Design Principles for a Universal Robotic Software Platform and Application to URBI

Jean-Christophe Baillie

***Abstract*— The number of generic robots available today on the market is increasing significantly. While this represents a great opportunity to carry on research using advanced robotic devices, including humanoid robots, there is still a serious limitation: all these robots are totally incompatible in term of software, and they are generally quite hard to program, lacking the proper abstractions and properties needed in robotics. We will present here a discussion on possible design rules for such a universal software platform and review a few already existing platforms with these criteria in mind. We will also introduce the URBI platform which is the result of our effort in an attempt to design a universal platform following these guidelines.**

I. INTRODUCTION

Looking at what happens today in Japan, Korea and the US, it appears that the number of generic robots available on the market will be increasing significantly in the coming years [9]. By ”generic robots”, we mean robots not specifically designed for an industrial task but general purpose robots, commonly usable in labs for research. Many of these robots, including humanoid robots, integrate cameras, wifi and on-board computers, and most of them are available for an affordable price. This is a great opportunity for research but while the hardware is making progresses, there is still no clear generic software platform emerging that could make these robots compatible and bring the benefits of cross-robot compatibility and component reuse. This is a major issue and several studies have highlighted the benefits of having such a universal software platform [4], [8], [5], [6].

Of course, robot programming can be done in C++, even in C, and to a certain extend every software platform is formally identical. But we believe that the debate should not be between using Java vs C++, or using a given technology vs another. The important question is to think about good properties of such a universal platform that would facilitate its adoption by a vastly heterogeneous community, mostly knowledgeable and sometimes already on-track with a software platform of their own. How could such a paradox be solved: to have a universal but also very flexible platform, which still brings value added in the process and does not stand just as an empty shell, while providing a clear orientation towards robotics. This will be the topic of our discussion in this article.

We claim that the URBI software platform developed in our lab makes a significant progress in the direction of such a universal platform. We will shortly present the key characteristics of URBI and compare it to other existing software platforms.

This work was supported by Gostai S.A.S., France

J.C. Baillie is with the National Institute of Advanced Technologies, Cognitive Robotics Lab, 75015 Paris, France jean-christophe.baillie@ensta.fr

II. WHAT ARE THE KEY REQUIREMENTS FOR A UNIVERSAL ROBOTIC SOFTWARE PLATFORM?

The challenge that we are going to investigate is: how could we design and promote a universal robotic software platform in the research community, what are the key requirement for such a platform?

As a guideline, there are three main questions that we should keep in mind when we investigate the potential features: are these features necessary, acceptable and usable by the community?

- Being *necessary* is an obvious requirement, however sometimes overlooked by “over-featured” solutions that can drawn the user in useless complexity.
- Being *acceptable* means that the community can realistically accept to use this feature which by essence creates certain constraints or enforces a point of view. For example, forcing the use of the C language exclusively or forcing the use of a very constrained class-based architecture will not be acceptable by many users.
- Finally, *usability* means that, however necessary and acceptable the feature is, it should be relatively easy to understand and integrate. The risk is otherwise to have a kind of ”complexity barrier” that prevents the widespread usage of the platform.

Breaking one of the three above requirements might be one the main reasons why we have not seen any successful platform emerging yet.

We detail below four key characteristics that we believe are required for a successful universal platform and whose implementation should fit with the necessary/acceptable/usable constraints presented above.

A. Flexibility

Being flexible means that the platform should be working with any operating system, be interfaced easily with any programming language and of course be suitable for any type of robotic application.

A non flexible platform will typically enforce several design choices and will, to a certain extend, constrain the user into a predefined way. While this has the advantage to improve general coherence and structure, it is also generally not acceptable (in the sense defined above) by the community. In fact, many research topics are precisely about finding a good architecture, a good paradigm or structure for robotics.

One can always bypass these constraints but at the cost of heavy custom-made layers. The difficulty here is that most platform developers claim to have a flexible solution, in the sense that it is possible to do anything with it, as long as one is ready to put enough effort in the process. A C++ library, like OPENR on the Aibo[10], is a good example. Here, the notion of *usability* that we have introduced before plays a central role. The flexility must be easy to implement, natural (using well known concepts, like objet oriented programming) and possibly transparent: no need to read a long documentation to know how to use it, and no need to develop, port or recreate preliminary tools/interfaces to benefit from it.

Note that non-flexible platforms have been imposed already in other domains, like desktop computers, but there was already a quasi monopolistic player to enforce it, which is not the case today for robotics. So we believe that a non-flexible platform (in the *usable* sense) will unlikely become a widely used universal robotic platform.

B. Modularity

Modularity adds the possibility to plug components in the platform to extend its capabilities. This is necessary to be able to start cooperations between labs and the industry, by developing and reusing modules for robotics. It is the basis of a software industry for robotics. The need for modularity is widely recognized, see for example [11], [15], [12]. Modularity is also imperative to move towards plug&play hardware components for robotics.

Again, this constraint will be acceptable only if it is very easy to deploy. Anyone should be able to develop components and the unavoidable structuring constraints should be reduced to the minimal, for the solution to be usable.

C. Powerful abstractions for robotics

Robotics is unlike classical computer related domains. The main difference is that robots have to explore hypotheses, runs things in parallel, react to events at various levels and generally speaking integrate vastly more software technologies than a regular PC. Among the important technical differences, parallelism and event-based programming are two core requirement in any robotic application. The underlying platform must support these abstractions natively.

Thread-based libraries on top of C++ or other languages do not offer the appropriate abstraction needed for robotics. The limitations of threads have been widely discussed already [16] and it is now recognized to be a major issue in parallel application development, especially now that we can see quad-core processors on the market. The main limitation of threads in their most primitive declination is that memory safe locking and code synchronization, which is are very challenging problems, are left to the programmer. This rapidly lead to bugs, even for careful and experienced programmers.

Something new is needed to handle parallelism at the right level of abstraction, just like object-oriented programming brought something new to modular development. The benefits of the abstractions available in C++ today compared to C are now widely recognized, even if some resistance existed at the beginning, and we claim that such a paradigm shift is needed for robotics as well, to integrate parallelism and event-based programming at the core of the platform.

Introducing some smart C++ classes to handle parallelism will raise the problem of flexibility. Theses classes will not be available in Python, Java or Matlab and, even if it is possible, a important effort would have to be done to provide a unified set of abstractions for all possible languages. A intermediary programming scripting language, with proper interfaces with other languages and integration of parallel and event-driven abstractions is the direction we have chosen. More and more, scripting languages are used as glue between other compiled languages. Python or Ruby are two famous examples. What these languages lack however for the moment is the necessary abstractions adapted for robotics, at least with a sufficient level of *usability*, as described above.

D. Simplicity

The above requirements generally tend to give birth to very complicated software architectures which are confusing and prevent the users to adopt them. Simplicity for *usability* is a requirement of both the modular component architecture and of the core platform technology in general.

It is of course hard to assess what is difficult and what is simple, it will highly depend on the level of experience of the user. This criteria can only be enforced by following some common sense principle: reuse notions that exist, limit the number of abstractions, build layered architectures where the complexity of the knowledge necessary to perform a task is proportional to the complexity of the task, etc.

It is however possible to measure the level of simplicity reached once the platform has been released by looking at the average user appreciation. This measure should be done by any platform developer to validate his approach.

III. Adoption strategies

Beyond the technical questions mentioned above, the question of the promotion of the platform is a generally underestimated problem. The issue is not only to have the platform adopted by the research community but also by robot manufacturers. Relying on standardization comities is a long and uncertain process, and releasing open-source version on the Internet is generally not enough to attract the proper attention on the solution, at least not in a short time. In fact, some marketing is required.

Active promotion and continuous development, maintenance and support are typically demanded by robot manufacturers. This support can be achieved by a gathering of universities granted with appropriate funding and political will, or by a private company spin-off from the lab and supported by the community. We believe that individual initiatives or small lab projects will not be sufficient to efficiently promote a universal software platform for robotics. The issue of ”adoption strategies” should not rely only on

the quality of the solution proposed, but deserves its proper attention and financing.

IV. The URBI Platform

The URBI platform, developed initially by J.C. Baillie[1] in the Cognitive Robotics Lab of ENSTA (Paris), is an attempt to design a universal robotic software platform that follows the above guidelines.

The URBI platform is based on a client/server architecture, built on top of a new programming language called URBI. It is possible to log into the robot with a simple telnet client and start to enter URBI code to control the robot and the software inside the robot. More advanced graphical and developer-oriented tools are also available.

We detail here the main characteristics of URBI. This is a very brief introduction, the reader is invited to get more information in [2], [1], [3].

A. Why a new programming language?

There are already so many programming languages, why should we add a new one to the list? The novelty of URBI is that it brings new abstractions to handle parallelism and event-based programming, directly integrated into the language semantics. A component architecture is also directly integrated in the language via a familiar object oriented approach.

In terms of the general look & feel (types, control structures, objects, etc), URBI is fairly similar to other scripting languages like Ruby, with a syntax based on C/C++ to make it as familiar as possible to new users (*usability* constraint).

1) Parallelism: Parallelism is not handled via threads or callbacks in the language, instead URBI introduces four command separators: the classical semicolon, and the new comma, pipe and ampersand. While the semicolon has its usual serial semantics, the other separators add more possibilities, including parallelism, to state how two commands should be executed. Figure 1 illustrates the four semantics.

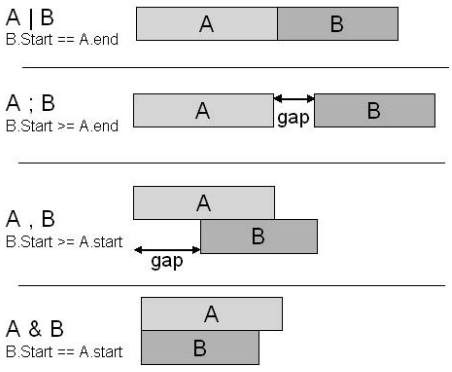

Fig. 1. Four types of command separators in URBI

Note that separators like pipe or ampersand are *strict*, since they enforce respectively an immediate serialization or parallelization of the commands. This can be used to precisely describe synchronization processes. Comma and semicolon however are *loose*, and the semantics allows for a gap to exist.

One can build complex serial/parallel constructs very easily. In the following code, C will start as soon as A and B have both finished:

```
{ A & B } | C;
```

Threads are not required as they are transparently handled by the URBI scheduler running in the language kernel interpreter, thus providing the right level of abstraction needed for parallelism.

2) Event-based programming: Another important abstraction in URBI is how events can be handled in the language. Consider the following code:

```
at (condition) action;
```

The action part will be executed once, each time the condition becomes true. Another event handler is `whenever`, which will execute the action in loops whenever the condition is true and as long as the condition remains true. These two commands are a bit similar to `if` and `while`, but they remain in background and will constantly monitor the condition to trigger the action if necessary.

It is also possible to emit events with parameters to send signals between codes running in parallel or between clients. The example below will trigger `action(42)`:

```
at (myevent(x)) action(x);

emit myevent(42);
```

We will see in the next section an example that combines parallel and event-based programming together with the component architecture.

B. The UObject component architecture

URBI is an object oriented language, so you can use and define objects as usual. What is more important is that you can also plug C++ objects into the language very easily, which is the base of the component architecture available in URBI. The C++ object will then be visible and usable as any other object in URBI; it is called a ”UObject”.

UObjects can be plugged, but they can also be unplugged and then ran remotely as autonomous executable programs, taking the address of the URBI Engine server as a parameter. This allows to integrates distributed object management directly in the language itself. For *usability*, note that, as we already explained, the C++ code of the UObject is the same whether it is plugged or ran in remote mode, it does not even need to be recompiled and it can be switched at runtime (relocatable objects).

CORBA, RT-Middleware or openHRP objects can also be seen from within URBI, like in fact most existing distributed object architecture, thus making URBI a central platform to integrate various technologies.

In summary: URBI plays the role of a central coordinator/hub for a set of plugged or remote heterogeneous objects, with parallel and event-based built-in capabilities. Shared memory and message passing can be done simply via the language structures.

Example

The example below can be used on the Aibo robot (there is a free version of URBI for Aibo available on www.urbiforge.org), and it will track the ball whenever the robot sees it:

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

`headPan` and `headTilt` are two UObjects controlling the head motors, and `ball` is a UObject detecting and localizing the ball in the image. Typically, `headPan` and `headTilt` will be plugged in the robot like "hardware drivers", while `ball` or other computer intensive objects can be running remotely outside of the robot. The UObject code is exactly the same in both cases (it is not even necessary to recompile it) and the way to use the object inside URBI is identical, which brings important features in term of transparent distributed object management and usability.

Note that the UObject architecture provides ways to get a notification whenever a specified variable is modified in URBI. For example, the `ball` UObject requests a notification on `camera.val`, thus making `ball` aware of any new image to process.

This simple example shows also how URBI enables to control motors by assigning the 'val' attribute of a motor object in the language. This is very simple and can be used in educational applications.

The UObject components in C++ have been designed with simplicity in mind. A limited set of small changes is necessary to port an existing C++ class into a UObject. Extensions to Java and C# are in development, to extend the *acceptability* of the feature.

C. Advantages of the client/server approach: liburbi

The client/server approach allows to interact with URBI through a socket connection, from any existing programming language. Liburbi is a set of GPL libraries for various languages like C++, C, Java, Matlab, Ruby, Python and others, that allows to create connections to an URBI Engine, send commands and receive messages asynchronously. Liburbi is designed with simplicity in mind, as a proxy to the URBI Engine and it does not add new abstractions or complexity to the platform, which is essential for *usability*.

In the most degraded case, URBI can be seen as a simple driver for the robot hardware, and can be used with liburbi from within any programming language to provide a common interface to any robot.

D. Advanced features for robotics

In addition to parallelism, event-based programming and the object oriented UObject component architecture, URBI brings several unique features that we briefly review here. These features are direct consequences of the parallel nature of the language and would not be semantically well defined otherwise.

A simple assignment in URBI can target a variable to reach a value in a given time or at a given speed, or set a sinusoidal oscillation on it. The assignment is not instantaneous anymore and can be run in parallel with others:

```
  neck.val = 10 time:450ms
& leg.val  = -45 speed:7.5
& tail.val = 14 sin:4s ampli:45;
```

Variables have a blend mode which specifies how conflicting simultaneous assignments should be handled - an extension of the concept of 'mutexes':

```
x->blend = add;
x = 1 & x = 3;
//now x equals 4
```

Any portion of code can be prefixed with a tag. It is then later possible to stop, freeze, unfreeze this code from anywhere using the tag name, which brings powerful features to control the flow of execution of parallel codes:

```
mytag: { some code };
stop mytag;
freeze mytag;
unfreeze mytag;
...
```

Hierarchical tags and multi-tags are also available.

More details about the advanced features of URBI can be found in the tutorial or reference manual [2].

E. Limitations

Currently embedded versions of URBI are limited to robots who have a reasonably powerful CPU embedded (typically ARM7 or more). It is written in C++ and must be compiled for the hosting platform. This limitation is counterbalanced by the fact that most robots can be controlled remotely via a serial link or wifi connection, URBI running on a PC or a Mac on the side. The general trend in robotics is also to have more and more linux based robots with sufficient CPU onboard.

Another limitation of URBI is the need to develop a set of hardware UObjects for each new robot. This is similar to the problem of driver development. Most of the time however, the robot comes with a simple C API that can be wrapped inside C++ UObjects in a few days. URBI is not an Operating System and will rely on existing interfaces to the hardware, thus limiting the driver development problem.

The real time aspect in URBI is dependent on the underlying operating system. If the OS is capable of real-time scheduling, then putting URBI in the highest priority

will allow to guarantee a first level of real-time features in URBI (at the level of URBI commands, not plugged UObjects). The second stage is to include in URBI some real-time oriented features, like priority flags for commands. This is in development for version 2. Finally, work has to be done on integration of UObjects in the real-time framework, and integration with existing real-time platforms like RT-Middleware.

Finally, the current limitation of URBI is its relatively limited adoption and youth. There are currently nine different robots that run URBI and one simulator (Webots), and about 20 to 25 universities that use it on a daily basis. One spin-off company of our lab is currently promoting URBI towards robot manufacturers to extend its coverage.

V. Comparison of URBI with existing platforms

URBI has been designed with the four above key constraints in mind: flexibility, modularity, parallel abstractions for robotics, and simplicity. Other attempts are currently made in the same direction and we will shortly review some of the most significant here.

A. Player/Stage

Player/stage is a client/server based platform built on top of C++. It is widely used in mobile robotics applications, in particular with the Pioneer robots.

As a C++ library, it does not bring new abstractions in term of parallelism or event programming: the user must use threads and an event loop of his own. Flexibility is also limited by the choice of a specific programming language, but the client/server architecture enables in principe other language interfaces, with a question mark on *usability*.

There is no distributed component architecture in player/stage, except C++ built-in objects.

B. Microsoft Robotics Studio

The recent announcement by Microsoft to offer a universal robotic interface has been perceived as a very good sign of the maturity of the robotics industry. It represents a serious effort in the right direction for more advanced abstractions for robotics, supported by a major software actor capable of active promotion. It is however relying on a .NET architecture which needs Microsoft Windows either on the robot or on a remote computer, raising questions in terms of flexibility. Generally speaking, several users have reported that MRS remains relatively complex to master for the moment.

C. CORBA

CORBA, the distributed component architecture developed by OMG, provides much flexibility as it has been ported to many type of languages. CORBA has no particular abstractions available for parallelism, except the object level parallelism, and events are handled with a usual asynchronous object message passing mechanism.

One of the limitation of CORBA is that it is perceived as a complex solution and it lacks of a central coordination mechanism to interface the different CORBA objects. The same applies to RT-Middleware currently developed in Japan.

Note that since URBI allows to dynamically create methods on an existing object at runtime, it is possible to create a UObject that will connect to a CORBA or RT-Middleware object, read the IDL and create appropriate hooks as URBI object methods, to transparently interact with the remote object. Effectively, this allow to transparently access CORBA or RT-Middleware components, seen as regular objects in URBI.

D. Others

Many others platforms exist, like Tekkotsu [6], Marie [7], Orocos [11], Orca [12], ERSP [13] or Pyro [14]. Some are based on a C++ library approach and rely on a rather complex architecture. Some others, like Pyro, are notably more simple and based on python, but do not include native abstractions for parallelism or distributed objects at the moment.

While each of these platforms brings its own clear benefits, and do a great job in their specific domain, none of them at the moment fits altogether with all the four requirements that we have detailed.

VI. Conclusion

While there is still no clear leader in the field of universal robotics software platform, the number of candidates is increasing, like the number of robots. Reminding us of the beginning of the 80's and the Personal Computer revolution, it is likely that one major platform will stand out in the future. But unlike computer users in the 80's, robot specialists and end-users are today more knowledgable and aware of the importance of design principles in order to have a successful, flexible and adapted platform. We have presented four key guidelines that should drive the development of such a platform: flexibility, modularity, parallel abstractions for robotics, and simplicity. We have claimed that each of these characteristics is necessary and should be acceptable and usable to make it suitable in a universal platform. Finally, we have introduced the URBI platform which is the result of our efforts towards such a universal platform. We also summarized briefly some others candidates by comparing their approach to the four criteria that we have stated.

URBI is currently available for free on www.urbiforge.org, with documentation and a user forum. It is available in particular for the Aibo and Mindstorm robots. We hope that URBI can contribute to the development of robotics in research and in the industry.

References

[1] J.C. Baillie, "URBI: Towards a Universal Robotic Low-Level Programming Language", *in Proceedings of IROS'05*, 2005, pp 820-825.

[2] J.C. Baillie, "The URBI Tutorial" *www.urbiforge.org/tutorial*, v 1.2, 2007.

[3] urbiforge, "URBI Technology website, with documentation and software exchange platform " *www.urbiforge.org*, 2004-2007.

[4] G.M.Biggs and B.A.MacDonald, "Specifying Robot Reactivity in Procedural Languages", *in Proc. IEEE/RSJ Int. Conference on Intelligent Robots and Systems*, 2006.

[5] G.M.Biggs and B.A.MacDonald, "A survey of robot programming systems", *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, Brisbane, 2005.

[6] David S. Touretzky and Ethan J. Tira-Thompson, "Tekkotsu: A framework for AIBO cognitive robotics", *in Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005, Menlo Park.

[7] Ct, C., Brosseau, Y., Ltourneau, D., Raevsky, C., Michaud, F., "Robotic Software Integration Using MARIE", *International Journal of Advanced Robotic Systems - Special Issue on Software Development and Integration in Robotics*. Vol.3, No.1, 55-60.

[8] 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 Robot Systems*, 2003, pp 2121-2427.

[9] UNECE - United Nations Economic Commission for Europe, "2004 World Robotics Survey", *www.unece.org/press/pr2004/04robots_index.htm*, 2004.

[10] Sony Corporation, "Open-R SDK for Aibo robots", *www.openr.aibo.com*, 2005.

[11] OROCOS, "Open Robot Control Software", www.orocos.org.

[12] A. Makarenko, A. Brooks, T. Kaupp, "Orca: Components for Robotics". *IEEE/RSJ International Conference on Intelligent Robots and Systems*, (IROS 2006) Workshop on Robotic Standardization

[13] Evolution Robotics, "ERSP: http://www.evolution.com/products/ersp/"

[14] Blank, D.S., Kumar, D., Meeden L., and Yanco, H., "Pyro: A Python-based Versatile Programming Environment for Teaching Robotics.", *Journal of Educational Resources in Computing (JERIC)*.

[15] A. Mallet, S. Fleury and H. Bruyninckx, "A specification of generic robotics software components: future evolutions of GenoM in the Orocos context", *in International Conference on Intelligent Robotics and Systems*, Lausanne (Switzerland), 2002.

[16] John K. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)", *Talk at the 1996 USENIX Technical Conference*, January 25, 1996, http://home.pacbell.net/ouster/threads.pdf