The Urbi Universal Platform for Robotics

Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, Matthieu Nottale, Samuel Tardieu

Gostai, R&D Lab, 32 Bd Victor, FR-75015 Paris, France
http://www.gostai.com
lastname@gostai.com

Abstract. Robots can free humankind from everyday chores, they can entertain us, and even educate our children. They can carry loads, walk, dance, sing, and express emotions. Hundreds of different robots are already sold in shops, and complex applications are being developed actively around the globe. So why are robots so *not* present today?
In our experience the lack of standard in robotics, be it from the hardware or software point of view, makes the development of advanced applications for robotics *un*productive. This is very similar to the early days of personal computers, until the emerging of sufficiently widespread Operating Systems increased the return-on-investment for software development.
The Urbi platform sits on top of the large variety of software and/or hardware components for robotics, and provides the user with a unified, standardized, interface with which complex and portable applications can be developed. In this paper, with present the Urbi platform and some of its prominent components.

Introduction

Urbi is a software platform used to develop portable applications for robotics and artificial intelligence [1, 2]. It is based on a parallel and event-driven script language, and on a distributed component architecture.

The lack of standards in robotics makes the Urbi platform particularly sensitive to differences of Application Program Interfaces (APIs) between robots, components and so forth. In this paper, we present the Urbi platform and how it achieves its goals.

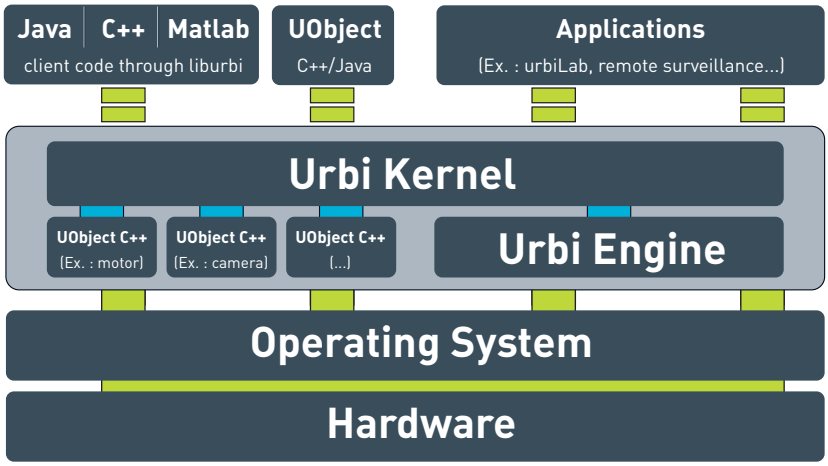
Outline Sect. 1 describes the whole platform, focusing on the problems to solve, and the selected, layered, solutions. On top of it portable applications can be built; Sect. 2 presents some of the applications developed at Gostai. A key component of the platform is the UrbiScript language[1], introduced in Sect. 3. Sect. 4 presents ongoing and future work, and Sect. 5 concludes.

1 A Universal Platform

The Urbi platform, including the UrbiScript programming language, was designed to be simple and powerful. It targets a wide spectrum of users, starting

[1] UrbiScript was formerly known as the Urbi programming language.

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591



The Urbi server is on top of the Operating System (OS) to abstract from (computer) hardware idiosyncrasies, yet it may be deployed bare-board. The Urbi server (the central rounded-box) is composite: (i) the Urbi engine and the kernel (Sect. 1.1) abstract the Central Processing Unit (CPU) and OS, and (ii) robotics or algorithmic components are unified using the *UObject* API (Sect. 1.2). Differences in conventions (clockwise? anticlockwise?), units etc. are addressed in Sect. 1.3. The intrinsically concurrent and event based nature of robotic software is fully integrated in the Urbi platform, including in the UrbiScript programming language (Sect. 1.4).

Fig. 1. The Urbi platform components

from children willing to customize their robots, up to researchers who want to focus on complex scientific problems in robotics rather that on idiosyncrasies of some robot's specific API.

As such, the Urbi platform has already reached its goal: its youngest known users are about twelve years old, using UrbiScript a hobbyist won a Sony "best Aibo dance" prize, and dozens of universities throughout the world develop advanced research in robotics using it.

Several challenges stand in the way of ease-of-use, starting with the need to cope with a wide range of architectures. The platform was structured to cope with the lack of proper standards for robotics, and to be simple yet powerful. A bird-eye view of its architecture in presented in Fig. 1, and the following sections detail the various components.

1.1 Computer Components

To gain CPU independence, UrbiScript runs on top of the Urbi Virtual Machine (UVM), which is part of Urbi *server*: to port UrbiScript to a new architecture, the UVM only needs to be adapted. The UVM is tailored to work with local

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

components, or device drivers, developed in low-level languages. UrbiScript is a scripting language: it was designed to develop high-level algorithms and behaviors from smaller components. For instance, computer-vision primitives are expected to be tailored for a specific robot and written for instance in C++, while UrbiScript is suitable to implement reactive behavior based on vision.

The Urbi platform relies on *engines* to interact with the underlying operating-system. Primitive services such as the UVM are provided by the Urbi kernel, which is a generic library. It is the engine that makes it a complete and runnable Urbi server. The engine launches the main loop and provides a few system-dependent core information to the server, such as time. Thanks to this modeling, the Urbi server can be ported very simply to a wide range of environments (embedded systems, robots, simulators, ...).

1.2 Robotics Components

The diversity of hardware components in robots is another source of complexity. In the Urbi platform, it is addressed by the UObject architecture. It installs a standard API on top of low-level device drivers (sensors, actuators, motors and so forth) and of software components (computer vision, voice recognition etc.). Thanks to this API and its associated middle-ware architecture, low-level and/or high-level components can communicate simply, and interact with UrbiScript programs. Existing C/C++ libraries are easily made usable from UrbiScript, and specific code can be written to take advantage of Urbi features such as event-driven programming, timers and variable change notifications. The UObject API can also be used to handle complex data flow between multiple software components thanks to *callback functions* that are notified when any variable is accessed (read and/or write). The UObject API maximizes code reuse: if a robot is built from off-the-shelf components, so is the Urbi server.

UObjects can either be *plugged* into the server, ensuring maximum reactivity, or be standalone *remote* processes. Remote UObjects can be local, running on the very robot, or on some slave computer. In the former case the server is protected from faults in the components, and in the latter case, CPU intensive computations can be handled by auxiliary machines. Therefore, before deploying it on a robot, the Urbi server is dimensioned, depending on the available resources.

Thanks to these abstractions the server does not know whether it is running on an actual robot, or a simulation thereof. UrbiScript programs run seamlessly on robots and simulators such as Webots [6, 12]. Developing robotic applications without simulators is virtually impossible. They considerably speed-up the development process: they are cheap and reproducible as opposed to robots that can be very expensive apiece, test suites are easier to write and to automate, etc. They don't even require the robot itself to exist: they provide a means to examine the behavior of a robot during its design.

Low-level consistency, while necessary, is not sufficient to maximize the portability of high-level applications.

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

1.3 High-level Universality

One can expect a dramatic increase of the market of hardware and/or software components for robots: motors, sensors, text to speech, voice recognition, navigation, face recognition... to name a few. To cope with this variety, the Urbi Naming Standard defines common interfaces (or *facets*) and naming scheme that must share the UObjects for equivalent components. Thanks to this abstraction, the customer is free to select a particular component seller depending on his own criteria.

Subtle differences between components can ruin a multi-million-dollar project [13]. To avoid this, unit support is developed in UrbiScript. The Urbi Naming Standard also defines the orientation axis and so forth, to standardize measurements and actions. This document also defines the names and interfaces of limbs and organs for usual robot anatomies: legs, fingers, heads, joints, sensitive skin, accelerometers, gyroscopes etc.

These abstraction and standardization layers allow the development of portable applications (on sufficiently similar robots). Games or utilitarian applications are being developed at Gostai: UrbiLab (Sect. 2.2) to create elaborate remote control for any robot, remote surveillance, educative application etc.

1.4 Concurrent and Event Driven

Traditional programming languages follow a sequential execution model. Code is executed one statement after the other, ultimately leading to the expected result. Even when threads are used, the code running in each thread follows a strict iterative scheme, and their interaction may be difficult to describe [7].

Robots are designed to evolve in the real world: while they are moving around, they need to be aware of their environment and its possible brutal changes. Unpredictable events such as an animal crossing the robot path need to be asynchronously and timely acted upon. While traditional programming languages typically offer limited asynchronous capabilities through signals and callbacks, the Urbi platform integrates them at the heart of the execution model, and the UrbiScript language provides the user with short and intuitive constructs, which we will describe in Sect. 3.2 and 3.4.

1.5 Clients/Server Architecture

Like other environments such as Player/Stage [10], the Urbi platform features a client/server architecture. Several clients (human users, remote components, ...) can interact simultaneously with the server, via UrbiScript commands. This is easily achieved since the server has concurrency built-in features made available in UrbiScript, and thanks to the event-driven nature of the language. The communication protocol is based on a restricted subset of UrbiScript. The liburbi library provides an API to the middle-ware and communication layer.

This server architecture has several advantages.

Several (human) users can evaluate code simultaneously on the same server. They can share the control of a robot, or to let one user move it around while another uses the camera feedback. This make UrbiScript more “interactive” than

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

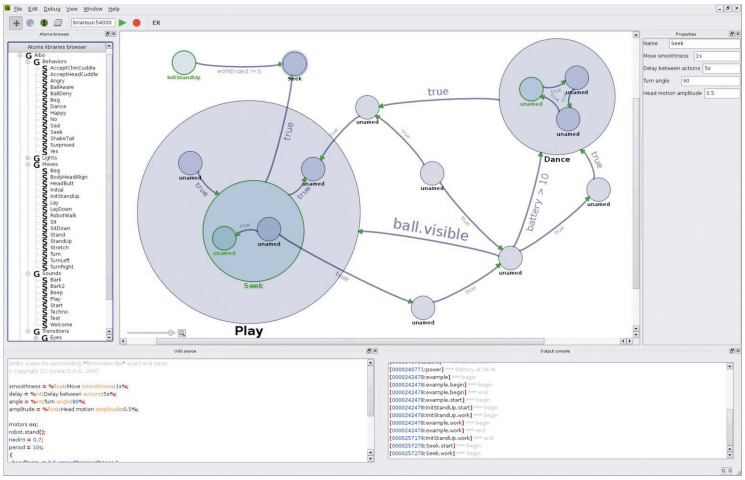


Fig. 2. The UrbiLive behavior editor

most other interpreted languages: while scripting languages are usually geared toward writing scripts, UrbiScript is designed to be used directly interactively to control robots. It behaves like a remote *shell* (an interactive command-line interpreter) used to command robots at distance.

It is also the C/S architecture that allows to support remote UObject components compiled as autonomous applications: they are simple clients. They connect to the server, issue commands and retrieve results.

Using the Urbi communication library, one can connect various kinds of programs to the server that act like interfaces. Complex applications that interact with an Urbi server are be developed, sending commands and exchanging values. For instance, Gostai develops several Graphical User Interface (GUI) to program or remote-control a robot (see Sect. 2).

Thanks to bindings of the Urbi communication library in various languages, Urbi is open to environments such as Java, MATLAB, and Python. Although the Urbi platform makes a special place to C++ and UrbiScript, one does not need to know these languages to program robots or components.

2 Top-Level Applications

Thanks to the server architecture, complex applications can interact transparently with an Urbi server. This enabled Gostai to develop several powerful graphical applications that ease interaction with robots.

2.1 UrbiLive

UrbiLive (Fig. 2) is a behavior editor. One graphically composes and chains actions depending on external events in a graph-set similar fashion. Behaviors

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

are represented by directed graphs whose vertices are actions and edges are transitions labeled by events. The behavior starts in a given node, and executes its action. When one of the conditions on the outgoing edges is verified, the transition is made and the action of the activated node is executed, and so on.

The graphs are hierarchical: every node can contain subgraphs, every action can be decomposed into a graph. Several states can be active simultaneously, for instance in several subgraphs of the current node. This makes designing high-level concurrent, event-driven behavior easier.

Under the hood, UrbiLive is a standard client connected to an Urbi server, as a human would be. It simply generates UrbiScript code for the input graph, and sends these commands to the server.

2.2 UrbiLab

UrbiLab [5] is a graphical Urbi server inspector and effector. It enables to represent and arrange as graphical widgets Urbi values to monitor and modify them. It can be used to create remote controls for robots, including video feedback and motors controls. Every widget is related to one or several variables in the Urbi world. Behind the scene, UrbiLab is a regular Urbi client. Thanks to serialization and introspection, it can inspect and modify Urbi values.

3 The UrbiScript Programming Language

Much of the power of the Urbi platform comes from its architecture, and its language, UrbiScript. Of course there are already too many programming languages, so why a new one? Why not extending an existing language, or rely on some library extensions. While this debate is beyond the scope of the paper, several observations must be made.

First, Domain Specific Language (DSL) are gaining audience because they are much more productive than the library-based approaches. No one would consider managing a database without a DSL such as SQL. Today, research is very active to embed SQL within general purpose languages to improve security, to expose new opportunities for domain-specific optimizations and so forth. Second, programming robots requires a complete rethinking of the execution model of traditional programming languages: concurrency is the rule, not the exception, and event-driven programming is the corner stone, not merely a nice idiom. These observation alone justify the need for innovative programming languages. Yet, programmers are usually, and for good reasons! reluctant to learn new languages. This is why UrbiScript is trying to stay with the (syntactic) spirit of some major programming languages: C++, Java, JavaScript etc.

Not only is UrbiScript designed to write robotic applications and to interactively control robots, it is also the foundation of the communication protocol on top of which is built the client/server architecture.

In the following interactive session examples, lines starting with `[00000000]` are answers from the server; the others were entered by the user.

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

3.1 Prototype-Based Object-Oriented Language

With respect to the syntax, UrbiScript belongs to the C family: it is inspired by C, C++, Java, and so forth. UrbiScript is dynamically typed, introducing variables does not require to make their type explicit.

Functions are *first-class entities*: they behave like ordinary values. They can be assigned to variables, passed as arguments to functions and so forth. UrbiScript supports *closures*: functions can capture references from their environment, then later use those references to retrieve or set their content.

UrbiScript is an Object-Oriented Language (OOL): values are *objects*. An object is composed of a list of *prototypes* (parent objects) and a list of *slots*. A slot maps an identifier to an object. UrbiScript is *prototype-based*, or *classless*, like Self [9], Io [3] and others. In class-based OOL, *classes* are templates that describe the behavior and expected members of an object. They are *instantiated* to create a value; for instance the `Point` class serves as a template to create values such as `one = (1, 1)`. The object `one` holds the (dynamic) values while the class captures the (static) behavior. Inheritance in class-based languages is between classes. In classless languages instantiation is replaced by *cloning*: an object serves as a template for a fresh object. Inheritance relates objects.

UrbiScript is fully dynamic. Objects can be extended at run-time: prototypes and slots can be added or removed.

```
var one = Object.clone();    var one.x = 1;    var one.y = 1;
function one.asString() { return "(" + x + ", " + y + ")"; };
one;
[00000000] "(1, 1)"
one.protos;
[00000000] [<Object>]
```

Operators are functions that have special syntactic properties: they are prefix, infix or suffix, and obey to associativity and precedence rules.

```
// Truncate strings longer than len.
function String.operator/(len) {
  if (size() < len)
    return this
  else
    return this[0, len] + "...";
};
```

UrbiScript features *introspection*: objects can be examined and modified at run-time.

```
one.slotNames;
[00000000] ["*", "+", "asString", "y", "x"]
for (var s in one.slotNames)
  echo(s + " = " + one.getSlot(s).asString() / 14);
[00000000] *** "* = function (rhs)..."
[00000000] *** "+ = function (rhs)..."
[00000000] *** "asString = function () {\n..."
[00000000] *** "y = 1"
[00000000] *** "x = 1"
```

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

3.2 Concurrency Support

UrbiScript has been designed from the beginning as a concurrent language. Many threads may be run in parallel, and a global scheduler takes care of giving each one its fair share of time.

Every code block may be dynamically marked with one or several *tags*; actions on tags allow to freeze, unfreeze or stop code execution. Tags are the basis of every flow control operation in UrbiScript. Constructs such as `return`, `break`, `continue`, and exceptions are implemented using tag operations.

The following code snippet ensures that the call to `f` will either terminate in less than `maxdelay` or will return `nil` and print a message if it takes longer:

```
// Implement the functionality of the built-in ''timeout'' construct.
function timeOut(f, maxdelay) {
  var checker = new Tag();
  checker: {
    // Setup the guard in a background thread (using '','').
    {sleep(maxdelay); checker.stop()},
    return f(); // Call the user-provided function.
  };
  // We are here because ''checker'' has been stopped.
  echo("The computation has been interrupted");
  return nil;
};
```

The statement run in background, launched with a comma at its end, is also subject to being killed through the `checker` tag, thus aborting it if the call to `f` terminates before the allotted time-span.

The Urbi scheduler has been designed to handle many parallel threads of control through the use of *coroutines*. Coroutines are light threads with their own stack and execution pointer implemented as a library, and they are commonly used in modern interpreted languages such as Io [3]. They allow the use of the Urbi kernel on robots whose operating system does not provide support for multiple processes or kernel threads.

Multiple prioritized tasks can run concurrently in the same Urbi kernel. Tasks are either created explicitly by the user or implicitly by certain constructs, such as `timeout` which provides as a built-in the functionality demonstrated in the previous example. UrbiScript also offers high-level objects such as *semaphores* and *barriers* in the core language to synchronize concurrent access to shared data.

In addition, the user can indicate that several computations must occur in parallel and wait for all of them to terminate by using the `&` separator:

```
// Execute ''f'', ''g'', and ''h'' in parallel, then ''u''.
f() & g() & h();
u();
```

3.3 Time Management

UrbiScript enables to write very simply time-related code, thanks to parallelism separators and other primitives. It provides many constructs to execute

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

code at given intervals, with a timeout, ... However they are not primitives and can be expressed directly in Urbi, as shown in the timeout example.

The UrbiScript standard library provides assignment modifiers to generate trajectories, which is very useful when manipulating motors for instance.

```
// Move to position 10 in 5s.
motor.val = 10 time: 5s;

// Move to position 0 with a maximum acceleration of 0.1 unit/s^2.
motor.val = 0 acc: 0.1;

// Oscillate around 1 with period π and amplitude 5.
motor.val = 1 sin:pi ampl:5;
```

The UrbiScript standard library provides smooth tests, which for instance can test whether a condition is true for a given duration, which is very useful to filter out noise.

```
// React when the sensor value is greater than 10 for at least 1 second.
at (sensor > 10 ~ 1s)
  react();
```

Timeout example When programming interactive behaviors, one very common need is to perform an action with a given timeout, or an action that must be executed periodically. While this is often painful to write in classical languages, it can be written very simply in UrbiScript thanks to temporal separators:

```
// ''timeout (maxdelay) action'' executes ''action'' and aborts it
// if it takes longer than ''maxdelay'' to execute. This builtin is
// similar to the ''timeOut'' function defined in an earlier example.
function f() {
  sleep(1s);
  echo ("f is terminating");
};

// Test it.
timeout (2s) f();         // No timeout, ''f'' terminates.
[00000000] *** f is terminating
timeout (500ms) f();      // Timeout triggered, ''f'' is aborted.

// ''every (delay) action'' executes ''action'' every ''delay''.
timeout (2.5s) { every (1s) f(); };  // Will execute three times.
[00000000] *** f is terminating
[00000000] *** f is terminating
[00000000] *** f is terminating
```

Many more UrbiScript constructs allow the user to mix time-based and event-based programming. Using them, complex applications with numerous concurrent behaviors can be written painlessly.

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

3.4 Event Driven Communication

To accompany its concurrency features described in Sect. 3.2, the Urbi kernel offers a full-featured *events* mechanism. The user can define any number of events, and install *event handlers* that will be executed when the corresponding event is emitted.

Events can carry values that will be matched against patterns given in the event handlers; the corresponding action will be executed only if the system can unify the values carried in the event with the provided pattern:

```
// Create a new event called ‘‘e’’.
var e = new Event;
// Install an event handler for ‘‘e’’.
at (?e(5, (var b))) { echo("Received e(5, " + b + ")"); };
// Emit event ‘‘e’’ twice with different values.
emit e(1, 3);     // No match, nothing is printed.
emit e(5, 7);
[00000000] *** Received e(5, 7)
```

In addition to the events system, Urbi provides a *publish-subscribe* mechanism allowing one or several tasks to subscribe to channels, while other tasks will publish objects into those channels [4]. Every subscriber will receive a copy of the data in the same order as it has been published.

The following example illustrates a logging system that presents diagnostic messages to the user with a maximum of one message per second:

```
// Create the publish/subscribe object.
var logpool = new PubSub;
// Launch a background rate-limiting logger (note the comma at the end).
{
  var id = logpool.subscribe;
  while (var msg = logpool.getOne(id)) {// Retrieve message.
    echo(msg);                          // Print it.
    sleep(1s);                          // Limit rate at 1 msg/s.
  };
},
// Create a function logging a message and returning immediately.
function log(msg) {
  logpool.publish(msg);
};
// Log two messages in a row, they will be printed with a 1s interval.
log("first message"); log("second message");
[00000000] *** first message
[00000000] *** second message
```

The publish-subscribe paradigm may easily be used to create more complex abstractions. For example, the event system uses it internally, and building mail-boxes such as found in Erlang [11] or rendez-vous such as found in Ada [8] is straightforward.

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

3.5 Features for Robotics

Other features were specifically designed to make the implementation of robotics applications easier, but their description goes beyond the scope of this paper. *Groups* allow to manipulate a set of equivalent objects as a whole.

```
class Leg {
  // Leg constructor, invoked by ''new''.
  function init(name) { var this.name = name; };
  function walk()     { echo("Leg " + name + " walking."); };
};
var legl = new Leg("left"); var legr = new Leg("right");
var legs = new Group(legl, legr);
legs.walk();
[00000000] *** Leg left walking.
[00000000] *** Leg right walking.
// Queries to a group yield groups of values to enable chaining queries.
legs.name;
[00000000] Group ["left", "right"]
legs.name = "central";
legr.name;
[00000000] "central"
```

4 Future Work

As described in Sect. 3.2, the Urbi kernel uses coroutines to implement concurrent control flows. It is likely that in a few years more and more robots will have multiple processors, either strongly connected and sharing memory, as in desktop multiprocessor systems, or loosely connected through a communication bus, as in distributed automotive high-end cars. We plan to implement support for those two configurations through the use of automatic remote calls, thus allowing the user to balance the load between the processors. We then plan to add live code migration to Urbi, paving the way to fault tolerance and automatic reconfiguration.

5 Conclusion

We introduced the Urbi platform as it is today. Since previous versions [1, 2] the core programming language, UrbiScript, and its Virtual Machine (VM) were completely redesigned to provide a richer set of features tailored for the development of robotic applications. A significant part of the efforts was devoted to the establishment of standard interface to cope with the diversity between existing components and the unstoppable creativity of newcomers. Our approach was successfully used to deploy a few complex applications on top of several very different robots. Yet much remains to be done to achieve the level of standardization that a mature robotic market needs to finally be able to hold its promises: a robot in every home, or almost.

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591

Bibliography

[1] Jean-Christophe Baillie. URBI: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, pages 820–825, 2005.

[2] Jean-Christophe Baillie. Design principles for a universal robotic software platform and application to URBI. In Davide Brugali, Christian Schlegel, Issa A. Nesnas, William D. Smart, and Alexander Braendle, editors, *IEEE ICRA 2007 Workshop on Software Development and Integration in Robotics (SDIR-II)*, SDIR-II, Roma, Italy, April 2007. IEEE Robotics and Automation Society.

[3] Steve Dekorte. Io: a small programming language. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 166–167, New York, NY, USA, 2005. ACM.

[4] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[5] Gostai. The UrbiLab remote control environment for robots, version 1.6. `http://www.gostai.com/urbilab.html`, 2008.

[6] Olivier Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[7] John Kenneth Ousterhout. Why threads are a bad idea (for most purposes), January 1996.

[8] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1.* 2007.

[9] David Ungar and Randall B. Smith. SELF: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22, pages 227–242, New York, NY, 1987. ACM Press.

[10] Richard T. Vaughan, Brian Gerkey, and Andrew Howard. On device abstractions for portable, resuable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'03)*, pages 2121–2427, October 2003.

[11] Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall PTR, 2 edition, 1996.

[12] Webots. http://www.cyberbotics.com, 2008. Commercial Mobile Robot Simulation Software.

[13] Wikipedia. Mars climate orbiter — Wikipedia, the free encyclopedia, 2008. [Online; accessed 8-September-2008].

Workshop Proceedings of SIMPAR 2008
Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS
Venice(Italy) 2008 November,3-4
ISBN 978-88-95872-01-8
pp. 580-591