Software architecture for an exploration robot based on Urbi

Jean-Christophe Baillie[1] Akim Demaille[1] Guillaume Duceux[1,2] David Filliat[2]
Quentin Hocquet[1] Matthieu Nottale[1]

[1]Gostai S.A.S., 15 rue Jean-Baptiste Berlier, 75013 Paris.
[2]ENSTA ParisTech, Unité Électronique et Informatique, 32 boulevard Victor,75739 Paris Cedex 15.

May 10, 2011

Abstract

We present the software architecture of a mobile robot whose goal is to autonomously explore an unknown indoor environment and to build a semantic map containing high-level information. This robot was developed under the Panoramic and Active Camera for Object Mapping (PACOM) project whose goal is to participate in the French exploration and mapping contest CAROTTE. The software architecture puts together a large set of software components that deals with mapping, planning, object recognition and exploration. This architecture is based on the Urbi framework. We present the main functionalities of Urbi that were used for this project and show in detail how these capabilities make it possible to integrate all these modules.

1 Introduction

The Panoramic and Active Camera for Object Mapping (PACOM) project addresses the understanding of how an autonomous embodied system can build and extract information from sensory and sensory-motor data and generates plans and actions to explore and navigate in typical indoor environmental settings. In particular, we seek to extract high-level semantic information that is easy to understand and interesting to the robot users such as surrounding objects and the environment structure. To achieve this goal, the system requires different sensing modalities and also needs to act in order to improve its understanding of the environmental situation or to disambiguate its interpretation. This project is conducted by three partners: ENSTA ParisTech, UPMC (ISIR) and Gostai.

The project goal is to participate in the CAROTTE challenge organized by the Délégation Générale pour l'Armement (DGA) and by the Agence Nationale pour la Recherche (ANR). This challenge proceeds over three years with an increase in the difficulty over the years. The competition between 5 selected teams takes place in an arena of approximately 100 m^2 where objects are laid. The environment contains several rooms, typically 10 or more, with variable grounds and various difficulties (fitted carpet, tiling, grid, sand, stones. . .). Several kinds of objects are present, either isolated or gathered, in multiple specimens, which must be detected, located, and identified or characterized by the robot. Examples of the objects used in the competition are: boxes, paper-boards, journals, books, telephones, keys, bottles, plants, cameras, radios, robots. The complete description of the challenge can be found on the CAROTTE website[1].

Developing a robot to achieve such a challenge requires the integration of many software functionalities such as mapping, planning, obstacle avoidance, object detection and exploration. In our project, we used the Urbi middleware for this integration which makes it possible to easily implement parallel and distributed processing on multiple computers. We also used the urbiscript language in order to describe the robot behavior.

This paper will first detail the hardware architecture of our robot and describe the software components we have used for this competition. Section 3 then presents some functionalities of Urbi and Section 4 describes how this functionalities were used in our robot. Section 6 presents new functionalities of Urbi that we will use for the next version of our robot.

2 System Overview

We developed a robot (Figure 1) based on a Pioneer P3-DX from Mobile Robots Inc. The robot was fitted with 2 scanning laser range finders (one horizontal SICK LMS 200 and one vertical Hokuyo UTM 30 LX), a ring of sonar sensors, a Pan-Tilt-Zoom camera and three on-board computers. Our robot has not been optimized for the competition only, as it can be used for assistive applications for which the functionalities of semantic mapping are interesting.

Several software components were developed for the competition. Each component is a UObject (see Section 3), written in C++. These components support hardware access, 2D mapping, path planning, servoing, data logging, object recognition, exploration and semantic mapping. Only a short description of these components is provided here. See (Jebari et al., 2011) for details.

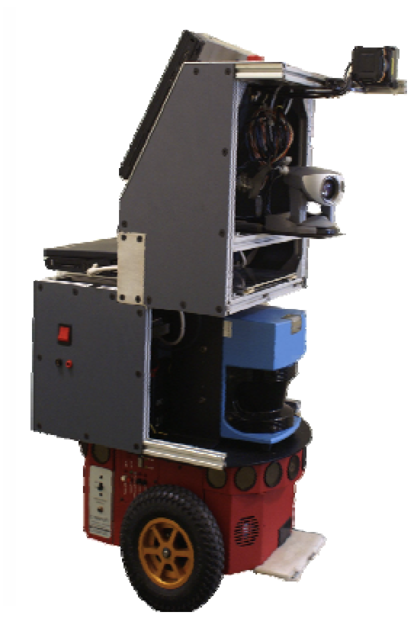
Figure 1: The modified Pioneer P3-DX robot used for the CAROTTE competition.

2.1 Hardware Access

A set of components support access to the robot hardware. One component is dedicated to communication with the micro-controller that controls the platform speed and the reception of data from the embedded sensors (sonar, SICK laser, odometry, battery level). A second component is dedicated to accessing the image and controlling the direction of the pan-tilt camera. Finally, a third UObject controls the Hokuyo laser sensor.

2.2 2D Mapping

2D SLAM is performed using the horizontal laser scanner and the Karto software library[2], which provides good performances and robustness in indoor environments. This library uses

[1]CAROTTE — http://www.defi-carotte.fr.
[2]Karto SDK — http://www.kartorobotics.com.

scan matching to correct the robot odometry drift and provides a 2D occupancy grid map of the environment upon request.

2.3 Path Planning

This module performs global path planning to reach any goal point given the current occupancy grid map of the environment. It also carries out obstacle avoidance of dynamic objects by locally modifying this global path taking into account lasers and sonars data. This module is also based on the Karto library.

2.4 Servoing

This UObject is responsible from converting the path computed by the Path planning module into speed commands for the platform using a simple PID controller.

2.5 Data Logging

Data Logging records data during the mission for offline mission replay and delayed processing. In particular, this module records images taken by the camera along with the associated robot position and camera orientation. This make it possible to detect objects in images and add them to the semantic map asynchronously.

2.6 Object Recognition

The object recognition module detects the objects in the images and estimates their location in the camera reference frame. Object detection is performed using a two-step method. The first step includes a fast extraction of the salient regions (Butko et al., 2008) in each image in order to segment the image to allow multiple object detection. This also improves detection and localization speed by reducing the processed area. The second step is based on local features extraction and on a bag of visual words approach (Nister and Stewenius, 2006) to detect if an object is present in each region. Two different feature detectors are used: SURF (Bay et al., 2006) in order to detect textured objects, and local color histograms (Filliat, 2007) in order to detect texture-less, colored objects.

Once an object is detected, its distance from the camera is estimated using the hypothesis that it lies on the floor if it has been detected using color information. If it has been detected using SURF features, the scale change between the detected object and the reference image is computed to estimate the object distance. Objects to be detected are known beforehand and a database has been built containing each object over different points of view and different known distances.

2.7 Exploration

In our context, exploration is required to completely map the environment using the horizontal laser scanner and to search for objects detected using the pan-tilt camera. We took advantage of the fact that the two sensors have similar fields of view (a semi-circle in front of the robot) to integrate these two objectives into a single algorithm.

We use a stochastic sampling strategy inspired by the Randomized Art-Gallery Algorithm (González-Banos, 2001) to search for the next robot position that discovers the most unseen area. Each sampled position is evaluated using a scoring function that takes visual object search and laser 2D mapping into account. For object search, a score S_{obj} is computed as the size of the area visible through the camera that has not been observed yet. For mapping, a score S_{map} is computed as the number of frontier cells between free and unknown area (Yamauchi, 1997) that are visible through the laser sensor from the position. Finally, a score S_{dist} is computed as the opposite of the travel distance from the current position. The final score is a weighted sum of these components:

$$S = \lambda_{obj} S_{obj} + \lambda_{map} S_{map} + \lambda_{dist} S_{dist}$$

The sampled position with the highest score is returned as the next robot position.

2.8 Semantic Mapping

This module adds information to the 2D map such as the 3D structure of the environment, the position, name and images of the detected objects and the detected rooms. The 3D structure is built as a point cloud using the Hokuyo laser sensor, assuming that the 2D mapping module gives a perfect position. We estimate the object position by integrating multiple detections using a Kalman filter for each object (Smith and Cheeseman, 1986). For each object, we can therefore estimate an uncertainty ellipse associated to its position. Rooms are detected in the occupancy grid through an algorithm that detects doors and analyzes the resulting connected components of open space.

3 Urbi as a middleware

Urbi is a software platform for the development of portable robotic applications (Baillie et al., 2008). To cope with the diversity of hardware components in robots and to provide extensibility, Urbi relies on the UObject architecture: an Application Program Interface (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 middleware architecture, low-level or high-level components can communicate simply. Existing C++ and Java libraries can easily be bound in Urbi, to take advantage of its features such as event-driven programming, timers and variable change notifications.

3.1 Events

Devoted to robotics, Urbi provides the user with a natural means to specify reactive behavior: *events* (Baillie et al., 2010a). Event-based programming is typically used for "background jobs that monitor some conditions". Commonly used in Graphical User Interface (GUI) environments to implement reactions to user input, they can be used similarly to program the human-robot interfaces ("do this when the head button is pushed"). They enforce separation of concerns: the monitoring and handling of exceptional situations (collision avoidance, battery level. . .) are not entangled into the regular code (path planning, movement. . .).

Events in Urbi are related to "publish/subscribe" architectures, such as ROS Topics (Quigley et al., 2009). Since they can be sustained, they are also comparable to "signals" in other environments, such as the Esterel programming language (Berry and Gonthier, 1992).

3.2 Plugged UObjects

Urbi is an object-oriented platform that relies on the "instantiation" of components metaphor. The natural way to bind native components (written in C++ or Java for instance) is to wrap them in objects (*UObjects*) that aggregate functions, and attributes (*UVars*). These UObjects are compiled as shared libraries that can be loaded on demand from the Urbi runtime. UObject can then be instantiated as needed.

Imported this way, the UObjects share the same address space as the Urbi runtime; in other words, they are truly part of Urbi as if they had been compiled together. This provides shared-memory and maximal efficiency at the expense of responsiveness: the bound functions can keep the CPU and starve the other components. For CPU intensive functions, Urbi features an alternative way to bind them so that they will be run in separate threads. Locking is taken care of by Urbi based on the user specifications ("only one function per object at a time", or "only one activation of that function at a time", etc.), or left completely to the user if requested.

UObjects can invoke each other. Urbi is then in charge of serializing and transporting the data from one to another. This is performed via *UVars*.

3.3 UVar

The UVars are sort of attributes exposed to Urbi. They serve as incoming/outgoing interface between a component and Urbi. Serialization is performed transparently: basic integer/floating types, strings, lists, arrays, maps, sounds, images, binary blobs, and compositions of them.

UVars also benefit from the events: one can request to be notified on UVar accesses (read and/or write), and hook functions to these events. This is the typical way data-flow processing is implemented in Urbi.

3.4 Remote UObjects

Urbi aims at supporting a wide range of platforms, from cheap devices up to high-end humanoid robots. While the latter typically feature a powerful CPU, the former can hardly run any significant scientific computations. In that case UObjects can be run on local or even remote computers. No specific recompilation needs to be performed: the very same shared-libraries are used, leaving to Urbi the details of data serialization/deserialization, and event notification propagation over the network.

Depending on the available features of the platform, the communication may be either on top of TCP, or RTP (Real-time Transport Protocol, RFC 3550), which is better suited for large streams such as audio/video.

It should also be emphasized that, since remote UObjects are run in separate processes (they are not embedded in Urbi, rather they are different programs communicating via the network), they provide a natural protection against serious bugs. Suppose for instance that a UObject infringes on its memory space (the infamous "segmentation violation"): the Operating System (OS) will then kill the culprit. If the UObject is plugged, the OS also stops the Urbi runtime whereas in remote mode, it would halt only the UObject process.

3.5 urbiscript

In addition to the UObject middleware, Urbi also features urbiscript, a domain specific scripting language featuring concurrency and event-based programming (Baillie et al., 2010a). The urbiscript language is object-oriented, and UObjects appear as regular objects: functions and UVars can be called, read, and changed from urbiscript and/or C++, Urbi taking care of possibly needed conversions, notifications, serializations etc. The Urbi runtime is a server: with a simple telnet connection users are offered an interactive urbiscript session to interact with the robot.

Since this paper focuses on the UObject architecture, readers interested by urbiscript are referred to the documentation (Gostai, 2011).

4 Practical Implementation

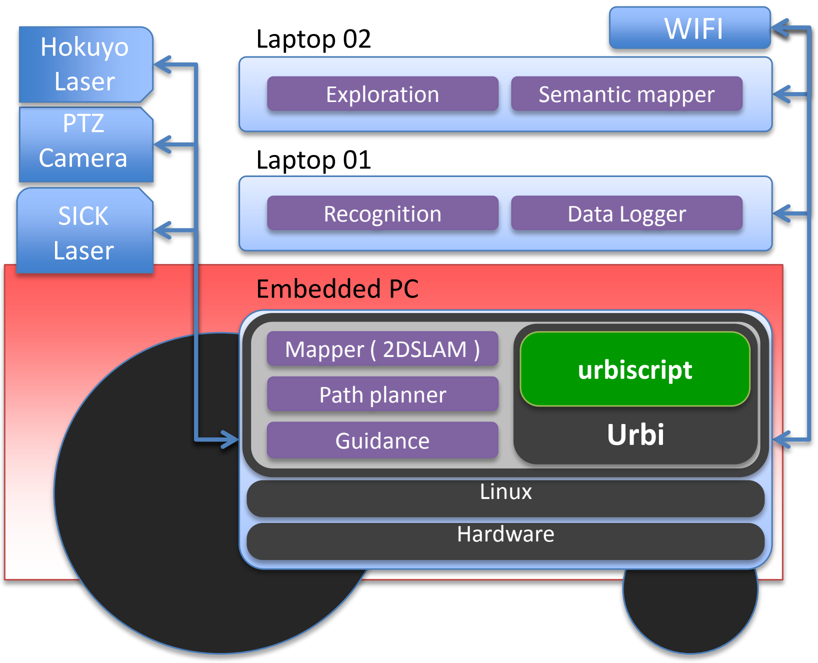

Figure 2: The software components distribution in our robot.

For this competition, several people including PhD students and interns were involved in the development of the various software components. As we wanted to avoid putting too much emphasis on embedded software optimization, and leave more time for research on new solutions, we chose to put an oversized processing power on the robot, in order to be able to run algorithms that would not fit on a single processor embedded in the platform. Hence, our robot is fitted with one embedded PC and two additional standard laptops linked together by an Ethernet network. Urbi made it possible to benefit easily from parallelism across these computers and to integrate almost transparently the software components that were distributed over the computers (Figure 2).

As we've seen before, the main software components of our robot are implemented in C++ components (UObjects). The UObjects running on the embedded PC were used as plugged components (Section 3.2), those running on the laptops were used as remote components using TCP connections (Section 3.4).

The global behavior of the robot is implemented in urbiscript that provides several way to orchestrate those components. Describing the full script controlling the robot is outside the scope of this paper, but the next section describes the main urbiscript features that are used in the control architecture of the PACOM robot.

4.1 Data-flow for SLAM

The data flow supporting SLAM, obstacle avoidance and path following is handled with the "notify change" Urbi feature. Periodically, the UObject responsible for interfacing the robot controller updates the different sensors readings and motor commands in UVars. The different components relying on those values are automatically notified of the changes through this feature. When notified, those components can execute a callback function, possibly in a separate thread. For instance, the SLAM module will receive odometry and laser readings from the robot controller. After update, the SLAM will provide the corrected position of the robot in another UVar whose update can launch other components such as the path following UObject. As the UVars are thread safe, this corrected position can be sent to other modules in different threads of the main Urbi server or to modules on different computers at the same time.

The code below shows an example of how we use the "notify change" feature. Here `servoing` is the UObject that computes the translation and rotation speed of the robot to follow a prescribed trajectory and `robot` is the UObject that interfaces the robot micro-controller.

```
servoing.&speedDeltaCompute.notifyChange(
  closure()
  {
    robot.speedDelta = servoing.speedDeltaCompute
  }
);
```

The code `servoing.&speedDeltaCompute` is used to get the *slot* (i.e., the "attribute") `speedDeltaCompute` of the `servoing` UObject, because calling `servoing.speedDeltaCompute` would return the actual *value* contained in the slot. The method `notifyChange(`*`function`*`)` of an UVar registers *`function`* and call it each time the UVar is updated.

The function called by `notifyChange`:

```
closure()
{
  robot.speedDelta = servoing.speedDeltaCompute;
};
```

will simply copy the value of `speedDeltaCompute` to the `speedDelta` UVar of `robot`. As the `robot` UObject mirrors its values periodically to the micro-controller, it will send the value contained in `speedDelta` to the micro-controller, which in turn will control the motor speed. The `closure` keyword means that the function can access variables that are outside of its scope.

4.2 Events for Mission Control

The mission of our robot is implemented in order to react to different situations that are transmitted through Urbi events. For example, when a target is reached, which is detected by the path follower, this module emits an event caught in urbiscript, and the behavior programmed for this event is triggered, like searching objects with the camera while in parallel computing the next exploration position with the exploration module.

Another example is when the path planner fails to find a path to reach the target, it will emit a `targetUnreachable` event that will trigger the computation of a new exploration goal. The code below shows how events are created in urbiscript:

```
var Global.targetUnreachable = Event.new();
```

Exploiting such an event is possible through the use of the `at` construct which will launch some code each time the event is triggered. Here is an example used in our robot:

```
at (targetUnreachable?())
{
  echo("target unreachable!");
  Global.reached = false;
  driveAuto.freeze;
  Global.robotIsDriving = false;
};
```

The purpose of the code is to stop the robot. `driveAuto` is a *tag* that stops the execution of `servoing` (see Section 4.3 for an explanation of tag use). As soon as the robot is stopped, another part of the urbiscript code will make the `exploration` module to search for another target before resuming the robot control.

In urbiscript, the operator `?` is used to catch events (see the example above); to emit an event use `!`:

```
targetUnreachable!();
```

To be caught by the previous code, the number of arguments for the emitted event has to match the number of arguments of the catcher function. Through not used in this example, any type of value can be used as event arguments. It is also possible to add a notion of time in the event. The following example shows how to emit an event with two parameters during two seconds, and how to catch an event that has been sustained at least during one second.

```
targetUnreachable!(val1, val2) ~ 2s;
at (targetUnreachable?(var arg1, var arg2) ~ 1s)
  // ...
```

4.3 Data Logging and Parallelism

Different ways to handle parallelism exist in the Urbi framework. In our project, we mostly used the remote UObjects to achieve that. For example, the object recognition UObject requires a lot of computation time, and processing each image when it is acquired would force the robot to move very slowly. Therefore a data logging mechanism has been implemented that collects data for this module so that it can process images asynchronously and in parallel when more computing power becomes available.

An easy way to do so in urbiscript is to bufferize the values needed by the `objReco` module in a list. During movement for example, an image and the corresponding robot position is put in this list every $100ms$. In parallel, whenever this list is not empty, the `objReco` module is launched with the last pair image/position and processes it in background. As processing an image takes more than $100ms$, images will accumulate during robot movement, while the list will be emptied when the robot is stopped. A protection from image stack overflow is implemented by preventing image accumulation if the image list size is too large (15 images in our implementation).

```
var buffer = [];
var imageRecord = Tag.new();

imageRecord : every(100ms)
{
  buffer.insertFront([camera.val,robot.position]);
  // In case of congestion, keep only the most recent images.
  if (buffer.size > 15)
    buffer.removeBack();
},

whenever (!buffer.empty)
{
  objReco.process(buffer.removeBack);
},
```

Image recording is started and stopped using *tags*, the job-control feature of urbiscript (Baillie et al., 2010b). The command `imageRecord.unfreeze()` will start image recording, while `imageRecord.freeze()` will stop it.

4.4 Failure protection

Under the pressure of the final developments, some modules contained errors in their C++ code that can occasionally lead to segmentation faults and that we did not have time to correct before the competition. In order to be able to perform the whole mission, these modules were used as remote UObjects so that they don't shutdown the entire system when they fail. A small urbiscript code is responsible for checking if the UObject is still working, and recreating it if necessary.

For example, the uobject `ObjReco` is instantiated in a variable `objReco` as a slot of `Global`:

```
var Global.objReco = uobjects.ObjReco.new();
```

The failure protection was made using the `every` feature:

```
every (1s)
{
  if (!Global.hasSlot("objReco") && uobjects.hasSlot("ObjReco"))
    Global.objReco = uobjects.ObjReco.new();
},
```

This code checks every second if `objReco` is still running. If eventually the UObject fails and is killed by the OS, `Global.hasSlot` will return false and `objReco` will be re-created in the urbiscript.

Beside this urbiscript code, a shell script '`objReco_launch.sh`' is used in order to reconnect the remote uobject automatically in case of segmentation fault:

```
#! /bin/sh
while true
do
  urbi-launch --remote ObjReco.so --host robot
done
```

While this solution only works for stateless UObjects such as our object recognition module, it is a good way to prevent complete failure even with some unreliable components.

5 Experimental Results

While this is not the main subject of this paper, Figure 3 shows an example of an exploration trajectory and of a map produced by the system.

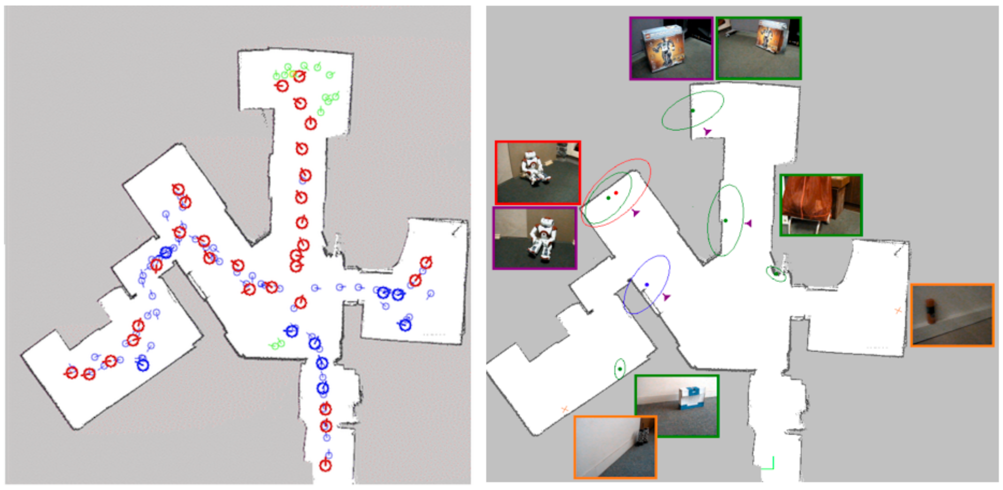

Figure 3: Left: exploration trajectory performed by the robot. Right: Example map created. Green color corresponds a correct object detection with a correct identity. Red color refers to a false detection and orange color refers to missed objects.

6 Future Developments

Under user request, newest releases of Urbi provide significant improvements over the version used for the competition. For instance, when binding a function with thread management by Urbi, it is now possible to specify congestion control: queue all the incoming invocations, or keep only one active and drop the forthcoming ones etc.

As exemplified by this paper, before being plugged in the system, UObjects can be put in quarantine and be run as separate process, to protect the whole system from their failures. It would be useful that Urbi detect such a situation and relaunches the components.

Urbi is part of the PROTEUS effort ("*Plateforme pour la Robotique Organisant les Transferts Entre Utilisateurs et Scientifiques*") (Martinet and Patin, 2008). Amongst improvements brought by this effort is an alternative way to implement data-flows in Urbi, in a more extrusive way, from urbiscript. In addition to providing support to register notification handlers on UVars, Urbi now provides *input ports*. In this approach, inputs of a component are declared as local input ports, and the binding between these input ports and the output of another component is done in urbiscript using the >> operator between two UVars. The following urbiscript code would be used to initialize an ObjectTracker given a camera:

```
var tracker = ObjectTracker.new();
camera.&val >> tracker.&input;
```

Regarding the PACOM project, the robot mechanical structure is currently optimized to better adjust to the new competition constraints such as slopes on the ground. Several software components are also developed further to enable mapping of environments containing glasses and mirrors and to enhance object recognition and environment classification by fusing range and visual information.

7 Conclusion

Urbi and urbiscript are powerful tools that make event-based programming and data-flow control very easy to use. They make it possible to gain a lot of time for development and to orchestrate all the work that have been done by many different people.

Furthermore, the hardware abstraction provided by Urbi also makes possible to easily test components in simulation in order to facilitate developments.

The features presented in this paper are only part of the many Urbi functionalities. For a more in-depth review of Urbi, readers are referred to the complete Urbi SDK documentation (Gostai, 2011) available on Gostai web site.

8 Acknowledgment

The work presented in this paper was financed by the PACOM project funded by ANR under the grant 2009 CORD 102.

References

Baillie, J.-C., Demaille, A., Hocquet, Q., and Nottale, M. (2010a). Events! (reactivity in urbiscript). In Stinckwich, S., editor, *First International Workshop on Domain-Specific Languages and models for ROBotic systems*.

Baillie, J.-C., Demaille, A., Hocquet, Q., and Nottale, M. (2010b). Tag: Job control in urbiscript. In Bouraqadi, N., editor, *Fifth National Conference on Control Architecture of Robots*.

Baillie, J.-C., Demaille, A., Hocquet, Q., Nottale, M., and Tardieu, S. (2008). The Urbi universal platform for robotics. In Noda, I., editor, *First International Workshop on Standards and Common Platform for Robotics.*

Bay, H., Tuytelaars, T., and Gool, L. (2006). SURF: Speeded up robust features. In *9th European Conf on Computer Vision.*

Berry, G. and Gonthier, G. (1992). The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152.

Butko, N. J., Zhang, L., Cottrell, G. W., and Movellan, J. R. (2008). Visual saliency model for robot cameras. In *Proceedings of the IEEE International Conference on Robotics and Automation, ICRA*, pages 2398–2403.

Filliat, D. (2007). A visual bag of words method for interactive qualitative localization and mapping. In *Proceedings of the International Conference on Robotics and Automation (ICRA).*

González-Banos, H. (2001). A randomized art-gallery algorithm for sensor placement. In *In Proc. 17th Annu. ACM Sympos. Comput. Geom*, pages 232–240.

Gostai (2011). Urbi SDK 2.7.1 manual. http://www.gostai.com/downloads/urbi/2.7.1/doc/urbi-sdk.htmldir/.

Jebari, I., Bazeille, S., Battesti, E., Tekaya, H., Klein, M., Tapus, A., Filliat, D., Meyer, C., Ieng, S.-H., Benosman, R., Cizeron, E., Mamanna, J.-C., and Pothier, B. (2011). Multi-sensor semantic mapping and exploration of indoor environments. In *Proceedings of the 3rd International Conference on Technologies for Practical Robot Applications (TePRA).*

Martinet, P. and Patin, B. (2008). PROTEUS: A platform to organize transfer inside french robotic community. In *Proceedings of the Third National Conference on Control Architectures of Robots, CAR 2008.*

Nister, D. and Stewenius, H. (2006). Scalable recognition with a vocabulary tree. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition - CVPR06.*

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software.*

Smith, R. C. and Cheeseman, P. (1986). On the representation and estimation of spatial uncertainly. *International Journal of Robotics Research*, 5.

Yamauchi, B. (1997). A frontier-based approach for autonomous exploration. In *Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation.*