	6
Making "global" variables	6
Expressions	6
Lists	6
Running commands in parallel	7
Conflicting assignments	8
Useful device variables and properties	9
Useful commands	9
4. More advanced features	10
Branching and looping	10
if	10
while	10
for, foreach	10
loop, loopn	11
Event catching mechanisms	11
at	11
whenever	12
wait, waituntil	12
timeout, stopif, freezeif	12
Soft tests	13
Emit events	13
Command tags, flags and command control	14
Objects grouping	15
Function definition	16
Error messages and system messages	16
5. Objects in URBI	18
Defining a class	18
Virtual methods and attributes	19
Groups	20
Broadcasting	21
6. The ball tracking example	23
Ball detection	23
The main program	23
Programming as a behavior graph	25
Controlling the execution of the behavior	26
7. Images and sounds	28
Reading binary values	28
Setting binary values	28
Associated attributes	29
Binary operation examples	30
8. The liburbi in C++	31
What is liburbi?	31
Components and liburbi	31
Getting started	32
Sending commands	32
Sending binary data and sounds	33
Receiving messages	33

Data types	34
UMessage	34
UValue	34
UBinary	35
USound	35
UImage	35
Synchronous operations	35
Synchronous read of a device value	35
Getting an image synchronously	36
Getting sound synchronously	36
Conversion functions	36
The "urbiimage" example	37
9. Create components: the UObject architecture	39
UObject	39
The basics	39
Adding attributes	40
Binding functions and events	42
Timers	43
Advanced types for binaries	43
The "load" attribute	44
The "remote" attribute	44
The colormap example	44
The practical side: how to use create an UObject?	48
10. Putting all together	54
Typical usages examples	55
A. Copyright	57

List of Figures

4.1. A typical motor device hierarchy	15
6.1. The ball tracking behavior graph	25
10.1. The general URBI architecture, putting all together	55

Chapter 1. Introduction

URBI (Universal Real-time Behavior Interface) is a scripted interface language designed to work over a client/server architecture in order to remotely control a robot or, in a broader definition, any complex system. As it will be shown in this tutorial, URBI for robotics is more than a simple driver for the robot, it is a universal way to control the robot, add functionalities by plugging software components and develop a fully interactive and complex robotic application in a portable way.

The main distinctive qualities of URBI are the following:

- **Simplicity:** easy to understand, but with high level capabilities, makes it suitable both for educational and professional applications.
- **Flexibility:** independent of the robot, system, OS, platform, interfaced with many languages (C++, Java, Matlab,...)
- **Modularity:** object based component architecture is available to extend the language. The components can be remote or plugged in the URBI Engine, they can be written in any language.
- **Parallelism:** Parallel processing of commands, concurrent variable access policies, event based programming,...

Probably one of the most important points for this tutorial is the first one: URBI has been designed from the beginning 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. The way URBI has been designed is to have layered levels of complexity: the more complex your application is, the more complex things you have to learn, but simple applications remain simple to develop. If all you want is to move the robot joints, you can do that in one minute. And if you want to build AI applications, the tools are there for you to do so.

URBI is available with many robots and the number is increasing. Currently, there is an URBI version for Aibo, for the HRP-2 humanoid robot, for the Webots universal simulator and the Pioneer robots, the Philips iCat robot, and other humanoids are on the way.

The Webots simulator compatibility means that it is possible to switch from the real robot to simulation with a simple IP address change, and this makes URBI particularly suitable for applications that need to frequently go back and forth between real/simulated robots.

In this tutorial, we have tried to make a step by step description of URBI which goes from simple motor commands up to more complex programming including software components integrated in URBI. It is meant to be understandable by people having little or no background in robotics and programming (except for the C++ sections, which require that you understand C++ at a basic level). However, from time to time, we have inserted explanations or complements that will probably make sense only for advanced users or academics/industrials. These inserts are presented with a small academic sign as shown on the left of this text.

Chapter 2. Installing URBI

We cannot detail in this tutorial how to install URBI for any particular robot type, but the general idea is to have the URBI server program loaded and running on your robot. The process to do so should be described in the INSTALL file of the package you have downloaded. In the ideal situation, URBI is preinstalled on your robot anyway.

Since we will use many Aibo examples in the tutorial, we give here the instructions on how to install URBI on an Aibo robot. We also describe how to install URBIlab which is a simple and convenient cross-platform graphical client to replace telnet.

Installing the memorystick for Aibo

First, download the precompiled memorystick for your specific robot. There are two possibilities at the moment:

- ERS2xx : <http://www.urbiforge.com/ers200>
- ERS7 : <http://www.urbiforge.com/ers7>

Quick instructions:

Unzip the archive and put the content of the MS-xxx directory on a blank memorystick, updating the WLANCONF.TXT file with your specific network config.

Detailed instructions:

1. Untar/Unzip the memorystick archive corresponding to your Aibo. You should get a directory named MS-ERS7 or MS-ERS200. Enter into this directory.
2. From the MS-ERS7 (or MS-ERS200) directory, go to the OPEN-R/SYSTEM/CONF directory. There should be a WLANCONF.TXT file here (or you must create it), to configure the network properly. There is no official documentation on the how to write the WLANCONF.TXT file, but here is an example that you can customize for your robot:

```
HOSTNAME=aibo.mydomain.com
ETHER_IP=192.168.1.111 # <- your IP here
#
# WLAN
#
ESSID=0a3902 # <- your SSID here
WEPENABLE=1 # <- WEP or not
WEPKEY=0x4B2241785B # <- the key: hexa
#WEPKEY=ABCDE # <- ASCII with ERS2xx
APMODE=1

#
# IP network
#
USE_DHCP=0
SSDP_ENABLE=1

# This part can be omitted
# Your network config here ->
```

```
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.0.3
DNS_SERVER_1=192.168.1.1
```

You can use URBI on Aibo without the network if you don't have a wifi access point, by putting your URBI programs in the URBI.INI file.

3. Copy the content of the MS-ERS7 or MS-ERS200 directory in the root of a blank programmable pink memorstick (a "PMS")¹. Be careful that this is **not** the Aibo Mind memorstick or one of the blue memorsticks: actually, you must go and buy a specific aibo programming memorstick from Sony, it is unfortunately not included in the Aibo package. Then, put this memorstick in the robot and start it. Your URBI robot is ready.

You can run telnet² on port 54000 of the robot to check if everything is OK:

```
telnet aibo.gostai.com 54000
```

You should get a URBI Header at start, which looks like this:

```
[00020380:start] *** *****
[00020380:start] *** URBI Language specif 1.0 - Copyright (C) 2006 Gostai SAS
[00020380:start] *** URBI Kernel version 1.0 rev. 100
[00020380:start] ***
[00020380:start] *** URBI Engine 1.0 for Aibo ERS2xx/ERS7 Robots
[00020380:start] *** (C) 2004-2006 Gostai SAS
[00020380:start] ***
[00020380:start] *** URBI comes with ABSOLUTELY NO WARRANTY;
[00020380:start] *** This software is free, and you are welcome to use
[00020380:start] *** it under certain conditions; see LICENSE for details.
[00020380:start] ***
[00020380:start] *** See http://www.urbiforge.com for news and updates.
[00020380:start] *** *****
[00020380:ident] *** ID: U595075704
```

One interesting benefit of the client/server architecture of URBI is that you can start right away to send commands to your robot with a simple telnet client. It is of course possible and desirable to interface URBI with a C++, Java or Matlab program, which will be described later with the liburbi (chapter "The liburbi in C++"), but for the moment we will use a simple telnet interface.

However, telnet is a very crude and limited client (which does not always work well under Microsoft Windows³), and we have developed a cross-platform graphical alternative called URBI Remote that you are encouraged to use. URBI Remote is free and released under a GNU-GPL License. You can download it here (available mid 2007):

http://www.urbiforge.com/index.php?option=com_content&task=view&id=75&Itemid=136

Other free third-partie graphical interfaces, like "Aibo-Telecommande" can be downloaded right now on urbiforge.com.

³ It works if you use cygwin, otherwise carriage returns are badly interpreted by the native Windows implementation of telnet

Chapter 3. First moves

In the following, we will use examples from the Aibo robot, but you can easily transpose them to your particular robot. Each element of the robot (sensors, motors, camera,...) is an object and it has a name. In the Aibo, you have objects for the head motors called headPan and headTilt. The camera object is called camera. In the following will often use the term 'device' to refer to an object that handles some piece of hardware in the robot. We have the camera device, the motor devices, etc.

You can find out what devices are available for your particular robot by checking the associated URBI Doc online documentation (<http://www.gostai.com/doc.php>), or simply by typing the command group objects;

Setting and reading a motor value

We will make use of the motors in the following, so first of all we have to start them:

```
motors on;
```

"motors off" is of course also available and you can on/off any device (or more generally, objects) with "device_name on/off". Now, let's start by moving the headPan motor to 30 degrees:

```
headPan = 30;
```

Now, let's ask what is the value of the headPan device:

```
headPan;  
[139464:notag] 30.102466
```

The server responds with a server message (written in italic font here to make it easier to distinguish it from commands) prefixed by a timestamp and a tag between brackets. Since there is no tag associated to the command in this example, notag is used by default. It is very simple to associate a tag to a command in URBI by prefixing the command with the tag and a colon:

```
mytag:headPan;  
[139464:mytag] 30.102466
```

The message has now the mytag tag. This will be crucial to know who is sending what when several commands are running in parallel, or to stop commands that are running in the background.

You can try to set different motors, like legRF1 or tailTilt, or play with LEDs like ledF1 or ledBMC, or read sensor values like the distance detector distanceNear or the accelerometer accelX, accelY, accelZ. The syntax is always the same: device = value;.

What is really behind the scene here is not device = value, but device.val = value; which is actually setting the val variable of the device object to 'value'. But to make life simpler for beginners, all devices in aibo have an alias which looks like that:

```
alias headPan headPan.val
```

So, you won't see the hidden .val which is not necessary in normal operations and for beginners. You can remove those aliases (which are defined in URBI.INI) with unalias.

Setting speed, time or sinusoidal movements

The above examples set the value of the device as fast as the hardware of the robot allows it. Of course, you might want to do more complicated things like reaching a value in a given time (in milliseconds):

```
headPan = 30 time:3000;
```

Which will reach the value 30 (degrees) in 3000ms. Whenever you need to express a time value in URBI, you can explicitly use units like this:

```
headPan = 30 time:3s;  
headPan = 30 time:3000ms;  
headPan = 30 time:3m;  
headPan = 30 time:3h26m15s;
```

You can assemble days (d), hours (h), minutes (m), seconds (s) and milliseconds (ms), with decimal values. By default the unit is milliseconds if no unit is specified or when a variable expression is used.

Alternatively, you can also set the speed used to reach the value, expressed in *unit/s*:

```
headPan = 30 speed:1.4;
```

Or the acceleration (expressed in *unit/s²*):

```
headPan = 30 accel:0.4;
```

One very useful way of assigning a variable with a dynamic profile is to use a sinusoidal oscillation:

```
headPan = 30 sin:2s ampli:20,
```

This will make the headPan device oscillate around 30 degrees with an amplitude of 20 degrees and a period of 2s. Note that the command ends with a comma and not a semicolon. We will explain why later, but the reason is that the sinusoidal assignment never terminates and the comma sort of "puts it in background" to allow other commands coming after it to be executed. Otherwise, with a semicolon, nothing coming after this sinusoidal assignment could be executed since the command never ends. This is a common mistake by beginners using URBI.

time, speed or sin are called modifiers. Many other modifiers are available like phase, getphase or smooth. Check the URBI Language Specification for a comprehensive description of modifiers, or just play with them to see what they do.

One particularly powerful modifier is function which assigns an arbitrarily complex function of time as the variable trajectory. This is described in the URBI Language Specification and will only be available in servers with kernel 2.0 or above.

Discovering variables

You can use variables in URBI. Simply assigning a value to x will create a variable x local to your connection:

```
x = 4;  
x;  
[146711:notag] 4.000000
```

General structure for variables

In URBI kernel 1.0, variable names are always of the form `prefix.suffix` and when no prefix is supplied, a prefix local to the connection is silently added so that `x` in one connection will not interfere with `x` in another connection.

For example, when you type `x URBI` will in fact use `U596851624.x` in its memory, with `U596851624` being the identifier of your current connection (where you typed `x` in). In the same way, function calls have a local namespace attributed, so that you can do recursive function calls without interferences. This will be redesigned in URBI 2.0 with advanced name resolution and name space support.

Device values and `.val` alias

As we already said before, there is one important exception to the rule saying that variables without prefixes are local: when you type `headPan`, URBI do not treat this as a local variable, but instead it applies an alias that transforms the expression into `headPan.val`, which is a standard URBI variable containing the device value. So, in reality, `headPan` do not refers to a local variable but to the global variable `headPan.val`. Aliases are usually defined in the `URBI.INI` file.

Making "global" variables

There is no real concept of local or global variable in this version of URBI, as we have explained. Everything is of the form `prefix.suffix`. Without prefix, the variable is local to the connection but you can use your own prefix to make your variable "global":

```
myprefix.x = "hello";
```

Actually, `myprefix` can be seen, and also defined, as an URBI object, as we will describe it in the chapter "Objects in URBI" which details the object oriented features of URBI.

Expressions

Note that the type of the variable (numeric, string, list or even binary as we will see later) is automatically inferred by URBI.

You can evaluate arbitrary complex expressions, including variables or known functions like `sin`, `cos` or `random` (see the URBI Specification for the full list):

```
x=pi/2;
calc:sqrt(1+sin(x));
[148991:calc] 1.414213
```

One interesting feature is that modifiers in complex assignments are constantly reevaluated so that if they contain variables, the value of the modifier might change over time as the corresponding variable is evolving. Consider the following example which assigns to `x` a sinusoidal oscillation within a sinusoidal envelop between 15 and 25:

```
the_amplitude = 20 sin:10s ampli:5,
x = 0 sin:2s ampli:the_amplitude,
```

Complex interactions between variables and devices value can be established with this feature.

Lists

You can store several elements in a list with URBI, simply by putting them between brackets:

```
mylist = [1,2,35.12,"hello"];
mylist;
[139464:notag] [1.000000,2.000000,35.120000,"hello"]
```

You can easily add elements or add a list:

```
mylist = [1,2] + "hello";
mylist;
[146711:notag] [1.000000,2.000000,"hello"]
x = 1;
mylist + [45,x];
[148991:notag] [1.000000,2.000000,"hello",[45.000000,1.000000]]
```

Then, you can scan the content of a list with a foreach command:

```
list = [1,2];
foreach n in list { echo n };
[151228:notag] *** 1.000000
[151228:notag] *** 2.000000
```

Note that, for technical reasons, the code executed in the foreach command must be enclosed between brackets, even if there is only one command in it.

You can also directly access an element of a list with its position, like in an array:

```
mylist = [1,2,"hello"];
mylist[2];
[146711:notag] "hello"
```

If you have lists containing lists, you can use multiple indexes like mylist[3][4] to access sub-elements.

Finally, you can also "traditionnaly" get the first element with head and the rest of the list (excluding the first element) with tail:

```
mylist = [1,2,"hello"];
head(mylist);
[146711:notag] 1.000000
tail(mylist);
[146711:notag] [2.000000,"hello"]
```

Running commands in parallel

Commands in URBI can last during a certain amount of time, this is completely new compared to most other languages. We have seen so far that we can assign values with a certain time or with a certain speed, or even assign values in a sinusoidal way, which lasts forever. There are many ways in URBI to get these commands to run in parallel. We have already seen how to do it by using a comma to separate commands instead of a semicolon.

There is another way to specify that commands should be run in parallel, by using the & separator:

```
x=4 time:1s & y=2 speed:0.1;
```

The difference with the comma separator is that & forces the two commands to start at exactly the same time. In particular this means that the first command cannot start until the second command is fully

available. So, typing `x=4 time:1s &` in the console will not start anything, because URBI is waiting for what comes next, after the `&` (that's why we have the comma separator, which is less constraining and allow you run commands interactively).

In the same way commands can be run serially, exactly one after the other, by using a pipe separator:

```
x=4 time:1s | y=2 speed:0.1;
```

There will be no time gap between the two commands so, here again, URBI waits for the second command to be available: unlike the semicolon separated commands, the second command must start exactly after the first, so it must be ready in advance.

Using semicolon or comma separators is more permissive, because it will start immediately any command standing before the separator. But you might need strong time synchronization constraints, and that's why `&` and `|` separator are here for.

Note that you can group commands between brackets and build a more complex architecture of parallel and serial commands, like this:

```
{ { x=4 time:1s | y=2 speed:0.1 } & z=0 sin:200ms ampli:4 } | t=2,
```

TIP: In general, it is a good idea to end commands entered in a console (URBILab or telnet) by a comma, to avoid blocking the connection after entering a never-ending command.

Conflicting assignments

Since it is possible to run commands in parallel, possible conflicts might arise. For example, what will happen if something like this is executed?

```
x=1 & x=5;
```

`x=5` is a conflicting assignment since it accesses the variable `x` at the same time together with the first assignment. URBI has several blending modes to handle these conflicts, and you can specify these blending modes with the `blend` property of the variable. For example:

```
x->blend = add;
```

This will tell URBI to add the numerical values of any conflicting assignments on `x`. So, the result of the above command will be `6`. There is also a `mix` mode available, which does an average of conflicting assignments (the result would be `3`) and a `queue` mode which will queue conflicting assignments (the result will be `5`). Other blending modes are available and described in the URBI Language Specification.

Variables in URBI have properties which can be accessed with a `->` redirector. Properties are not identical to object attributes, they are part of the language semantics and therefore cannot be redefined. There are many properties available, like `rangemin`, `rangemax`, `speedmax`, `delta`. They are described in the URBI Language specification.

Note that blending modes also apply for sound devices, like `speaker` on the Aibo, and changing its blending mode from `mix` to `queue` will either superimpose sounds or queue them when they are played together.

The `add` and `mix` modes are very useful to superimpose sinusoidal assignments to design complex periodical movements, using a Fourier transform of the signal and keeping only the most significant coefficients.

Useful device variables and properties

For the Aibo robot, and for most standard robots, you will find the following motor device variables useful (replace device by the actual device name):

- `device.load` : sets the torque power in a joint, between 0 (totally loose) and 1 (rigid).
- `device.PGain` : set the P gain of a joint in the associated PID.
- `device.IGain` : set the I gain of a joint in the associated PID.
- `device.DGain` : set the D gain of a joint in the associated PID.

You also have useful properties, which are not variables in a strict sense (properties are part of the language semantics), but you can read and set them:

- `device->rangemin` : minimal value of the device
- `device->rangemax` : maximal value of the device
- `device->delta` : precision of the device, used in fuzzy tests
- `device->unit` : unit of the device (for information only in URBI 1)
- `device->blend` : the device blend mode (normal, mix, add, queue, discard, cancel)
- `device->info` : some information about the device.

Note that the above properties are not properties of the device, but they are in fact properties of the `device.val` variable, since we still assume here that aliases are defined on the device name.

Useful commands

Here is a short list of useful commands that you might need in your URBI programs:

- `reset` : does a virtual software reboot of the robot. Useful to erase a set of scripts and send a new version in the development stage.
- `stopall` : stop all commands in every connections. A bit radical, but useful sometimes.
- `reboot` : reboot the robot.
- `shutdown` : stops the robot.
- `uservars` : display a list of the user variables.
- `strict` : start the strict variable definition control policy (see the URBI Language Specification).
- `unstrict` : cancels the effect of `strict`.

Chapter 4. More advanced features

At this point, you are already capable of reading and setting sensors and motors in your robot, execute complex scripts or actions and superimpose motion patterns. This could be already enough for most users, but there is more in URBI and the URBI language gives you access to all the programming constructs found in modern languages plus other new constructs useful for robotics.

Branching and looping

The branching and looping constructs of C/C++ are also available in URBI: if else, for, while. The following examples illustrate these constructs (the echo command that you will see simply displays the expression as a system message).

if

if performs a single test and executes the associated command if the test is true:

```
if (backSensorM > 0) {
    pressed = 1;
    echo "Back sensor pressed";
};
```

Note that the last command between brackets doesn't need to be ended by a semicolon like in the above example. This is because semicolons are command separators and not command terminators. You can put a semicolon at the end like in C, but it is not required and it has no effect (it adds an empty command).

```
if (distance < 10)
    echo "Obstacle detected"
else
    echo "No obstacle";
[167322:notag] *** No obstacle
```

Note that, unlike in C, there is no semicolon before else, but there is a semicolon (or any other command separator) after the concluding }.

distance and backSensorM are two Aibo devices related to the head infrared distance sensor and to the middle (M) back sensor.

while

The while construct is similar to what is available in C:

```
i=0;
while (i<=2) {
    i:echo i;
    i++;
};
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

for, foreach

The for construct is similar to what is available in C:

```
for (i=0;i<=2;i++)
  i:echo i;
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

Unlike in C, URBI has specific constructs to handle parallel and serial loops: `for&`, `for|` and `while|`. These constructs will start every iteration in parallel (with `&`) or in series (with `|`) with a guaranteed time constraint. More details are available in the URBI Language Specification.

As we already mentioned before, there is also a `foreach` and `foreach&` construct to iterate lists:

```
foreach i in [0,1,2] {
  i:echo i;
};
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

`foreach` is an exception: even when the iterated command is a single command, like in the above example, you must enclose it between brackets.

loop, loopn

For practical reasons, URBI has added two more constructs, `loop` and `loopn` to create infinite loops in the first case and loops iterating n times in the second case. The syntax is:

```
loop { ... }

and

loopn (n) { ... }
```

Event catching mechanisms

at

`at` works a bit like `if`, expect that it is always running in the background:

```
at (distance < 50)
  echo "Obstacle appears";
```

The `echo` command in the above example will start at the time when the test becomes true, only once. To be more precise, `at` triggers the command when the test switches from false to true. It is very useful to start an action when a condition is met to react to this condition. If you run the above code, the message "Obstacle appear" will be displayed once when you move your hand in front of the Aibo.

`onleave` is a bit like `else` and is followed by an action that will be executed when the test switches from true to false:

```
at (distance < 50)
  echo "Obstacle appears"
onleave
  echo "The obstacle is gone";
```

whenever

whenever works a bit like while, except that it never terminates and run in the background:

```
whenever (distance < 50)
  echo "There is an obstacle";
```

The echo command will be executed whenever the test is true. Then, it will be executed again if the test is still true, and so on, until the test becomes false. Whenever the test switches to true again, the loop restarts and the command is executed. Compared to the at example above, the difference is that the message "There is an obstacle" will be displayed several times, as long as you leave your hand near the head of the robot.

Alternatively, you also have an else construct available to specify something to do when the test is false:

```
whenever (distance < 50)
  echo "There is an obstacle"
else
  echo "There is no obstacle";
```

whenever and at are the two fundamental constructs that you will use when doing reactive programming and event catching mechanisms on your robot.

wait, waituntil

The command wait (*n*) will wait for *n* milliseconds before ending. It is useful to have a temporal break in a series of commands, typically motor commands:

```
headPan = 0 | wait(1s) | headPan = 90;
```

The command waituntil(*test*) waits until the test becomes true and can be useful to synchronize different parallel programs on a given condition.

timeout, stopif, freezeif

The command timeout (*n*) cmd will execute the command cmd and stop it after *n* milliseconds if it is not already finished.

```
timeout(10s) loop legRF2 = legLF2;
```

The command stopif (*test*) cmd will execute the command cmd and stops it when the *test* becomes true. Of course, if the command is already finished, nothing special happens.

```
stopif(distance<50) robot.walk();
```

The command freezeif (*test*) cmd will execute the command cmd and freeze it when the *test* becomes true. When the test is false again, cmd is unfreezed.

```
freezeif(!ball.visible) trackball();
```

This can be very useful to specify that certain portions of code should run only when certain conditions are met.

Soft tests

The tests used in event catching commands like `at`, `whenever`, `waituntil`, `stopif` or `freezeif` can be associated to time constraints, becoming "soft tests":

```
at (headSensor >0 ~ 2s)
  echo "Head touched...";
```

This means that the test has to be true for 2 seconds before it becomes actually true for the `at` command. You can specify the time in *s* or *ms* by using the appropriate suffix and it should be separated from the test by a tilde `~`

Soft tests are usable with any event catching command and they are very useful in robotics as a simple noise filter for sensor inputs.

Emit events

Event programming is a very useful feature and a good way of doing robot programming. The basic idea of event programming is that some command emit an event and some other catches this event and do something.

Simple events

To emit an event, the `emit` command is available in URBI, and you can use `at` or `whenever` to catch it:

```
at (boom()) echo "boom!";
  emit boom;
[139464:notag] *** boom!
```

Note that the `boom` event here is local to the connection. If you want to make the event visible from other connection, you should use a prefix, like `myprefix.boom`.

Events with parameters

You can add parameters to events like this:

```
emit myevent(1,"hello");
```

The parameters can be retrieved when the event is caught:

```
at (myevent(x,y))
  echo "catch two: " + x + " " + y;

at (myevent(1,x))
  echo "catch one: " + x;
```

The second `at` here is interesting as it is doing a filtering on the event parameters, accepting only events whose first parameter equals 1:

```
emit myevent(1,"hello");
[146711:notag] *** catch two: 1.000000 hello
[146711:notag] *** catch one: hello
emit myevent(2,15);
[148991:notag] *** catch two: 2.000000 15.000000
```

Event duration

Events usually have a virtually null duration, they are just spikes (Dirac functions of time). However, you can explicitly request that an event lasts for a certain duration by specifying this duration between parenthesis like this:

```
emit(10s) boom;
emit(15h12m) myevent(1, "hello");
```

This will make a difference between at and whenever event catcher for example: whenever will loop during the whole event duration.

Pulsing events: the every command

You can have any command repeated at specific time intervals in URBI, using the every command. The following example will say "hello" every 10 minutes:

```
every (10m) echo "hello";
```

One typical usage is to use the every command to pulse events at regular intervals:

```
every (100ms) emit pulse;
```

To stop the emission, just use stop on the every command with the appropriate tag :

```
mypulse:every (100ms) emit pulse;
stop mypulse;
```

Command tags, flags and command control

The tagging mechanism described in the beginning of this tutorial is actually more than just a message tagging facility. For example, you can stop any running command with the stop command, from any connection:

```
myloop:loop legRF2 = legLF2,
...
stop myloop;
```

You can also freeze a command with the freeze command and unfreeze it (it will restart where it was before freezing) with the unfreeze command. There is also a block/unblock pair of commands to block new commands with a given tag and prevent them to be executed. Note that tags can prefix a set of commands between brackets, like { ... }, and it can be associated to large portions of code, not only single commands.

Next to the tag, it is possible to use one or more flags. Flags are keywords prefixed by a + sign. The most useful flags are +begin and +end which send a system message when the command starts or stops, or +bg which puts the command in background. Here are a set of illustrating examples:

```
mytag+begin:
loop legRF2 = legLF2,
[139464:mytag] *** begin
```

```
+begin+end:
wait(1s);
```

```
[521200:mytag] *** begin
[522200:mytag] *** end
```

More flags are described in the URBI Language Specification.

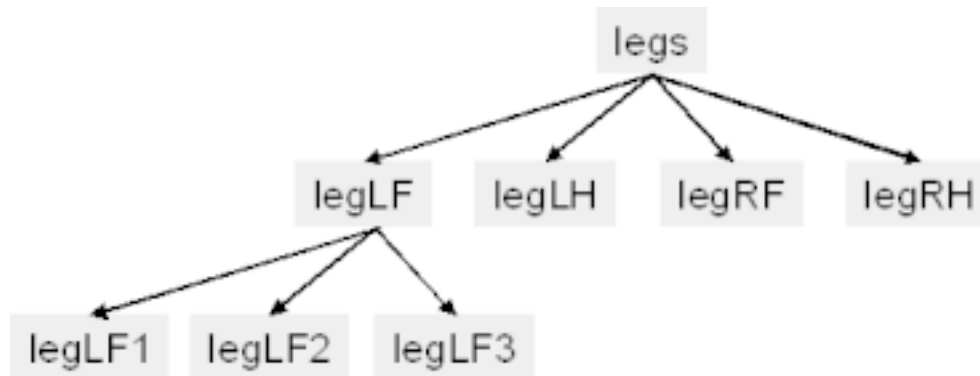
Since URBI 1.0, you can use hierarchical tags like `mytag.subtag`. The advantage is that you can stop a whole hierarchy based only on the highest tag: the above tag can be stopped with a `stop mytag.subtag` and a `stop mytag` as well, and you can group commands more easily with this mechanism. Future version will also include multi-tagging to even increase the possibilities.

Objects grouping

An important feature of URBI is the capacity to group objects into hierarchies. This is done with the `group` command: `group groupname { object1, object2, ... }`, for example:

```
group legLF {legLF1, legLF2, legLF3};
group legs {legLF, legLH, legRF, legRH};
```

Figure 4.1. A typical motor device hierarchy



This grouping feature is associated to the notion of "broadcasting", which is used for several things. One is to make multi-object assignments: any assignment is executed for the group and is recursively passed to child subgroups. In other words, using the example above, the command `legLF.val = 0` will set the value of `legLF1.val`, `legLF2.val` and `legLF3.val` to 0 (note that aliases work also here if you want).

```
group ab {a,b};
ab.n = 4;
a:a.n, b:b.n,
[167322:a] 4.000000
[167322:b] 4.000000
```

For any robot, there will usually be a hierarchy of object grouping available at start. This is usually done in the URBI.INI file, or `std.u`.

For example, with the Aibo, there is a `motors` group to store all motors and a `leds` group which contains all led devices. You can easily set every LEDs to a random value with a command like this:

```
leds = random(2); // alias applies here
```

For fun, you can run something like: `fun: loop leds = random(2)`, and see the result.

There is more about groups and broadcasting, which is a very powerful feature of URBI. We will come back on this subject in the chapter "Objects in URBI".

Function definition

To define functions, you will be using the function keyword, followed by the function name in prefix.suffix notation (or simply suffix for a function local the connection), and the parameters between brackets (or an open/close bracket () if there is no parameters). You can use return to return a value or to exit from the function, like in C:

```
function adding(x,y) {
  z = x+y;
  return z;
};
```

```
function print(x) {
  echo x;
  if (x<0)
    return
  else
    echo sqrt(x);
};
```

Note that there must be a semicolon or another command separator after the function definition, since defining a function is a command like any other command in URBI.

The parameters are always local to the function call. Non-global (i.e. without prefix) variables in the function body are also local to the function call. Consider the following example:

```
a=4;
b=5;
function display(b) {
  display_b:b; // b is local
  var a=b; // creates a local variable a
  display_a:a;
};
display(10);a:a;b:b;
[139464:display_b] 10.000000
[139464:display_a] 10.000000
[139464:a] 4.000000
[139464:b] 5.000000
```

A good idea is to put all your functions in a separate file like "myfunc.u", and load them with the load command: load("myfunc.u"); This can be done from the URBI.INI file for example, or when you actually need them.

To undef a function, simply use:

```
delete myfunction;
```

Error messages and system messages

When a command fails in URBI, it will send an error message, prefixed by three exclamation marks:

```
impossible:1/0;
[167322:impossible] !!! Division by zero
[167322:impossible] !!! EXPR evaluation failed
```

Note that the tag of the command is used in the error message, which is extremely convenient to know what has failed in a complex program.

Error messages are different from system messages prefixed by three stars, and which usually display information normally outputted or requested by the command. A typical example is echo with a +begin and +end flag:

```
mytag+begin+end:echo "hello there!";  
[146711:mytag] *** begin  
[146711:mytag] *** hello there!  
[146711:mytag] *** end
```

Chapter 5. Objects in URBI

Object oriented programming is integrated in URBI, with many innovative features like virtual attributes and broadcasting. This chapter introduces the most important features of objects in URBI. It can be skipped by novice programmers but it is not very complex and can be useful reading for everyone.

Defining a class

Like in C++, you define a class in URBI with the class keyword:

```
class myclass;
```

You can of course define what will be in the class, including three types of elements: variables, functions and events:

```
class myclass {
  var x;
  var y;
  function f(a,b);
  event signalmyself(s);
};
```

It is important to notice here that, unlike C++ classes, myclass in the above example is also an instance¹ and you can very well assign values to myclass.x and use it.

One important function that you might want to define is init, which is the class constructor (this is another difference with C++, the constructor is not named with the class name). This function should return nothing or return 0 to indicate success or any other value to indicate failure.

To define the body of a class method, you should do it outside the class definition, like this:

```
class myclass {
  var x;
  function init(a);
};

function myclass.init(a) {
  x = a;
};
```

You can define a subclass (or instance, remember there is no difference), with a new command, like in C++:

```
mysubclass = new myclass(42);
```

This will create mysubclass and call mysubclass.init(42);

mysubclass inherits from myclass, so every attribute or method of myclass is also available in mysubclass, we will see in the next section how you can have default definition, or re-definition, and how they are handled².

¹This is called a prototype-based object oriented language, like javascript

²There is no public/private/protected accesses in the current version of URBI, but it will be integrated in version 2.0

Note that `mysubclass` can inherit from several classes by calling `new` on these different classes:

```
mysubclass = new myclass (42);
mysubclass = new myotherclass ();
```

This is a very special way of dealing with multiple inheritance, compared to C++.

Calling `new` without parenthesis, just the class name, will execute the `init` constructor with no parameters:

```
mysubclass = new myclass; // same as new
myclass();
```

Note that if `init` is not defined, or if `init` returns a value indicating a failure (non void and not zero), an error message will be output and the class creation will fail.

Classes can be extended at runtime, simply by creating new functions or new attributes related to them. For example:

```
class myclass {
  var x;
};
var myclass.newattribute;
myclass.s = "hello";
...
function myclass.f(a) {
  s = a;
};
```

Virtual methods and attributes

In URBI, every method and every attribute is virtual, which means that if your class redefines it, it becomes its own definition, otherwise the definition (or value) of the parent class will be used.

Consider the following example:

```
class myclass {
  var x;
  function f();
};
function myclass.f() {
  echo "I'm in myclass";
};
```

Then:

```
sub = new myclass;
sub.f();
[139464:notag] *** I'm in myclass
```

The definition of `f` is retrieved from `myclass`. Now, we can redefine it:

```
function sub.f() {
```

```
    echo "I'm in sub!";  
};
```

And call it again:

```
sub.f();  
[139464:notag] *** I'm in sub!
```

In the same way, attributes get their value from the parent class, unless redefined in the child class. The following example illustrates the case with the above myclass and sub prototypes:

```
myclass.x = 1;  
sub.x;  
[146711:notag] 1.000000  
sub.x = 4;  
sub.x;  
[146711:notag] 4.000000  
myclass.x;  
[146711:notag] 1.000000
```

Groups

We have already seen in previous chapters how groups can be used as a way to assign values to several object variables at the same time. In fact, the mechanism is more general and associated to the concept of broadcasting that we will define precisely in the next section.

First, a few words about groups. We have already seen how we can define groups with the group command. In the same way, you can simply add a member in an already existing group with the addgroup command, and remove one with delgroup, which allows you to handle dynamic group creation, if you ever need to:

```
group a {a1,a2};  
addgroup a {c,d};  
delgroup a {a1,d};
```

You can examine the content of a group by invoking the group command with the group name only:

```
group a {u,v,b};  
group a  
[146711:notag] ["u","v","b"]
```

Group subgrouping is possible, in that case the group content evaluation will return the list of terminal members only:

```
group a {u,v,b};  
group b {x,y};  
group a  
[146711:notag] ["u","v","x","y"]
```

A classical usage of the above feature is to iterate through a list of device objects, like motors, who have been gathered in the same motors group:

```
foreach m in group motors {
```

```
$(m) = ...  
}
```

Note: The \$ construct will return the variable whose name is the string given as parameter. In the above example, we suppose that there is a .val alias, otherwise you would use: \$(m+".val")

Now, we will see how you can make practical use of groups with broadcasting.

Broadcasting

When you execute a command at the level of a group, which can be a function call or an assignment, the command will be propagated in parallel to each element of the group and their subgroups. This is called broadcasting. This can also apply to classes, since you can define a group taking care of every sub class instantiation. A conventional practice is to name the group associated to a class with the plural of the class name, usually adding a simple 's'.

First, let's see how assignments are broadcasted. Consider the following example:

```
class a;  
a1 = new a;  
group as {a,a1};  
a1.x = 42;  
as.x = 4;  
a.x;  
[139464:notag] 4.000000  
a1.x;  
[146711:notag] 4.000000
```

Broadcasting on functions works similarly:

```
class a {  
  var x;  
  function f();  
};  
  
function a.f() {  
  echo x;  
};  
a1 = new a;  
group as {a,a1};  
a.x = 1;  
a1.x = 2;  
as.f();  
[139464:notag] *** 1.000000  
[139464:notag] *** 2.000000
```

The above function call on f is in fact executed as:

```
a.f() & a1.f();
```

Broadcasting is duplicating the commands in parallel.

As usual, subgroups are explored in the process.

Broadcasting functions can be very useful to execute tasks in parallel in a group of objects, without having to use for& or similar constructs. Broadcasting and inheritance complement each other, so when

the broadcasting is finished, the function definition can be searched upward in the class hierarchy, like in this example:

```
class a {
  var x;
  function init(v);
  function f();
};
function a.init(v) { x=v; };
function a.f() {
  echo x;
};
a1 = new a(1);
a2 = new a(2);
a3 = new a(3);
function a1.f() { echo "I'm different!"; };
group oneandtwo {a1,a2};
oneandtwo.f();
[139464:notag] *** 2.000000
[139464:notag] *** I'm different!
```

Broadcasting is clearly a new feature in the hands of programmers. You might or might not use it, but we believe that it will help to make many codes more concise by grouping logical actions in one line, instead of using for loops or similar iterating concepts. It also makes clear that certain actions should be executed in parallel on a group of objects, which is semantically meaningful.

Chapter 6. The ball tracking example

The best way to learn a new language is to study simple examples to see what can be done in practice. In this tutorial, we will concentrate on the red ball tracking application on the Aibo which is interesting because it is a simple behavior with two states and it involves a perception/action loop which is very typical of robotic applications. We will see how URBI can help to control the execution of the behavior in a simple way with command tags.

Ball detection

Detecting a ball involves image processing and cannot be written directly in URBI for obvious efficiency reasons. The best way to provide such algorithmic components (like visual processing or sound processing) is to write a UObject Component in C++, Java or Matlab, and to plug it in URBI. We will not describe at this stage how to write such a component, but instead we will already use one: the ball object.

The ball object is directly integrated in the Aibo URBI Engine and you can use it directly, just like any physical device. It has no `ball.val` variable but it has a `ball.x` and `ball.y` variable which are equal to the coordinate of the ball in the image, expressed between $-1/2$ and $1/2$. When there is a ball visible, `ball.visible` is equal to 1, zero otherwise. It also have a `ball.ratio` variable which give the ratio of pixels of the ball in the image, expressed as a percentage of the total image size. These simple object variables are already enough to do many interesting applications, as we will see below.

The main program

The ball tracking program is given as an example in the Sony SDK (OPEN-R) and does the following: when there is a ball in front of the robot, it will track it by moving the head in the ball direction, otherwise it will scan the surrounding environment by moving the head in circles.

Moving the head in the direction of the ball can be written very simply in URBI with these two lines of code:

```
headPan = headPan + camera.xfov * ball.x &
headTilt = headTilt + camera.yfov * ball.y;
```

The effect is to move at the same time (this is the meaning of the `&` separator) the head motors in pan and tilt directions, by an amount proportional to the x and y position of the ball in the image. The `camera.xfov` and `camera.yfov` coefficients are coming from the camera device that we will discover in the next chapter. They represent the x-angular and y-angular field of view of the Aibo camera, which are used here to convert the $[-1/2;1/2]$ unit segment of `ball.x` and `ball.y` into actual angles in degrees.

To actually track the ball, and not simply move once in its direction, we will use a `whenever` command:

```
whenever (ball.visible) {
  headPan = headPan + camera.xfov * ball.x &
  headTilt = headTilt + camera.yfov * ball.y;
};
```

This program is only three lines long and does the ball tracking behavior expected. However, on the Aibo, it might be too reactive and lead to small oscillations of the head around the ball position. To avoid this, a simple technique from robotic control is to use an attenuation coefficient, `ball.a`, to limit the reactivity of the system. For example:

```
ball.a = 0.8;
```

```
whenever (ball.visible) {
  headPan = headPan + ball.a * camera.xfov * ball.x &
  headTilt = headTilt+ ball.a * camera.yfov * ball.y;
};
```

The next step is to switch from this behavior to the scanning behavior when the ball is not visible. The scanning behavior can be expressed with a simple sinusoidal movement on both headPan and headTilt. We use in the following example the 'n' variable extension¹ which indicates that we are working with the normalized value of the variable, between 0 and 1, calculated from the known rangemin and rangemax properties. It is very convenient to avoid checking the actual range of a device and use it in a more general way:

```
period = 10s;
headPan'n = 0.5 sin:period ampli:0.5 &
headTilt'n = 0.5 cos:period ampli:0.5,
```

The cos modifier is identical to sin with a phase shift of $\pi/2$. Note how the central value of 0.5 with the amplitude of 0.5 allows to cover the full range of the device: [0..1]

The above command does the circular movement required but when this behavior is started, the first position in the circle will be reached abruptly from wherever the head was before the command starts. To avoid this, we can precede the command with a smooth transition in one second towards the initial position in the circle, which is headPan'n = 0.5 and headTilt'n = 1:

```
headPan'n = 0.5 smooth:1s &
headTilt'n = 1 smooth:1s;
```

The smooth modifier is similar to time but with a smooth S-shaped movement, instead of a linear movement.

Now, we can connect everything into one single behavior, using the 'at' event catcher as a glue. To avoid switching from the circular sweeping to the ball tracking too often, we also add a soft test, and we use the loadwav function to preload two wav files that we assign to the speaker device (described later) to play a sound when the ball is found or lost:

```
// Parameters initialization
ball.a = 0.9;
period = 10s;
found = loadwav("found.wav");
lost = loadwav("lost.wav");

// Main behavior
whenever (ball.visible ~ 100ms) {
  headPan = headPan + ball.a * camera.xfov * ball.x &
  headTilt = headTilt+ ball.a * camera.yfov * ball.y;
};

at (!ball.visible ~ 100ms)
search: {
  { headPan'n = 0.5 smooth:1s &
    headTilt'n = 1 smooth:1s } |
  { headPan'n = 0.5 sin:period ampli:0.5 &
    headTilt'n = 0.5 cos:period ampli:0.5 }
```

¹Other extensions are available in URBI. Extensions are a powerful way to modulate the evaluation of a variable. Check the URBI Language Specification for more details

```

};

at (ball.visible) stop search;

// Sound behavior
at (ball.visible ~ 100ms) speaker = found
onleave speaker = lost;

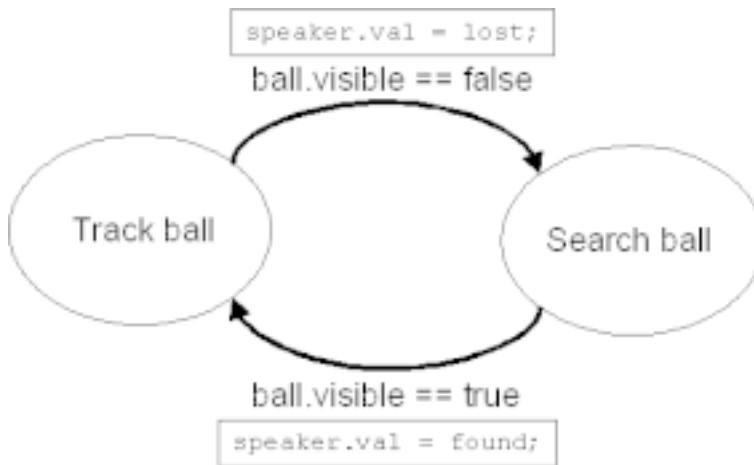
```

You can also use the onleave construct to group the two at (ball.visible) commands, but you must use the at& command in that case, to put the search command in background (because it is a never-ending command and at would never get the hand again otherwise).

Programming as a behavior graph

The above program works fine and is easy to understand and maintain. However, it is common in robotics to design programs in terms of behaviors expressed as finite state machines, which are graphs of states connected together with transitions. Fig. 6.1 illustrates the behavior graph of the ball tracking program, which is a very simple example of a two states behavior.

Figure 6.1. The ball tracking behavior graph



The ellipses represent states (in which the robot is doing some basic action/perception loop) and the arrows are the transitions, expressed over conditions. The squares attached to the transition specify some action to trigger when the transition occurs.

The best way to program this kind of behavior graph in URBI is to use a conjunction of functions with at and stop commands to link everything. First, let's define the two functions related to the two states of the ball tracking program:

```

// Tracking state
function tracking() {
  whenever (ball.visible) {
    headPan = headPan + ball.a * camera.xfov * ball.x
    &
    headTilt = headTilt+ ball.a * camera.yfov * ball.y;
  }
};

// Searching state
function searching() {
  period = 10s;
  {

```

```
    headPan'n = 0.5 smooth:1s &
    headTilt'n = 1 smooth:1s
  } |
  {
    headPan'n = 0.5 sin:period ampli:0.5 &
    headTilt'n = 0.5 cos:period ampli:0.5
  }
};
```

Now, we can simply "glue" the states together by stating the transitions as two at commands with stop commands to terminate the previous state:

```
// Transitions
at (ball.visible ~ 100ms) {
  stop search;
  speaker = found;
  track: tracking();
};

at (!ball.visible ~ 100ms) {
  stop track;
  speaker = lost;
  search: searching();
};
```

The advantage of rewriting the ball tracking program in terms of finite state machine behavior may not appear very clear at this stage, because the program is very simple. However, with more complex behaviors including tens of different states, each with several transitions, this is the best and safest way to program. It makes the code modular, clear and easy to maintain.

Finite state machines are a good way to describe behaviors for robots. They are certainly not perfect, but it's currently the most used technique in robotics. URBI as a programming language is also capable to describe subsumption-based architectures, hierarchical architectures or reactive architectures and many other behavior definition paradigm.

Controlling the execution of the behavior

The possibility to freeze, stop or block commands in URBI is a very powerful tool to control the execution of a behavior. For example, if the transitions which are expressed with a at command are prefixed by a tag, like this:

```
track_transition:
  at (ball.visible ~ 100ms) {
    stop search;
    speaker = found;
    track: tracking();
  };

search_transiton:
  at (!ball.visible ~ 100ms) {
    stop track;
    speaker = lost;
    search: searching();
  };
```

It becomes very easy to temporarily suspend or reactivate a transition by commands like:

```
freeze track_transition;  
...  
unfreeze track_transition;
```

Also, it is possible to block the execution of a state, but still accept transitions to this state (waiting silently for another transition to make the robot move to another state):

```
block search;  
...  
unblock search;
```

Using freeze, block and stop, it is simple to modify behaviors or reassign priorities online during the execution of a program, which is a very useful feature for robotics. The possibilities are numerous, since the behavior tuning can be controlled by events or other programs running in parallel, or even by a controlling remote program or a user over a telnet session.

Chapter 7. Images and sounds

Until now, we have only used numerical variables, like `headPan.val`. This, of course, is not sufficient to transmit images or sounds. Some devices, like for example camera, micro or speaker in Aibo, are binary devices. In that case, the `device.val` variable is not a numerical value but a binary value.

Reading binary values

You might have already tried to evaluate one of those binary variables:

```
camera;  
[139464:notag] BIN 5347 jpeg 208 160  
.....5347 bytes.....  
  
micro;  
[139464:notag] BIN 2048 wav 2 16000 16 1  
.....2048 bytes.....
```

URBI simply prefixes the binary data with a header starting with the keyword `BIN`, followed by the size (in octets) and a keyword indicating the type of the data. Optional parameters, like the size of the image or the sampling rate and stereo/mono status of a sound might follow. Then, after a carriage return, the actual binary data is returned (displayed above as a series of dots: `....`), which might confuse a telnet client but not a software client or URBI Remote¹.

What we call a "software client" is a client or a component written in a language like C++ or Java, as described in detail in the chapter "The liburbi in C++". This is the normal way of handling binary data when you want to do complex signal processing with URBI.

Setting binary values

As you might expect, setting a binary value into a speaker device for example is not more complex than reading it. To play a sound on Aibo, you could send to the server a command like this:

```
speaker = bin 54112 wav 2 16000 16;  
.....54112 bytes.....
```

It is important that the header ends with a semicolon (and nothing else). The binary content starts immediately after the semicolon, so you don't have to add an extra carriage return.

Of course, as we already said it, this kind of binary assignment will obviously not be done from a telnet or URBI Remote client, since you probably want that a program sends the binary content, and you cannot type it yourself in the terminal! (However, we will see in the next section how you can simply play a recorded sound from a telnet client if you need to).

This simple example illustrates a binary assignment and a binary reading in URBI from a telnet client, however it is a "toy" example:

```
mybin = bin 3;ABC  
mybin;  
[146711:notag] BIN 3  
ABC
```

¹URBI Remote understands URBI headers and displays images or plays sounds according to the type

Note that you can pass any parameters after the size of the binary data and they will be stored together with the binary content, inside the header:

```
mybin = bin 3 hello world 33;ABC
mybin;
[146711:notag] BIN 3 hello world 33
ABC
```

Do not confuse binary data and string data. The above example is different from:

```
mystring = "ABC";
mystring;
[148991:notag] "ABC"
```

Associated attributes

Usually, with a binary device object you have a set of associated attributes available. A typical example is the camera device which provides the following attributes on Aibo:

- camera.shutter : the camera shutter speed: 1=SLOW (default), 2=MID, 3=FAST
- camera.gain : the camera gain: 1=LOW, 2=MID, 3=HIGH (default)
- camera.wb : the camera white balance: 1=INDOOR (default), 2=OUTDOOR, 3=FLUO
- camera.format : the camera image format: 0=YCbCr 1=jpeg (default)
- camera.jpegfactor : the jpeg compression factor (0 to 100). Default=80
- camera.resolution : the image resolution: 0:208x160 (default) 1:104x80 2:52x40
- camera.reconstruct : reconstruction of the high resolution image(slow): 0:no (default) 1:yes
- camera.width : image width
- camera.height : image height
- camera.xfov : camera x Field Of View (degrees)
- camera.yfov : camera y Field Of View (degrees)

In the case of the speaker device, in charge of the speaker producing sound in the Aibo, you have:

- speaker.playing : equal 1 when there is a sound playing, 0 otherwise
- speaker.remain : number of milliseconds of sound to play, 0 when the buffer is empty.

With the speaker object, there is also a method that can be used to play a sound directly from a file stored on the memorystick:

```
speaker.play("mysound.wav");
```

Alternatively, to avoid having a disk access which might be slow, you can decide to store the content of the "mysound.wav" file in a binary variable kept in memory for frequent use, and then do a simple assignment. For this, use the loadwav function:

```
mybin = loadwav("mysound.wav");
```

```
speaker = mybin;
```

Binary operation examples

There is a possibility in URBI to add binaries, which is typically used for sound concatenation. For example, consider the following program:

```
sound = bin 0;  
timeout(10s) loop sound = sound + micro;  
speaker = sound;
```

This code will record 10 seconds of sound from the micro device and store it in the sound variable, and then will play it back by assigning sound to the speaker device. It shows how simple it can be to manipulate binary buffers with URBI for simple tasks like concatenation.

Chapter 8. The liburbi in C++

What is liburbi?

Using URBI with a telnet client is too limited. You need to be able to send commands and receive messages using a programming language of your choice, or in a more general way, you need to be able to interface URBI with other languages.

That's why we call URBI an "Interface Language": it's more than a simple protocol because it's a full featured script language acting as a protocol. In most applications where you have computer vision or sound processing, you use URBI together with C++ or another fast language to do the algorithmic part. URBI is here to run the architecture of your behaviors, your action/perception loops and other high level elements, using the output of the fast C++/Java/Matlab code as inputs for its decisions.

What is liburbi? You could program a TCP/IP layer for C++ or for your favorite language but this is trivial and should be done once and for all. This is why we made liburbi. What you want to be able to do are things like:

- Open a connection to your robot from within your favorite language (like C++)
- Send a command to your robot from within that language
- Ask for a variable value and receive it
- Listen to incoming messages from your robot and react to them appropriately

Actually, the last point is the most important and, even if it might differ from the way you may be used to write programs, it is essential to adapt to this way of thinking (called "asynchronous programming") because it is best suited for robotics. Robots are fundamentally asynchronous systems. You usually wait for messages from your robot and react to them (it's also called "event-driven programming"). That's what a robot does most of the time: react to events¹.

This chapter is a brief introduction to liburbi. You should read the official liburbi documentation on <http://www.gostai.com/docs.php> if you want a comprehensive description. If you program with C++, we suggest to use the UObject architecture described later in this tutorial, liburbi being only a complement to the new and more powerful UObject technology.

Components and liburbi

Extending URBI with code written in C++, Java or Matlab that will be made available to your URBI scripts can be done in two different ways. The first way it to use one of the liburbi flavor for your preferred language (C++/Java/Matlab) and build a software client. That is what we are going to describe in this chapter.

The second option, which is more powerful and described in the chapter "Create components: the UObject architecture" is to create a UObject Component which refers to an object in C++, accessible like any other URBI object from your URBI programs, sharing methods and object attributes. It is the most portable and flexible way of adding functionalities to URBI by mirroring objects, but let's start with the basic liburbi. One of the interest of liburbi is also that it is available for with many more language (the object binding is not always possible otherwise and is currently limited to C++), and in any case, knowing the liburbi approach is a good idea, since it might be more suited to your need in certain cases and it gives a good introduction to asynchronous programming.

¹Traditionally in AI, the way the robot reacts might be modified by higher level cognitive activities (hierarchical architecture) or by priorities (subsumption architecture) or by a complex combination of deliberative and reactive processes (hybrid architecture)

There is currently a C++, Matlab, Java and Python version of liburbi if you want to control your robot using C++, Matlab, Java or Python, and a liburbi-OPENR version if you want to recompile a liburbi-C++ based program to let it run on the Aibo, as an OPENR object (in that case, your robot will remain completely autonomous). However, we strongly suggest to abandon the OPENR version and switch to the UObject architecture to embed components in the Aibo. This only interest of this OPENR version is that it allows to have a sort of implicit non-blocking thread, which is otherwise impossible with the Aperios operating system from Sony.

We will not describe all the liburbi implementations here but only the C++ version, which gives the general ideas. Other versions are similar and have a specific documentation. We assume in the following that you have a basic understanding of C++. If not, please refer to a simple C++ tutorial, since the notion developed here will remain basic.

Getting started

To start with, you need to be able to compile a liburbi-based program. There are several ways to do so, depending on the fact that you are on Linux or Windows, with Borland or Microsoft compiler, etc. In general, you are simply supposed to include liburbi.h and link your code with the "-lurbi" parameters (gcc) or similar syntax specific to other compilers. See the appropriate documentation for more details.

On the code side, the first thing you need to do is to create a client connected to your robot "myrobot.mydomain.com". For this purpose, you have a UClient class in liburbi-C++:

```
UClient* client = new UClient("myrobot.mydomain.com");
```

If you have an IP address, you can use it instead of the server name.

Alternatively to explicitly calling the UClient class, you might want to use a function from the urbi namespace instead:

```
UClient* client = urbi::connect("myrobot.mydomain.com");
```

Of course, you can create as many clients as you like with these methods.

Sending commands

The UClient object has a send method which works like printf:

```
client->send("motor on;");
for (float val=0; val<=1; val+=0.05)
    client->send("neck'n = %f;wait (%d);", val, 50);
```

You can also use your client object as a stream if you prefer a more C++ like approach:

```
client << "headPan = " << 12 << ";;"
```

There is also a very convenient way of sending blocks of URBI code from a C++ program, using the URBI(...) macro:

```
URBI((
    headPan = 12,
    echo "hello" | speaker.play("test.wav") & leds = 1
));
```

The text between the double parenthesis will be sent verbatim to the first client created by your program, by default. This can be set with a call to `urbi::connect(...)`. The first described approach, using the `send` method, is more appropriate in general and the URBI macro should only be used to send initialization scripts in a convenient way at the beginning of your program, or for fast prototyping.

Remember that you can always give your robot a fresh start (a virtual reboot) by sending the `reset` command. This will avoid the multi definition of functions or restarting several occurrences of an at command each time you rerun your liburbi-based client. So many liburbi main programs will start with `client->send("reset;");`

Sending binary data and sounds

To send binary data, you will use the `sendBin` method, instead of `send`:

```
client->sendBin(soundData, soundDataSize,
               "speaker = BIN %d raw 2 16000 16 1;",
               soundDataSize);
```

The first two parameters are the binary data itself and the size. Then, the header, with optional parameters using a `printf` like syntax.

To send a sound, there is specialized method called `sendSound`, which is more convenient and also more efficient:

```
client->sendSound(sound, "endsound");
```

The first parameter is a `USound` structure, describing the sound to send. The second is an optional tag that will be used by the server to issue a "stop" system message when the sound has finished playing.

The function `convert` described in the documentation can be used to convert between various sound formats.

With `sendSound`, there is no limit to the size of the sound buffer, since it will be automatically cut into small chunks by the library. Since the data is copied by liburbi, the `USound` parameter and its associated data can be safely freed as soon as the function returns.

Receiving messages

URBI tags are going to prove very useful for receiving incoming messages from the server: each command has an associated tag (notag by default), and this tag is transmitted in any message originating from this command. The `UClient` class handles the reception of those messages in an independent thread created by the constructor, parses them and fills a `UMessage` structure. Then, callback functions with the associated tag can be registered with the method `setCallback`: each time a message with this tag is sent by the server, the callback function will be called with the `UMessage` structure as a parameter.

```
typedef UCallbackAction (*UCallback) (const UMessage &msg);
```

```
UCallbackID setCallback (UCallback cb, const char *tag)
```

The first parameter `cb` is a pointer to the function to call. The callback function must return `URBI_CONTINUE`, or `URBI_REMOVE`, in which case the function will be unregistered.

The best way to learn about how callbacks can be used with the liburbi is to look at some example, like the one described in the liburbi documentation page at:

<http://www.gostai.com/doc/en/liburbi-1.0/>

Data types

The data type used by the liburbi are described below:

UMessage

The UMessage structure is capable of storing the informations contained in any kind of URBI message by using a "type" field and an UValue (union of type-dependant structures). These two structures are defined as follows:

```
class UMessage
{
public:
    /// Connection from which originated the message.
    UAbstractClient &client;
    /// Server-side timestamp.
    int timestamp;
    /// Associated tag.
    std::string tag;

    UMessageType type;

    urbi::UValue *value;
    std::string message;
    /// Raw message without the binary data.
    std::string rawMessage;
};
```

UValue

```
class UValue
{
public:
    UDataType type;
    ufloat val; // value if of type DATA_DOUBLE
    union
    {
        std::string *stringValue; // value if of type DATA_STRING
        UBinary *binary; // value if of type DATA_BINARY
        UList *list; // value if of type DATA_LIST
        UObjectStruct *object; // value if of type DATA_OBJ
    };
};
```

The type field UMessageType can be MESSAGE_SYSTEM, MESSAGE_ERROR or MESSAGE_DATA. If the type is MESSAGE_DATA, the message contains an UValue. The UValue itself contains an UDataType which can take the values: DATA_DOUBLE, DATA_STRING, DATA_BINARY, DATA_LIST, DATA_OBJECT, DATA_VOID. Depending of this field, the corresponding value in the union will be set. If the UValue is of the binary type, it contains an UBinary structure defined hereafter. The UBinaryType in the UBinary structure will give additional informations on the type of data (BINARY_NONE, BINARY_UNKNOWN, BINARY_IMAGE, BINARY_SOUND), and the appropriate sound or image structure will be filled.

UBinary

```
class UBinary
{
public:
  UBinaryType  type;
  union
  {
    struct
    {
      void *data; /// binary data
      int  size;
    } common;
    UImage image;
    USound sound;
  };
};
```

USound

```
class USound {
public:
  char          *data;          // pointer to sound data
  int           size;          // total size in byte
  int           channels;      // number of audio channels
  int           rate;          // rate in Hertz
  int           sampleSize;    // sample size in bit
  USoundFormat  soundFormat;   // format of the sound data
                                   // (SOUND_RAW, SOUND_WAV, SOUND_MP3...)
  USoundSampleFormat  sampleFormat; // sample format
};
```

UImage

```
class UImage {
public:
  char          *data;          // pointer to image data
  int           size;          // image size in byte
  int           width, height; // size of the image
  UImageFormat  imageFormat;   // IMAGE_RGB, IMAGE_YCbCr, IMAGE_JPEG
};
```

Synchronous operations

The derived class USyncClient implements methods to synchronously get the result of URBI commands. You must be aware that these functions are less efficient, and that they will not work in the OPEN-R version of the liburbi, for instance. As a general programming rule with robots, synchronous programming should be avoided.

Synchronous read of a device value

To get the value of a device object (with a val attribute), you can use the method syncGetDevice. The first parameter is the name of the device (for instance, "neck"), the second is a double that is filled with the received value:

```
double neckVal;  
syncClient->syncGetDevice("neck", neckVal);
```

Getting an image synchronously

You can use the method `syncGetImage` to synchronously get an image. The method will send the appropriate command, and wait for the result, thus blocking your thread until the image is received.

```
client->send("camera.resolution = 0;camera.gain = 2;");  
int width, height;  
client->syncGetImage("camera", myBuffer, myBufferSize,  
                    IMAGE_RGB, URBI_TRANSMIT_JPEG, width, height);
```

The first parameter is the name of the camera device. The second is the buffer which will be filled with the image data. The third must be an integer variable equal to the size of the buffer. The function will set this variable to the size of the data. If the buffer is too small, data will be truncated .

The fourth parameter is the format in which you want to receive the image data. Possible values are `IMAGE_RGB` for a raw RGB 24 bit per pixel image, `IMAGE_PPM` for a PPM file, `IMAGE_YCbCr` for raw YCbCr data, and `IMAGE_JPEG` for a jpeg-compressed file.

The fifth parameter can be either `URBI_TRANSMIT_JPEG` or `URBI_TRANSMIT_YCbCr` and specifies how the image will be transmitted between the robot and the client. Transmitting JPEG images increases the frame rate and should be used for better performances.

Finally the width and height parameters are filled with the width and height of the image on return.

Getting sound synchronously

The method `syncGetSound` can be used to get a sound sample of any length from the server.

```
client->syncGetSound("micro", duration, sound);
```

The first parameter is the name of the device from which to request sound, the second is the duration requested, in milliseconds. Sound is a `USound` structure) that will be filled with the recorded sound on output.

Conversion functions

We also have included a few functions to convert between different image and sound formats. The usage of the image conversion functions is pretty straightforward:

```
int convertRGBtoYCrCb(const byte* source, int sourcelen, byte* dest);  
int convertYCrCbtoRGB(const byte* source, int sourcelen, byte* dest);  
int convertJPEGtoYCrCb(const byte* source, int sourcelen, byte* dest, int &size);  
int convertJPEGtoRGB(const byte* source, int sourcelen, byte* dest, int &size);
```

The size parameter must be set to the size of the destination buffer. On return it will be set to the size of the output data.

To convert between different sound formats, the function `convert` can be used. It takes two `USound` structures as its parameters. The two audio formats currently supported are `SOUND_RAW` and `SOUND_WAV`, but support for compressed sound formats such as Ogg Vorbis and MP3 is planned. If any field is set to zero in the destination structure, the corresponding value from the source sound will be used.

The "urbiimage" example

URBIimage is a simple program written in C++ with the liburbi-C++ to get and display images from an URBI server. URBIimage does two things: it sets a callback on a tag named uimg and then receives the images in this callback and send them to a display object Monitor. Let's have a look at the general code and the main function. First, the callback interface:

```
Monitor *mon;

/* Our callback function */
UCallbackAction showImage(const UMessage &msg)
{
    ...
}
```

Then, the main function:

```
int main(int argc, char *argv[])
{
    mon = NULL;
    client = new UClient(argv[2]);
    if (client->error() != 0)
        exit(0);

    client->setCallback(showImage, "uimg");

    // Some image initialization
    client->send("camera.resolution = 0;");
    client->send("camera.jpegfactor = 80;");

    // Start the loop
    client->send("loop uimg: camera,");
    urbi::execute();
}
```

The code to handle the image is stored in showImage:

```
UCallbackAction showImage(const UMessage &msg)
{
    if (msg.type != MESSAGE_DATA || ((UImage)msg).imageFormat == IMAGE_UNKNOWN)
        return URBI_CONTINUE;

    UImage img = (UImage)msg;
    unsigned char buffer[500000];
    int sz = 500000;
    static int tme = 0;

    if (!mon)
        mon = new Monitor(msg.image.width, msg.image.height);

    convertJPEGtoRGB((const byte *) img.data,
                    img.size, (byte *) buffer, sz);

    mon->setImage((bits8 *) buffer, sz);
    return URBI_CONTINUE;
}
```

```
}
```

It first tests for the msg type, and returns without doing anything if this is not the type expected (for example, if the callback is waken up by an error message).

Then, the conversion function `convertJPEGtoRGB` is used to transform the image buffer in something readable for the Monitor object, which then receives the image.

Finally, `URBI_CONTINUE` is returned to carry on receiving future callbacks.

This little program illustrates very well how a liburbi-based URBI program is built: set callbacks, send URBI scripts, receive callbacks in specified functions. You might have a look at the GPL source code of URBI Lab which is built with liburbi-C++ and shows a more advanced use of this methodology.

Chapter 9. Create components: the UObject architecture

The UObject architecture is the most advanced way to extend URBI and integrates powerful components in the language. It's currently limited to C++ but should generalize to other languages in the future. The idea is to take a C++ class and, after a few small modifications, to be able to plug this class in the URBI language so that one can access its methods and attributes as if they were pure URBI objects. A few words about terminology: the UObject architecture enables to add a component to the language, and this component will be seen as an object.

There are actually two ways of integrating your C++ class inside URBI:

- Mode plugin: You can plug the object directly in URBI (link it to the URBI Engine) and it will be part of the binary code of the URBI Engine.
- Mode remote: You can run it as an autonomous remote process that will connect itself to your URBI Engine and transparently add the object to the language, just like in the plugin mode, but remotely.

In both cases, we provide the necessary tools to make the link (described below). The good news is that the C++ source code of your object is exactly the same in both cases, and the way you use it inside URBI is also transparent. So, you can decide to plug/remote-run a component at will (hopefully in the future, you will be able to relocate the object at runtime, but not for now).

We will now see how to turn your C++ class into an UObject class, and we will see then how to connect the methods and attributes of the C++ class to URBI.

UObject

The basics

Let's create a colormap object, composed of colormap.cpp and colormap.hh. The colormap.hh should start like this:

```
#include <urbi/uobject.hh>
using namespace urbi;

class colormap : public UObject
{
public:
    colormap(std::string);
    ...
};
```

Whatever constructor you previously had should be renamed `init`. The default constructor `myclassname(std::string)` which is appropriate for UObjects must be used instead. For example, you might define the constructor `init` which takes a RGB point as a color definition like this:

```
public:
    colormap(std::string);
    int init (int r, int g, int b);
    ...
```

For the moment, that's all what you need on the class definition side. Let's have a look at the main code in colormap.cpp:

```
#include "colormap.hh"

UStart(colormap);

colormap::colormap(std::string s) : UObject(s)
{
    UBindFunction(colormap, init);
}

int colormap::init(int r, int g, int b)
{
    return 0;
}

...
```

Two new things here: you have to invoke the "magic" line `UStart(myobject)` in order to let the system know about it. Then, you must make sure that the default constructor calls the `UObject` constructor and passes the string, and also bind the `init` function to make it visible and export it in URBI. This is required if you want the `init` constructor to be called by URBI upon a new object creation. The `init` method should return `0` upon success, anything else in case of failure (you can also return `void` which is considered as a success).

There is nothing else to know, at this stage you already have a exportable object called 'colormap' with a method 'init'. Now, you can compile it and get a binary code ready to link.

Let's assume that you have linked this code to the URBI Engine, to make it a component in plugin mode (we will see how later). Now, how to use this new colormap object? Well, not much has to be done: it's already there. Remember that in URBI there is no difference between a class and an instance (prototype-based language), so defining colormap is enough to have a functional colormap object. You can try to evaluate it to see this:

```
colormap;
[139464:notag] OBJ [load:1.000000]
```

NB: By default, there is an exported load attribute in `UObject`, let's ignore it for the moment.

Let's define a subclass of colormap. This action will call the `init` constructor on the C++ side and spawn a new instance of the C++ colormap class, but of course this is all done automatically and you don't have to take care of that:

```
ball = new colormap(123,45,12);
ball;
[139464:notag] OBJ [load:1.000000]
```

You see that the syntax to create a new object in URBI is identical to the C++ syntax. Each time it was possible, we have kept the familiar C/C++ syntax in URBI, because there is no point to waste time learning stuffs we already know (as long as there is no confusion in term of semantics).

Adding attributes

Our colormap object is not much fun so far. To make it more useful, we can start to add attributes to the object and bind them to URBI. To add a `x` variable, we will simply add `UVar x`; inside the class definition:

```
#include <urbi/uobject.hh>
```

```
using namespace urbi;  
  
class colormap : public UObject  
{  
public:  
    colormap(std::string);  
  
    UVar x; // definition of the exported variable  
    ...  
};
```

and then add the binding code in the init method:

```
int colormap::init(int r, int g, int b)  
{  
    UBindVar(colormap, x);  
    ...  
}
```

Actually, you can put your binding code (UBindVar) anywhere you want, in particular it can be in the C++ object constructor or in the object init method. If you put it in the C++ constructor, it will make the variable available to the base instance (the one that is there at start and that you don't have to 'new'), or if you put it in the 'init' method, only 'newed' objects will have it. This is useful if the base instance is useless because you need to derive it to specify it. In that case you put all your bindings in the 'init' method only and the base instance is just a sort of ghost instance. Note that UObject::derived is a boolean that tells you if your class has been derived with a 'new' or if it is the base class.

You can check, now the colormap.x and ball.x will be there.

To assign a value to x from within your C++ class, simply use it as a normal variable, UObject will do the rest for you:

```
x = 42;  
    or  
x= "hello";
```

The = operator in C++ has been redefined for UVar, so that you don't have to worry and you can assign values to x as you would do it from within URBI.

Now, how to read the variable? We've tried to keep things simple again: you can simply use a C-style casting to get a value in the appropriate C++ type. For the moment, there is not exception raised if an error occurs, so be careful to what you are doing:

```
x = 42;  
printf("Value of x: %d n", (int)x);
```

x is called a "hook" to the URBI colormap.x variable. Actually, you can define hooks on any variable you like by defining your own UVar instance wherever you like (it will be automatically binded, no need to use UBindVar, the UVar constructor does it). Here are a few examples:

```
UVar("camera.val");  
UVar("camera", "val");  
UVar* myvar = new UVar("headPan", "val");
```

The reason why you have to call UBindVar for a UVar defined in the body of your class is that this UVar is a non-dynamically allocated UVar called with the default UVar() constructor. Such a UVar doesn't know its name at this stage and the UBindVar macro simply tells it who it is. You don't need this stage with a direct call to the UVar(std::string) constructor who takes the name as its parameter.

Of course, your C++ object can contain many attributes that will not be exported to URBI and will remain "private" to the C++ class. To make an attribute available to URBI, you need to define it as a UVar or to "UBindVar" one that is part of your object definition.

One important thing that one wants to do with attributes is to monitor them for changes or accesses. This is done by assigning a callback function to the variable, specifying whether you want to be called back on changes or on accesses:

```
UNotifyChange(x, &colormap::mycallback);  
UNotifyAccess(UVar("doo.daa", &colormap::myothercallback);  
UNotifyChange("another.variable", &colormap::anothercallback);
```

Notify on change means that the callback will be called each time the variable is modified on the URBI side (for variables attached to sensors, it means "each time the sensor value is updated"). Notify on access means that the callback will be called each time someone evaluates the variable on the URBI side, so that you have a chance to update its value before the evaluation. In that case, you are advised to put a time-based caching mechanism in your callback if the variable is called frequently inside expressions.

You will typically put those "Notify" lines in the init function or in the constructor of your object, the choice of one over the other being dictated by the same rationale than with UBindVar. Notice that you must pass a pointer to a function, which must be a method of your object. You have only two types of prototypes available for these callbacks:

```
UReturn mycallback();  
UReturn mycallback(UVar&);
```

The first one is the simplest and obvious one: the function is called when the condition is met. The second one does the same thing but passes the UVar as a reference parameter so that you can use the same callback with several variables and get the one that is related to the current call.

Binding functions and events

Just like you did with attributes, you can easily bind a function to the mirrored URBI object. There is not much to do there, simply use the following construct:

```
int colormap::init(int r, int g, int b)  
{  
    UBindFunction(colormap, dostuff);  
    ...  
}  
  
std::string colormap::dostuff(int, float)  
{  
    ...  
}
```

This will make the method dostuff visible to the outside. You don't need to worry about parameters, they will be recognized and exported for you. For the moment, you cannot overload a function with this mechanism (and in particular, you cannot overload the init constructor).

Similarly, you can bind an event to a method of your object, so that this method will be called each time the corresponding event is emitted on the URBI side, and you will get the parameters on the way. Simply do:

```
UBindEvent(colormap, reacttothis);
```

You can also ask to be notified when the event terminates (as you know, events can last during a certain amount of time in URBI). For example, if you want to be notified by calling the `endthis` method of your object, simply use:

```
UBindEvent(colormap, reacttothis);
UBindEventEnd(colormap, reacttothis, endthis);
```

`endthis` must have a simple prototype like this one:

```
void colormap::endthis();
```

Timers

You can easily set timers to be called back at regular time intervals. The syntax is:

```
USetTimer(time_in_ms, &myobject::mycallback);
```

With `mycallback` being a method of your object with the following prototype:

```
UReturn myobject::mycallback();
```

You cannot use a callback function coming from outside of your object.

Advanced types for binaries

For integers, floats and strings the assignment and reading-by-casting of `UVar` is straitforward. For binary data, like images and sounds, you will need two appropriate types: `UImage` and `USound`. Here is a copy of their definition from `uobject.hh`

```
///Class encapsulating an image.
class UImage {
public:
    char                *data;           ///< pointer to image data
    int                 size;           ///< image size in byte
    int                 width, height;  ///< size of the image
    UImageFormat        imageFormat;
};

///Class encapsulating sound informations.
class USound {
public:
    char                *data;           ///< pointer to sound data
    int                 size;           ///< total size in byte
    int                 channels;       ///< number of audio channels
    int                 rate;          ///< rate in Hertz
    int                 sampleSize;    ///< sample size in bit
    USoundFormat        soundFormat;   ///< format of the sound data
    USoundSampleFormat  sampleFormat;  ///< sample format
}
```

You recognize them, they are the types used in `liburbi`, no surprise. If your `UVar` is an image, like "camera.raw", you can simply cast it to a `UImage` and you will retrieve the relevant information in the appropriate attributes, in particular the binary content will be in `data` and the size in `size`. Same thing for a sound.

Be careful if you use `camera.val`: it might be a jpeg-compressed binary and you should convert it with one of those functions, as described in section #x1-690007.9:

```
int convertRGBtoYCrCb(const byte* source, int sourcelen, byte* dest);
int convertYCrCbtoRGB(const byte* source, int sourcelen, byte* dest);
int convertJPEGtoYCrCb(const byte* source, int sourcelen, byte* dest, int &size);
int convertJPEGtoRGB(const byte* source, int sourcelen, byte* dest, int &size);
```

If you want to assign a sound to, let's say `speaker.val`, simply fill up a `USound` variable and assign it to the appropriate `UVar`, the `=` operator has been redefined to handle this. So far, we handle only wav format.

Writing anything to an `uvar` will copy the memory. `USound` and `UImage` do no memory management at all, so assigning an `USound` to an other just copies the pointer. If you want memory to be automatically managed, you can use `UBinary`, which deletes its buffer when its destructor is called. As a consequence, returning a `USound` in a bound function is problematic. It would be better to wrap the `USound` in a `UBinary` and return the `UBinary`.

The "load" attribute

We have already mentioned the load attribute that is defined as a `UVar` and bound by default in `UObject`. This attribute can be used to test in your C++ code whether the object is activated or not on the URBI side. In URBI, a call to "myobject on;" will put load to 1 and a call to "myobject off;" will put it to 0. So, you can easily test in your various functions if you have to do the computation or not, based on the value of load.

This is extremely useful if you want to be able to activate/disactivate some CPU hungry calculation that would otherwise run for nothing in the background. For example, you can turn the ball detection off in Aibo with:

```
ball off;
```

Note that this is a broadcastable construct: if you on/off a group, it will recursively propagate to every member of the group. That's exactly what happens behind the scene with a command like `motors on`;

NB: You also have "myobject switch;" to alternate between on and off.

The "remote" attribute

In `UObject` definition, the remote attribute is available to know whether your object is running as a remote component or a plugged one. This can be useful when you want to behave differently in both cases, typically handling the transfer of large amount of data or images with or without compression. The colormap example below make use of the remote attribute.

The colormap example

Here is a real example of a colormap object as it is used in the Aibo, to calculate the average position of a blob of color defined by a subspace of the YCrCb color space. You see how we bind a callback to source, which is usually camera. The actual callback is set to the `.val` or `.raw` attribute of the source object, depending on the status of the object, remote or not. In remote mode, we want to use jpeg compression and work with the resulting image value, whereas in plugged mode, we can use shared memory on the raw buffer to get a better image without artifacts, and avoid compressing/decompressing for nothing.

You also see how we simply assign values to the x and y attributes and other attributes describing the shape of the blob:

First the colormap.hh file (extracts only):

```
#include <urbi/uobject.hh>
using namespace urbi;

class colormap : public UObject
{
public:

    colormap(std::string);
    int init(std::string,int,int,int,int,int,int,int,ufloat);

    UVar    x;
    UVar    y;
    UVar    visible;
    UVar    ratio;
    UVar    threshold;
    UVar    orientation;
    UVar    elongation;
    UVar    ymin, ymax, cbmin, cbmax, crmin, crmax;

    UReturn newImage(UVar&);
};
```

Here, we use ufloat instead of float because ufloat can be adapted to 32bits or 64bits or even no-FPU motherboards and thus it is more suitable for embedded applications.

Now, the main code:

```
#include "colormap.hh"

UStart(colormap);

//! colormap constructor.
colormap::colormap(std::string s) :
    UObject(s)
{
    UBindFunction(colormap,init);
}

//! colormap init function
int
colormap::init(std::string source,
               int _Ymin,
               int _Ymax,
               int _Cbmin,
               int _Cbmax,
               int _Crmin,
               int _Crmax,
               ufloat _threshold)
{
    UBindVar(colormap,x);
    UBindVar(colormap,y);
    UBindVar(colormap,visible);
    UBindVar(colormap,ratio);
    UBindVar(colormap,threshold);
    UBindVar(colormap,orientation);
    UBindVar(colormap,elongation);
    UBindVar(colormap,ymin);
}
```

```
UBindVar(colormap,ymax);
UBindVar(colormap,cbmin);
UBindVar(colormap,cbmax);
UBindVar(colormap,crmin);
UBindVar(colormap,crmax);

if (remote)
    UNotifyChange(source+".val",&colormap::newImage);
else
    UNotifyChange(source+".raw",&colormap::newImage);

// initialization
ymin      = _Ymin;
ymax      = _Ymax;
cbmin     = _Cbmin;
cbmax     = _Cbmax;
crmin     = _Crmin;
crmax     = _Crmax;
threshold = _threshold;
x         = -1;
y         = -1;
visible   = 0;
orientation = 0;
elongation = 0;
ratio     = 0;

return 0;
}

//! colormap image update
UReturn
colormap::newImage(UVar& img)
{
    if ((ufloat)load < 0.5) return(1);

    UImage img1 = (UImage)img; //ptr copy

    if (remote)
        convertYCrCb(img1); // this function is available in UObject 1.0 only

    int w = img1.width;
    int h = img1.height;

    //lets cache things
    int ymax = this->ymax; int ymin = this->ymin;
    int crmin = this->crmin; int crmax = this->crmax;
    int cbmin = this->cbmin; int cbmax = this->cbmax;

    long long x=0,y=0,xx=0,yy=0,xy=0;
    int size = 0;
    for (int i=0;i<w;i++)
        for (int j=0;j<h;j++) {

            unsigned char lum = img1.data[(i+j*w)*3];
            unsigned char cb  = img1.data[(i+j*w)*3+1];
```

```
unsigned char cr = img1.data[(i+j*w)*3+2];

if ( (lum >= ymin) &&
      (lum <= ymax) &&
      (cb >= cbmin) &&
      (cb <= cbmax) &&
      (cr >= crmin) &&
      (cr <= crmax) ) {
    size++;
    x += i;
    y += j;
    xx += i*i;
    yy += j*j;
    xy += i*j;
}
}

this->ratio = ((ufloat)size)/((ufloat)(w*h));
if (size > (int)((ufloat)threshold * (ufloat)(w*h))) {

    this->visible = 1;
    this->x = 0.5 - ((double)x /
                   ((double)size * (double)w));
    this->y = 0.5 - ((double)y /
                   ((double)size * (double)h));

    //orientation: first eighenvector of covariance matrice
    double m00 = (double)xx - (double)(x*x)/((double)(size));
    double m11 = (double)yy - (double)(y*y)/((double)(size));
    double m01 = (double)xy - (double)(x*y)/((double)(size));

    //bigest eighenvalue
    double l = (m00+m11)/2.0 + 0.5*sqrt((m00+m11)*
                                         (m00+m11)-4*(m00*m11-m01*m01));

    //first eighenvector orientation
    double angle = atan2(l-m00, m01);
    this->orientation = angle* 180.0 /M_PI;

    //variance on new axis => elongation
    double angle2 = angle + M_PI/2.0;
    double X = x*cos(angle)+y*sin(angle);
    double Y = x*cos(angle2)+y*sin(angle2);
    double XX = xx*cos(angle)*cos(angle)+yy*sin(angle)*sin(angle)+
                2.0*xy*cos(angle)*sin(angle);
    double YY = xx*cos(angle2)*cos(angle2)+yy*sin(angle2)*sin(angle2)+
                2.0*xy*cos(angle2)*sin(angle2);
    double vX = XX - X*X/(double)size;

    double vY = YY - Y*Y/(double)size;

    this->elongation = sqrt(vX/vY);
```

```
    }  
    else {  
        this->x=-1;  
        this->y=-1;  
        this->visible = 0;  
    }  
  
    return(1);  
}
```

The colormap object is then plugged in the URBI Engine and it is used to create a ball detector in the URBI.INI file:

```
ball = new colormap("camera",0,255,120,190,150,230,0.0015);
```

The practical side: how to use create an UObject?

You'll have to install an appropriate SDK (see hereafter), then to use umake in unix environments or visual sudio™.

We have seen that you can create two different types of UObjects: remote UObjects and plugged in UObjects. At the moment, with an Urbi-SDK, you can create remote UObjects only. To create a new engine with plugged-in UObjects, you must use the urbiengine-SDK which allows you also to create remote UObjects.

How to install a sdk to build/link components for your robot?

To build or use components for a given URBI server, you need to install the engine SDK corresponding to the server. Download it from the URBI website (or from the robot manufacturer's website). the installation procedure depends on the format of the downloaded file and of the OS:

mingw

If you are using a mingw under windows, unzip the package in the root mingwdir. In a mingw console, type:

```
cd /  
unzip DOWNLOADED_SDK.zip
```

Visual sudio/Visual C++ express edition

If you are using visual c++ express or visual studio under windows, unzip the package in the any dir you want using your favorite zip tool. You will find inside an "include" dir and a "lib" dir. Provide this directories in your visual project respectively in the include dir list and in the link path list.

mac, zip file

Usually x86 and powerPC versions are available. In a console, type:

```
cd /  
unzip DOWNLOADED_SDK.zip
```

rpm

If you are using a rpm based linux distribution (redhat, mandrake, fedora...), you can run as root in the directory where is your downloaded package:

```
rpm -ivh downloaded_SDK.rpm
```

You can also use your favorite graphical or command line package installer.

deb

If you are using a deb based linux distribution (debian, ubuntu...) you can run as root in the directory where is your downloaded package:

```
dpkg --install downloaded_SDK.rpm
```

with ubuntu there is no root account, use:

```
sudo dpkg --install downloaded_SDK.rpm
```

You can also use your favorite graphical or command line package installer.

tarball

If you are using a deb or rpm based linux distribution, you can either use the tar binary package. As root:

```
cd /  
tar -xvzf downloaded_SDK.tgz
```

or

```
cd /  
tar -xvjf downloaded_SDK.tar.bz2
```

sources

If you want to use (if available) a source package, use the standard commands:

```
tar -xvzf downloaded_SDK.tgz  
cd new_dir  
./configure;make  
sudo make install
```

Some robots requires build chains. For exemple, with aibo, you will need to have an installed OPENR-SDK. There is a good tutorial on how to install it there: <http://aibostuff.iofreak.com/wiki.php?n=OpenR.UbuntuInstall>

How to use umake to create engines and components?

Plugins and remote components are built the same way, using umake.

Basic usage

To compile all source files in the current directory (and its subdirectories) and link them with the Remote SDK, simply type **umake**. To compile all source files in the current directory and produce a library, type **umake-lib**. To compile all source files in the current directory and link with the Engine SDK, type **umake-engine**.

```
$ ls
```

```
foo.cc foo.hh

$ umake

/usr/local/gostai/core/linux/libtool
--tag=CXX --mode=compile g++ -O2 -pthread
-I/usr/local/gostai/core/include -c foo.cc -o foo.lo g++ -O2 -pthread
-I/usr/local/gostai/core/include -c foo.cc -o foo.o
/usr/local/gostai/core/linux/libtool --mode=link --tag=CXX g++ -O2
-pthread -L/usr/local/gostai/core/linux/remote -o
urbiengine-linux-remote ./foo.o
/usr/local/gostai/core/linux/remote/*.la mkdir .libs libtool: link:
warning: library
`/usr/local/gostai/core/linux/remote/libkernel-remote.la' was moved.
libtool: link: warning: library
`/usr/local/gostai/core/linux/remote/libkernel-remote.la' was moved.
g++ -O2 -pthread -o urbiengine-linux-remote ./foo.o
-L/usr/local/gostai/core/linux/remote
/usr/local/gostai/core/linux/remote/libkernel-remote.a
-L/tmp/urbi/gostai/core/linux/remote

$ ls

foo.cc foo.hh foo.lo foo.o urbiengine-linux-remote
```

Specifying the source

You can pass to umake a list of files and directory. Files can be sources, headers and libraries. Directory will be searched and all the sources and libraries they contain will be included in the build.

```
$ ls -R .

.:
uobj1 uobj2

./uobj1:
myuobj1.cc

./uobj2:
myuobj2.cc

$ umake uobj1 uobj2

/usr/local/gostai/core/linux/libtool --tag=CXX --mode=compile g++ -O2
-pthread -I/usr/local/gostai/core/include -c uobj1/myuobj1.cc -o uobj1/1.lo
g++ -O2 -pthread -I/usr/local/gostai/core/include -c uobj1/myuobj1.cc -o
uobj1/1.o /usr/local/gostai/core/linux/libtool --tag=CXX
--mode=compile g++ -O2 -pthread -I/usr/local/gostai/core/include -c
uobj2/myuobj2.cc -o uobj2/2.lo g++ -O2 -pthread
-I/usr/local/gostai/core/include -c uobj2/myuobj2.cc -o uobj2/2.o
/usr/local/gostai/core/linux/libtool --mode=link --tag=CXX g++ -O2
-pthread -L/usr/local/gostai/core/linux/remote -o
urbiengine-linux-remote uobj1/1.o uobj2/2.o
/usr/local/gostai/core/linux/remote/*.la mkdir .libs libtool: link:
warning: library
```

```
`/usr/local/gostai/core/linux/remote/libkernel-remote.la' was moved.  
libtool: link: warning: library  
`/usr/local/gostai/core/linux/remote/libkernel-remote.la' was moved.  
g++ -O2 -pthread -o urbiengine-linux-remote uobj1/1.o uobj2/2.o  
-L/usr/local/gostai/core/linux/remote  
/usr/local/gostai/core/linux/remote/libkernel-remote.a  
-L/tmp/urbi/gostai/core/linux/remote  
  
$ ls -R .  
  
. : uobj1 uobj2 urbiengine-linux-remote  
  
./uobj1:  
myuobj1.cc  
  
./uobj2:  
myuobj2.cc
```

Specifying a different host or SDK

To compile for a different host, you can use the `-H host` option. A SDK for the specified host must be installed. To specify the SDK to use, `-C sdk`.

```
$ ls  
foo.cc foo.hh  
$ umake --core webots  
  
/usr/local/gostai/core/linux/libtool --tag=CXX --mode=compile g++ -O2  
-pthread -I/usr/local/gostai/core/include -c foo.cc -o foo.lo g++ -O2  
-pthread -I/usr/local/gostai/core/include -c foo.cc -o foo.o  
/usr/local/gostai/core/linux/libtool --mode=link --tag=CXX g++ -O2  
-pthread -L/usr/local/gostai/core/linux/webots -o  
urbiengine-linux-webots ./foo.o  
/usr/local/gostai/core/linux/webots/*.la libtool: link: warning:  
library `/usr/local/gostai/core/linux/webots/liburbicore.la' was  
moved. libtool: link: warning: library  
`/usr/local/gostai/core/linux/webots/liburbicore.la' was moved. g++  
-O2 -pthread -o urbiengine-linux-webots ./foo.o  
-L/usr/local/gostai/core/linux/webots  
/usr/local/gostai/core/linux/webots/liburbicore.a  
-L/tmp/urbi/gostai/core/linux/webots  
-L/tmp/urbi/gostai/kernel/linux/engine -L/usr/local/webots/lib  
-lController  
  
$ ls  
  
foo.cc foo.hh foo.lo foo.o urbiengine-linux-webots
```

Specifying the output file

The option `-o` can be used to set the output file name. It defaults to `urbiengine-HOST-CORE` when building engines, and `uobject-HOST.a` when building libraries.

```
$ ls  
foo.cc foo.hh
```

```
$ umake --core webots -o urbi

/usr/local/gostai/core/linux/libtool --tag=CXX --mode=compile g++ -O2
-pthread -I/usr/local/gostai/core/include -c foo.cc -o foo.lo g++ -O2
-pthread -I/usr/local/gostai/core/include -c foo.cc -o foo.o
/usr/local/gostai/core/linux/libtool --mode=link --tag=CXX g++ -O2
-pthread -L/usr/local/gostai/core/linux/webots -o
urbiengine-linux-webots ./foo.o
/usr/local/gostai/core/linux/webots/*.la libtool: link: warning:
library `/usr/local/gostai/core/linux/webots/liburbicore.la' was
moved. libtool: link: warning: library
`/usr/local/gostai/core/linux/webots/liburbicore.la' was moved. g++
-O2 -pthread -o urbiengine-linux-webots ./foo.o
-L/usr/local/gostai/core/linux/webots
/usr/local/gostai/core/linux/webots/liburbicore.a
-L/tmp/urbi/gostai/core/linux/webots
-L/tmp/urbi/gostai/kernel/linux/engine -L/usr/local/webots/lib
-lController

$ ls
foo.cc foo.hh foo.lo foo.o urbi
```

Passing parameters to make

All the umake options of the form 'var=value' will be passed to make. To pass flags to the compiler and the linker, use the variables EXTRA_CPPFLAGS and EXTRA_LDFLAGS.

```
$ ls foo.cc foo.hh $ umake --core webots *.cc
EXTRA_CPPFLAGS=-I/usr/local/webots/include -o urbi -V

/usr/local/bin/umake: run. /usr/local/bin/umake: libs=''
/usr/local/bin/umake: sources=' 'foo.cc' ' /usr/local/bin/umake:
headers='' /usr/local/bin/umake: objects=' 'foo.o' '
/usr/local/bin/umake: make options='
'EXTRA_CPPFLAGS=-I/usr/local/webots/include
-I/home/thomas/project/liburbi-cpp/trunk/lib' OUTBIN=urbi
URBI_ENV=webots prefix=/usr/local' /usr/local/bin/umake: invoking make
-f /usr/local/gostai/core/linux/param.mk urbi
/usr/local/gostai/core/linux/libtool --mode=link --tag=CXX g++ -O2
-pthread -L/usr/local/gostai/core/linux/webots -o urbi foo.o
/usr/local/gostai/core/linux/webots/*.la libtool: link: warning:
library `/usr/local/gostai/core/linux/webots/liburbicore.la' was
moved. libtool: link: warning: library
`/usr/local/gostai/core/linux/webots/liburbicore.la' was moved. g++
-O2 -pthread -o urbi foo.o -L/usr/local/gostai/core/linux/webots
/usr/local/gostai/core/linux/webots/liburbicore.a
-L/tmp/urbi/gostai/core/linux/webots
-L/tmp/urbi/gostai/kernel/linux/engine -L/usr/local/webots/lib
-lController /usr/local/bin/umake: done.

$ ls
foo.cc foo.hh foo.lo foo.o urbi
```

Examples

Suppose you have a module whose sources are in the current directory.

- **umake** produces **urbiengine-linux-remote**, a remote object executable

- **umake-lib -o mymodule** produces **mymodule.a**, a library that can be linked using umake to a sdk.
- **umake-engine -o urbi .libs/othermodule.a** produces **urbi**, an URBI server containing the module othermodule.a and the module made by compiling the sources in the current directory
- **umake-engine --core=aibo --host=mipsel-linux** produces **urbi**, an URBI engine for aibo which can run on the aibo host. This is the same as **umake-aibo**
- **umake-engine -H arm EXTRA_CPPFLAGS=/usr/local/myLib/include** produces **urbiengine-arm**, an URBI engine for arm architecture containing your object, if you have the arm-engine URBI SDK installed. It will search for headers in the /usr/local/myLib/include directory.

How to distribute one of your components and make them available to others?

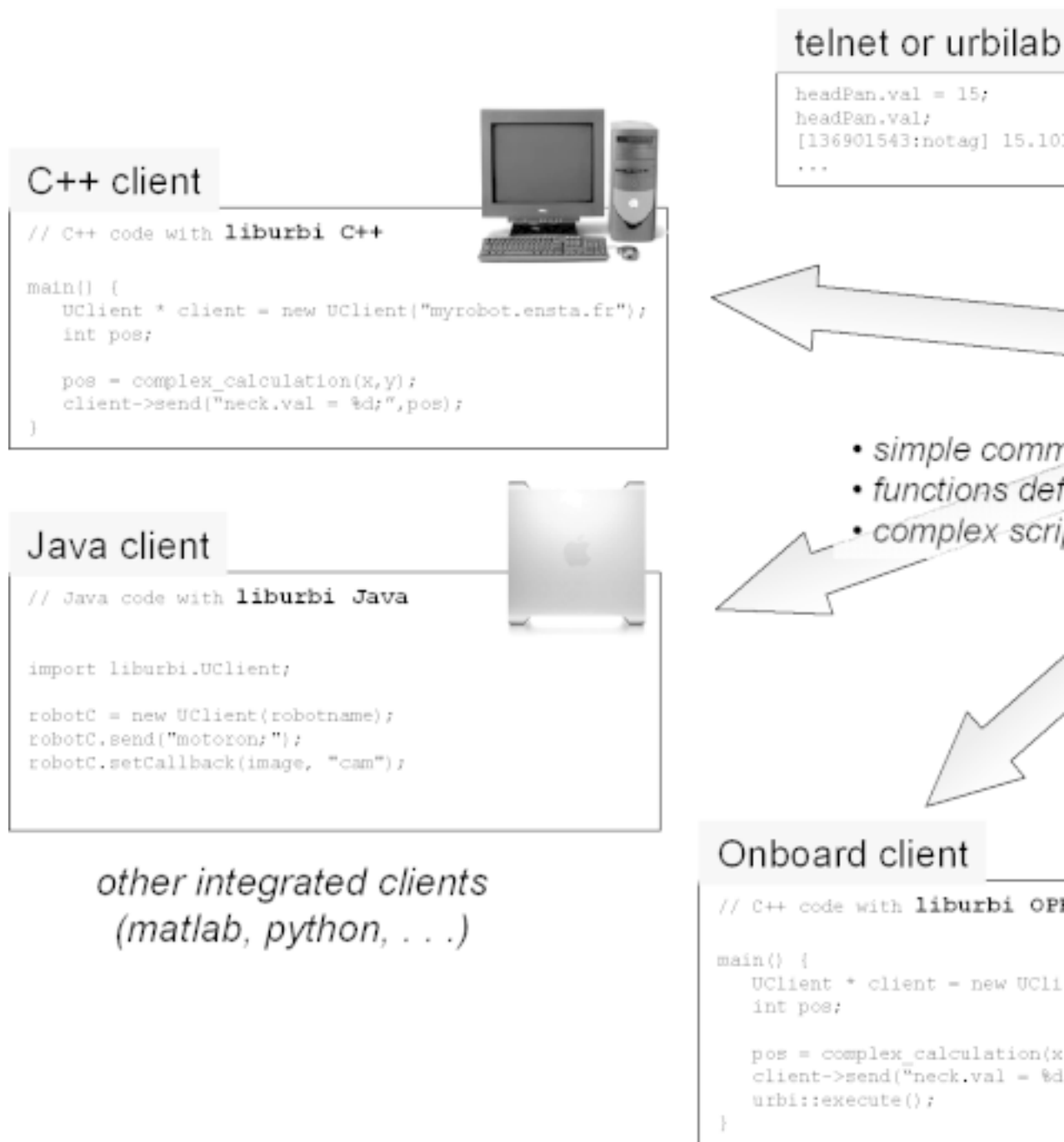
You can either distribute the sources of your component, or the library file (.a) generated by the build process as described above (umake-lib). People will then be able to link it to an URBI server as explained above. Please note that the library is architecture dependant: a component compiled for the Aibo can't be relinked as a remote module, the sources have to be recompiled. We strongly advice to publish both the source code for rebuilding purposes if your license allows it, AND one or several binary versions for people who just want to link it and use it as a remote component.

The website <http://www.urbiforge.com> is a platform to exchange components and URBI scripts. You can upload your work there so that the community can benefit from it.

Chapter 10. Putting all together

The following diagram shows a typical setting of clients and software architecture for an URBI application. You have clients in C++, Java and Matlab running on different machines (with Linux, Windows, Mac OSX), remote UObjects plus onboard clients and plugged UObjects and some telnet/URBILab scripting. There is also a bench of controlling scripts running from the URBI.INI file. This example shows how flexible URBI can be, having all those systems running in parallel to control your robot.

Figure 10.1. The general URBI architecture, putting all together



Typical usages examples

You have an URBI server running on your robot and...

1. Remote control of a robot

Your robot is equipped with a wifi connection, like Aibo, and you run a complex AI program on a powerful desktop computer to control it. This program is actually an URBI client written in C++

and uses the `liburbi C++` or `UObject` library to send URBI commands to the robot when needed and to receive URBI messages from the server asynchronously and react to them. You can replace `C++` by Java, Matlab or any language you like if you don't want `C++`. You also have some `UObject`-based components running some nice vision algorithm for example.

2. Fully autonomous robot with an onboard URBI client / `UObjects`

This time, you run the URBI client or `UObjects` on the robot and not remotely. Just like before, it is written in `C++` with the `liburbi C++` or with the `UObject` architecture. Instead of a TCP/IP wifi based connection between your client and the server, you have a direct interprocess communication on localhost or direct shared memory access with `UObject` plugin mode.

3. Fully autonomous robot controlled only by URBI scripts

In that case, it means that you have found all the functionalities you need in URBI (no need for external `C++` or Java programming) and you write directly all the action-perception loops with URBI scripts running in the URBI server, making use of some pre-existing or downloaded components in plugin mode. You need a simple telnet or URBI Remote to send your URBI scripts to the server and it is set. You can also store the script directly in the `URBI.INI` file and your robot will start it at boot up (no need at all for an external client or computer).

4. A mix of 1, 2 and 3

You have a robot controlled by several URBI clients at the same time, some on the robot, some on a desktop computer, some in `C++`, some in Java and Matlab. On top of that, you have several URBI scripts running in the server to perform reactive action-perception loops using some powerful `UObjects` written by you and also downloaded from the Internet, all this started from `URBI.INI` but also dynamically loaded by some of the clients when needed. This is the most interesting situation, making a full use of the URBI flexibility.

Appendix A. Copyright

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

1. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License. 2. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License. 3. "Licensor" means the individual or entity that offers the Work under the terms of this License. 4. "Original Author" means the individual or entity who created the Work. 5. "Work" means the copyrightable work of authorship offered under the terms of this License. 6. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

1. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the

Collective Works; 2. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Derivative Works. All rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Sections 4(d) and 4(e).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. 2. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works. 3. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; and to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit. 4.

For the avoidance of doubt, where the Work is a musical composition: 1. Performance Royalties Under Blanket Licenses. Licensor reserves the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work if that performance is primarily intended for or directed toward commercial advantage or private monetary compensation. 2. Mechanical Rights and Statutory Royalties. Licensor reserves the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions), if Your distribution of such cover version is primarily intended for or directed toward commercial advantage or private monetary compensation. 5. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor reserves the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions), if Your public digital performance is primarily intended for or directed toward commercial advantage or private monetary compensation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License. 2. Subject to the above terms and conditions, the license granted here is perpetual (for

the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License. 2. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable. 3. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent. 4. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.