Complementing Ada with Other Programming Languages

Samuel Tardieu
Institut Télécom/Télécom ParisTech
Networks and Computer Science Department
Paris, France
samuel.tardieu@telecom-paristech.fr

Alexis Polti
Institut Télécom/Télécom ParisTech
Electrical Engineering Department
Paris, France
alexis.polti@telecom-paristech.fr

ABSTRACT

This paper presents our experience in using Ada and the Ravenscar profile in a robotics non-profit association and in a robotics competition. While Ada is our primary and dominant language, we have complemented it with a hardware description language (Verilog) and an interactive language (Forth). We describe the interface between those languages, and the design choices that have been made to minimize the risks taken by leaving the Ada world. We also explain why we chose in some conditions to relax restrictions imposed by the use of the Ravenscar profile.

Categories and Subject Descriptors: D.3.2 [Programming Languages]: Language Classifications; D.4.7 [Operating Systems]: Organization and Design—*real-time systems and embedded systems, interactive systems*; D.2.11 [Softwarer Engineering]: Software Architectures—*languages (e.g., description, interconnection, definition).*

General Terms: Design, Languages.

Keywords: Robotics, FPGA, Ravenscar high integrity profile, Ada streams, hardware interface.

1. INTRODUCTION

Telecom Robotics is a non-profit organization founded in 2004 by passionate teachers, master students, and former students (now engineers) of the French Télécom ParisTech engineering school. Their common goal is to disseminate their knowledge and experience about robotics while themselves going on playing with robots.

To motivate people, the organization participates in the French robotics cup: every year, around 200 teams from France and nearby European countries gather in a small town and have to fight against each others in a tournament. Robots must be totally autonomous, and once the 90 seconds match has started, no interaction may take place between a robot and its team.

The rules change every year, from golf play to small building constructions from various pieces available around the contest table. Robots are not allowed to destroy each other during the one-to-one match and must implement reliable collision avoidance.

Shortly after our organization was created, it became obvious that some strategic choices needed to be made in order not to spread the work force between too many programming languages. Autonomous robots are inherently parallel, since they must at the same time work towards a goal and react to external events. Complex computations may need to be run in the background while the robot is moving or building structures, and asynchronous events such as a probable future collision must be promptly acknowledged and acted upon.

We have been using Ada at Télécom ParisTech for more than 15 years, both in teaching activities and research projects, including robotics related ones[6]. Other people also use Ada with great success in robotics projects or courses[9]. We approached AdaCore, a long-time partner of our institute, and they immediately offered to provide us with technical and financial assistance. At this time, some members who had never been exposed to Ada were reluctant to use this language perceived to be old-fashioned and extremely rigid, but others were able to convince them to at least try it before dismissing it.

Moreover, Télécom ParisTech had previously designed and built custom boards based around a Hitachi SH4 processor and a Altera Stratix FPGA (field-programmable gate array, a reprogrammable electronic component) for an unrelated research project. The boards, whose global architecture is shown in figure 1, could run a variant of the GNU/Linux operating system, an environment that most people were already familiar with. The GNAT Ada compiler maintained by AdaCore was already available for several Linux-based platforms, so it seemed to be a good potential combination.

This paper is articulated around the use of those boards in our robotics projects. After a short presentation of our development environment in section 2, we show how our Ada code has been organized and structured, as well as an example of how the robot is able to locate itself and its opponent in the game area (section 3). Then in section 4 we explain how we have used the Verilog hardware description language to offload some processing into the FPGA and interface it with Ada.

Workload distribution among several boards and communication strategies are detailed in section 5. Before concluding, we explain how we added interactive testing capabilities, and why we chose to relax some restrictions brought by the Ravenscar profile while on the field in section 6.

SIGAda'09, November 1–5, 2009, St. Petersburg, Florida, USA.

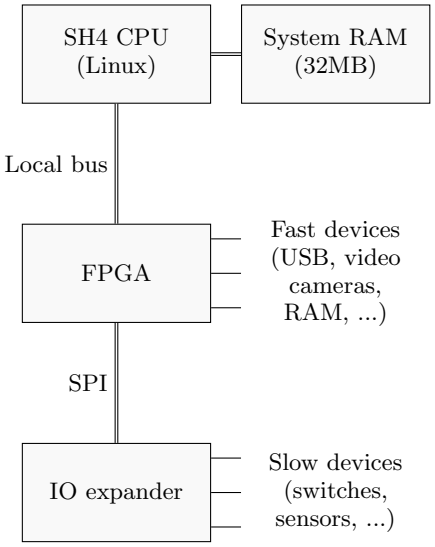

Figure 1: Board hardware architecture

2. DEVELOPMENT ENVIRONMENT

The first step was to port GNAT to the SH4 architecture running the GNU/Linux operating system. Unsurprisingly, this went very well and the port was completed and tested within a few days. The GCC back-end already knew how to generate assembly code for the SH4 architecture, and Linux targets are very similar to one another. This new GNAT port has since been integrated into the upstream GCC development tree and is now available in the recent GCC releases.

The next step was to select the most efficient way to teach Ada to newcomers or to members who had not taken any Ada classes during their studies. Since heavy parallelism was required and many people would be involved in developing co-dependent modules, we chose to restrict ourselves to the Ravenscar profile[3] while forbidding the use of dynamic heap allocation. We thought it would encourage good practices, such as:

- communication between tasks happen solely through protected objects, dramatically reducing the chance of a deadlock;
- protected objects may have a maximum of one entry, preventing them from being bloated with several functionalities;
- entry guards are limited to simple boolean tests, forcing developers to think about the best information to store in the private part of each protected object, and preventing them from synthesizing this information from more complex data;
- forbidding dynamic allocations clearly shows at link time what amount of memory will be needed at run time, encouraging developers to spare this scarce resource (32MB of RAM in our case).

Most software modules also need to work on regular computers. Since the robotics cup challenge changes every year, several months are necessary before we get a working mechanical platform. Algorithms need to be tested, as do the interactions between the modules. We turn on a large number of compiler warnings, and the code base must be warning-free on every platform. Some packages have been developed to simulate the robot behavior, and the use of GNAT project files allows each developer to compile the code either for the simulator using a native compiler or for the robot using the SH4 Ada cross-compiler.

The last constraint we decided on was to never exchange code directly between developers without going through the source code management system. We have settled on Mercurial, which is a fully distributed revision control system[18]. This has the advantage of letting developers commit their code locally first, with access to the full history, then synchronize the local repository with the central one. Also, while on the contest site, setting up a new central server on any of the developers' laptop is painless and can be done in no time.

3. PROGRAM ARCHITECTURE

As shown in figure 2, the code has been organized into well-isolated layers. This logical separation helped us reuse a maximum of code year after year. For example, the propulsion and guidance system is likely to always expose the same principles, even if we choose different wheels or encoders; its high-level interface provided by the devices interaction library will not change, unless we implement a totally new way of moving the robot, *e.g.,* by specifying Bezier curves control points instead of a succession of arcs.

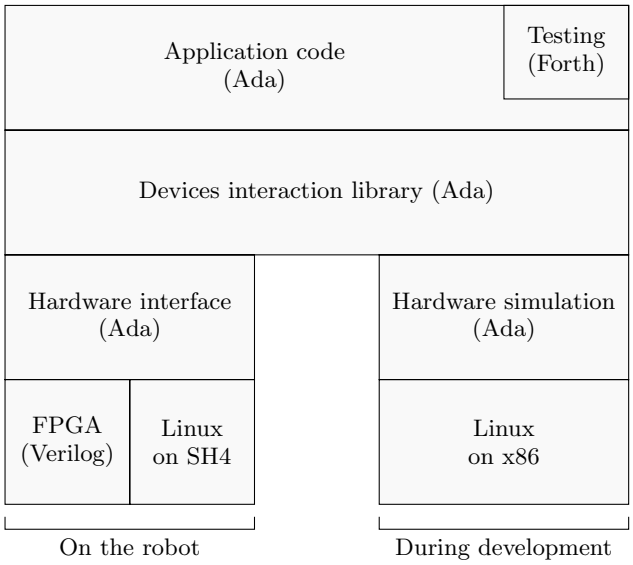

Figure 2: Robot code architecture and languages

3.1 Devices interaction library

The devices interaction library is in charge of providing the application programmer with high-level views of the various peripherals that can be used by the robot during the competition. For example, it may offer to set a robotic arm in a given position using a single subprogram call, while internally decomposing this operation into numerous submoves determined by complex inverse kinematics equations.

This layer exposes only a non-blocking interface to the application code. If an operation cannot be performed instan-

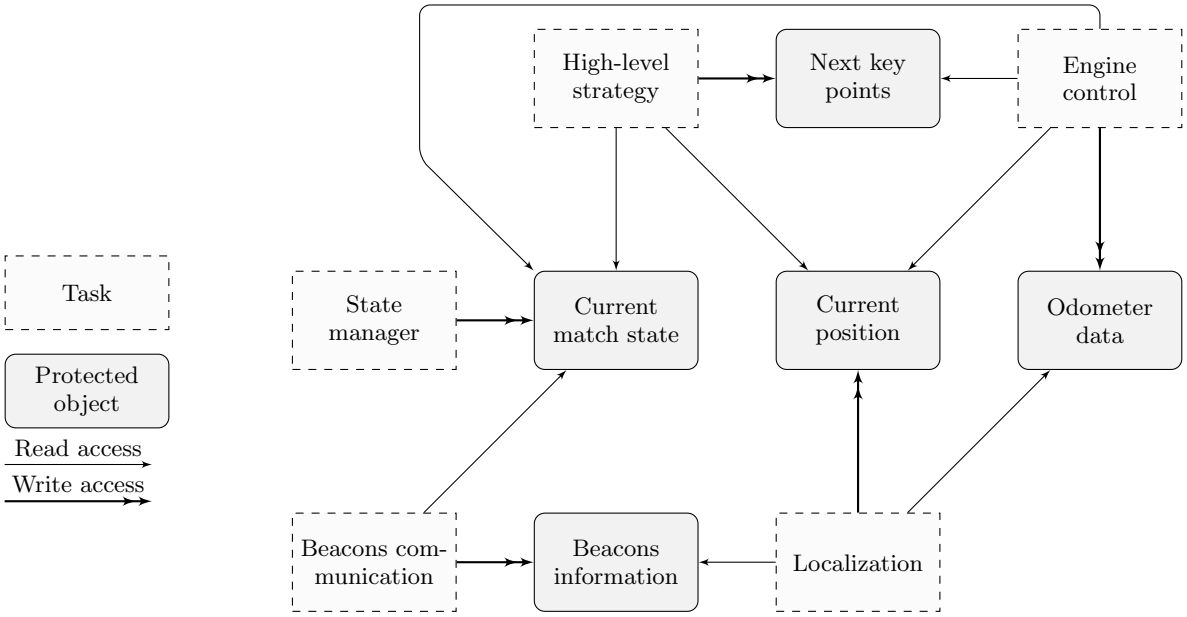

Figure 3: Communication through protected objects

taneously, it will be backgrounded using a dedicated task, and the caller will be able to later check the operation status if needed.

The devices interaction library uses the underlying hardware interface layer. This layer offers an access to the robot peripherals (detailed in section 4) and to the operating system services such as the file system and the network. During the development phase, an algorithm can be tested on a developer's machine by replacing the hardware interface by a hardware simulation environment.

3.2 Application code

As expected, the organization of the application code has been strongly influenced by the use of the Ravenscar profile. The robot intelligence has been split into non-terminating periodic tasks communicating through protected objects.

A simplified example of such communication is given in figure 3. The *Localization* task uses the odometer and beacons data to compute the current robot position and stores it into a protected object. Its inputs are themselves computed by other tasks, the *Engine control* and *Beacons communication* tasks, respectively.

You may notice that the *Localization* task does not need to read the current match state. Indeed, localization computations are always performed, independently of the match phase (preparation, match in progress, match terminated). However, a result will be made available to other entities only if the inputs are fresh enough to determine a credible robot position.

3.3 The state manager

Since all tasks are started during the elaboration and never terminate, they must know what state the robot is in before performing their duties. For example, there is no need to detect a possible collision with the other robot after the end of the match. Moreover, moving to avoid a collision once the match is officially over would provoke an immediate disqualification. Every task checks the current program state before acting, while a state manager takes care of updating it when needed (when the start switch is toggled, when a fatal error is reported, when the match duration has elapsed, etc.).

The competition rules are strict: once both teams have been called to the match table, a color (blue or red) is randomly assigned to each robot. The teams have a couple of minutes to install and configure their robot. Engines are allowed to be turned on if needed, switches may be toggled (*e.g.,* to tell the robot which color the pieces it should be seeking on the table will be) and computers may be connected, to launch the main program for example. Also, a mandatory starter string is installed on each robot: the string will be pulled to indicate the beginning of the match. This corresponds to the *Testing* state in figure 4.

After a referee notice, the teams are no longer allowed to touch the robots. Our robot enters the *Startup* state and immediately uses one of its video cameras to take a picture of the empty table and stores it for later use. A card is then randomly picked up from a deck containing all the possible match area configurations, and the game elements are accordingly placed onto the match table.

After a countdown, the match starts. Both team leaders must launch their robot by pulling on the starter string from a distance. We use a magnetic sensor to detect this action; once the magnet placed at the end of the string gets off the sensor, our robot enters the *In round* state and takes another picture of the match area. By comparing this picture and the previously stored one, it can easily and quickly determine the match configuration that has been installed.

Once 90 seconds have elapsed, the robots must stop and stay still; a moving robot would be immediately disqualified. Our robot enters the *Stopped* state, and can later optionally enter the *Vacuuming* state for a few seconds if asked to do so, a state in which it will release any game elements it still holds. This operation can be repeated if needed until all the items have been freed.

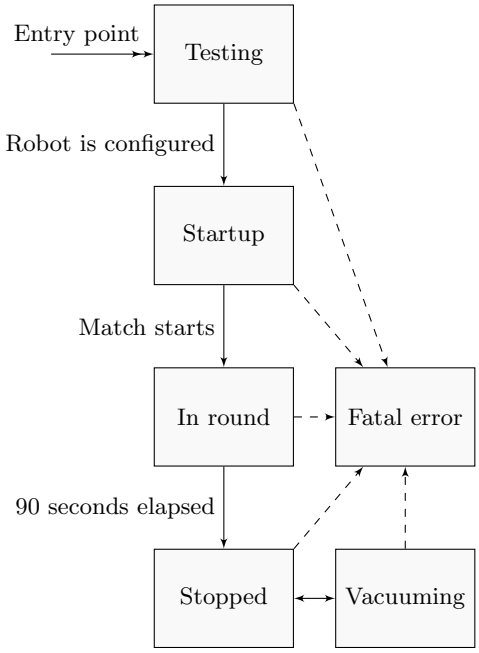

Figure 4: Possible program states

At any moment, any module may decide to put the robot into the *Fatal error* state. In this state, the robot will refuse to move any of its engines, which will be put in freewheel mode if possible. In particular, each robot must be equipped with a mandatory big red push button in case of emergency: in our code, a high priority periodic task checks the button and puts the robot in the *Fatal error* state when pushed, which is enough to prevent any harmful move.

3.4 Infrared localization system

According to the contest rules, collision avoidance is a strong requirement. A robot causing any damage to its opponent during a match may be disqualified.

In order to locate the other team's robot as well as ours, we use a triangulation algorithm with data obtained from beacons located at known positions on the match area[13]. Figure 5 illustrates it: on the match table (seen from above), B_1, B_2 and B_3 are our three fixed beacons, while R_1 and R_2 are the robots.

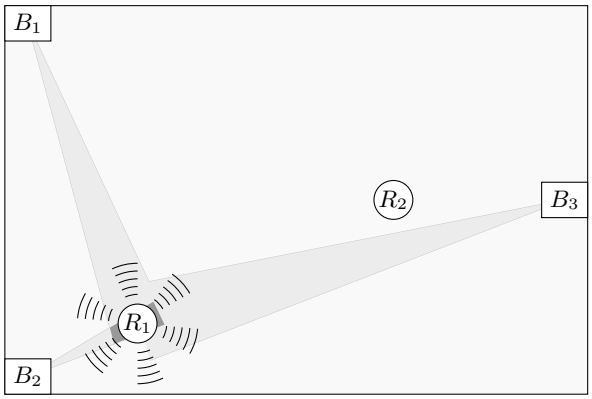

Figure 5: Robot localization using triangulation

Before the match, we place a small electronic device that we designed on top of each robot, as allowed by the contest rules. If the other team do not allow us to do so or chose not to build the mandatory pole and platform, they automatically loose the match by default. Each device contains a small battery powered processor, along with a Zigbee low-power wireless communication device[12] and a omnidirectional infrared emitter.

The three beacons are driven by similar electronic boards equipped with an infrared receiver mounted on a servo motor. They also communicate with our robot using Zigbee.

Our main program, also connected to a Zigbee device, directs one of the two emitters placed on top of the robots to continuously send modulated infrared signals until further notice. The signal is encoded in order to be able to distinguish it from the background infrared noise caused by other contestants and by the powerful lights of the television crews.

Each fixed beacon rotates its infrared receiver until it recognizes a proper signal. When it does, it sends its current servo motor angle to our main program through a Zigbee message. If we receive data from two beacons or more in a short time span, we can compute the possible position of the tracked mobile. In figure 5, the darkest area below the R_1 circle shows the computed possible positions when information coming from the three beacons is used.

Note that if the other robot obstructs the path between the emitter and one of the receivers as illustrated in figure 6, the position can still be computed unless the tracked mobile is located near the straight line going through the beacons. Of course, the precision will be decreased accordingly, as clearly shown by the greater surface of the dark area.

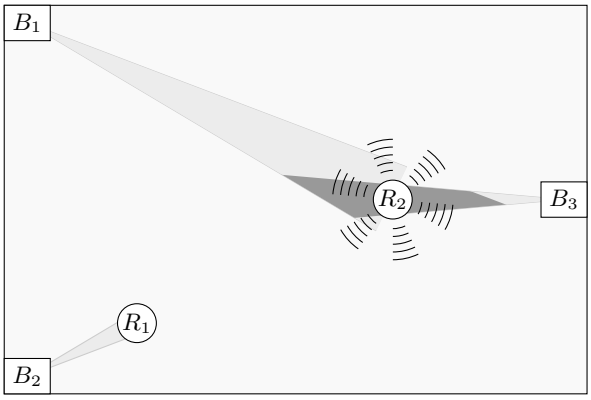

Figure 6: Obstacle in the way

Using this technique, we can locate either robot most of the time. The result is not very accurate, but is useful enough to avoid collisions with the other robot, and the errors do not accumulate over time. To locate our own robot, our odometer-based strategy proved much more precise; the relatively short match duration (90 seconds) does not allow the drift to become significant.

On our robot, the communication with the Zigbee device is done through a custom board built around a STM32 ARM processor running FreeRTOS[2]. This board, which also provides the robot with a LCD display used during the tests, is connected through a RS232 serial link implemented in the FPGA and driven directly from the Ada program address

space. In order to be able to easily extend the various messages exchanged between our main program and the STM32 board, a callback-based interface with registration capabilities is used.

4. HARDWARE CONTROL FROM ADA

Since the main CPU used in our robot does not have many available general purpose communication ports, external devices are physically connected to the FPGA which is, in turn, connected to the SH4 CPU using a regular memory bus interface. The FPGA appears to the SH4 as an external memory, and is controlled through regular read and write operations. The FPGA can also asynchronously signal events to the CPU (such as the completion of a long operation) by using dedicated interrupt lines.

Since unfortunately the FPGA itself has a limited number of general purpose inputs and outputs, slower devices are accessed through an IO expander chip connected to the FPGA by way of a serial peripheral interface (SPI) bus. This architecture is close to the one found on traditional computer motherboards: the FPGA plays the role of the northbridge, while the IO expander plays the role of the southbridge[16].

Every device connected to the FPGA is controlled through a few consecutive memory addresses mapped into the SH4 memory space. The FPGA recognizes accesses to those areas and acts accordingly by directing the read or write request to the appropriate hardware control submodule.

4.1 FPGA programming and Verilog

The FPGA located on our boards allows many hardware operations to take place in parallel; it is configured using the Verilog hardware description language[22]. Verilog, which is based on a C-like syntax, can be used to describe a hardware digital system as simple as a flip-flop or as complex as a microprocessor. It can represent either the structural (how it is built) or the behavioral (what it does) view of the described component. It is then possible to simulate the system, and to generate log files representing test results for behavior analysis and conformance verification. The component can also be synthesized from its Verilog description; the resulting netlist, corresponding to the physical description of the described hardware based on elementary logic gates, can be used to manufacture a chip, or, as is the case here, to program a FPGA.

A Verilog description uses a collection of modules similar to Ada generic packages. They have zero or more inputs, and zero or more outputs. Verilog modules need to be instantiated: their inputs and outputs are connected either to other instances, or to external circuit pads. It is common for modules to take a clock line as one of its inputs. This way, they can work synchronously on clock transition edges. In our case, we provide our modules with a 60MHz clock line derived from the 240MHz SH4 clock.

In order to interact with the outside world, we have described a memory bus in Verilog. All the modules in our FPGA in charge of a device are connected to this bus, as well as to the peripheral they control. Each module also takes a *bus enable* signal as input; another module, called a bus decoder, decodes the address given by the external component acting as a bus master (in our case the SH4 processor) and sets the *bus enable* of exactly one module if the bus read or write command is intended for this module. The module can then perform the requested operation and, depending on the direction of the command, look at or set the bus data lines, as illustrated in figure 7.

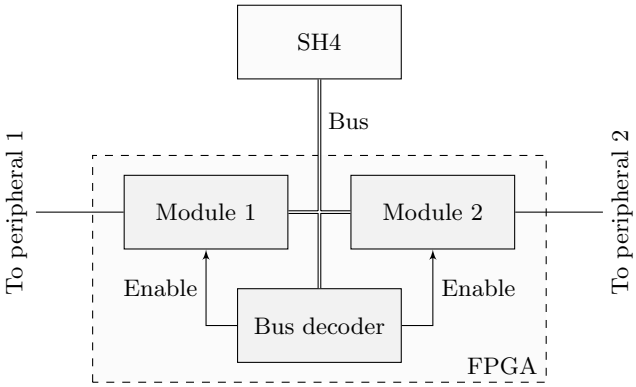

Figure 7: Multiplexing modules in the FPGA

Adding a new peripheral to the FPGA is easy: a new module is added to the system, and one more case is added into the bus decoder. Modules can have their own internal memory, although FPGA memory is a scarce resource. If they need more, they can also use an external RAM connected to the FPGA through an arbiter module; for example, a fast video RAM is made available to video-processing operations.

4.2 Interfacing the FPGA with Ada

The traditional way to interface an external device with user-land (as opposed to kernel-land) processes consists into writing a device driver[5] for the underlying operating system, and using the device driver from the desired programming language. The operating system is in charge of allowing or forbidding a process to access the device driver, while the driver itself must handle concurrency issues when multiple processes or threads attempt to issue commands to a device at the same time.

However, we chose to bypass this OS-centric view illustrated in figure 8 to adopt an Ada-centric view instead: since our devices are to be accessed only by an Ada program, writing the access and control protocol directly in Ada gives us more flexibility than writing a Linux device driver and an Ada interface to this device driver. Also, handling concurrent accesses using regular Ada concurrency tools such as protected objects and tasks lets us integrate parallel accesses to the device into the global Ravenscar-compliant scheduling.

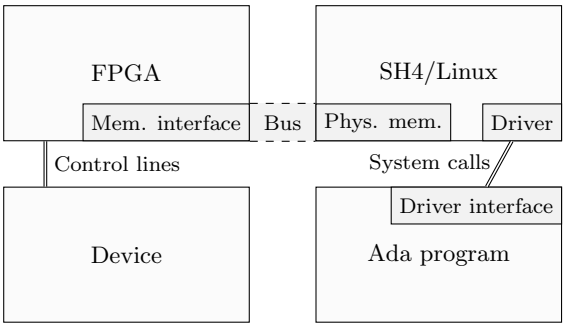

Figure 8: Using a traditional device driver

A minimalist Linux device driver allows the Ada program to map the physical FPGA address space into its own process virtual address space (using the `mmap` system call), as illustrated in figure 9. This mapping is obtained at elaboration time, and its base address is used in subsequent address representation clauses to exchange commands and data with the various devices.

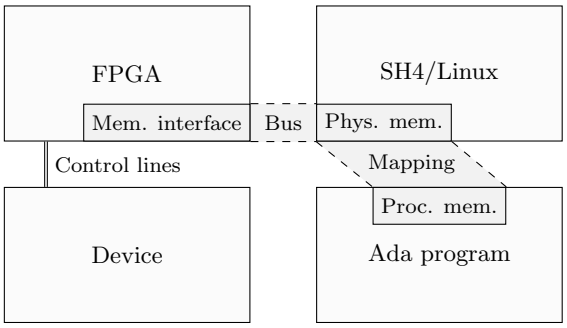

Figure 9: Implementing device control in Ada

Then each linked-in Ada interface package checks that the device it is in charge of is indeed supported by the current FPGA configuration by checking a device "magic number" and version information obtained from the FPGA. If anything goes wrong, the program aborts before finishing its elaboration.

In addition to giving us more control over the scheduling of concurrent accesses, this scheme brings us two other benefits:

- Context switches between the kernel and a process are very costly compared to regular memory read and write operations. Once the mapping between the physical address space and the virtual Ada process space has been established, no extra system call is needed to access the device as the address translations will be performed in hardware by the SH4 memory management unit (MMU).

- Communications between the Verilog developer for the FPGA and the Ada developer for the SH4 do not need to get through an extra layer, namely the Linux kernel driver developer. This is a very sensible topic during development sprints, where developers work together by experimenting various hardware/software splitting configurations.

Using this technique, we also moved parts of the propulsion control system into the FPGA, freeing the Ada program from the burden of counting the odometers ticks. Now, the main program directs the hardware to advance up to its target and gets immediate feedback about the current progress using a memory-mapped variable.

Also, rather than controlling the SPI interface from Ada to access the devices located behind the IO expander (see figure 1), we added custom Verilog modules into the FPGA to do the encoding and decoding of the information provided by the various devices transparently. The peripherals are presented as if they were directly connected to the FPGA, and they can be controlled from Ada as easily as any other device in the robot.

We have recently started developing a new tool to describe a device interface between the FPGA and the Ada world. This tool generates both the Ada hardware interface packages (declaration and body), and the Verilog module interface corresponding to this device. This will save us from writing many representation clauses to indicate which bit at which memory-mapped address corresponds to which hardware flag. Also, magic and version numbers consistency is automatically verified at elaboration time by the generated code.

5. THE NEED FOR PARTITIONING

Our initial idea was to use simple devices as inputs to the system, such as infrared telemeters, color sensors or bumpers. After a few weeks, the participants had written all the necessary modules as well as some core algorithms necessary for the robot to act on those sensors.

5.1 Early results

As teachers, we were very impressed with the outcomes of using Ravenscar as an Ada teaching tool: the restrictions brought by the profile actually made Ada easier to learn than with the full unrestricted language. The lack of asynchronous transfer of control or task termination led to a lot of small and clean periodic tasks, allowing developers to easily compute the expected response time in most situations.

More surprising to us was the fact that those restrictions forced the developers to separate functionalities into very independent modules. Adding a new functionality as an extension of an existing (and maybe unrelated) one would violate Ravenscar rules such as having a maximum of one task waiting on an entry queue at any given time.

Since everything was going so well, we became more ambitious and decided to use more complex sensors such as a video cameras. A physical interface for a tiny CCD camera was designed and built, and some Ada packages were written to detect shapes and colors in the image. Then a second video camera was added to the board. It became obvious that more cameras would lead to better environmental analysis, and we decided to use two main boards instead of one to double the processing power and the memory capacity.

5.2 Distributing the program

The communication between the boards had to satisfy a number of constraints:

1. Exchanging data between Ada programs running on both boards has to be easy and to preserve strong typing. Also, being able to replace one of the boards by the same program running on a developer's machine would be very useful during development.

2. Data exchanged between the boards will be composed of small real-time information about individual video frames or recent sensor readings. If a transmission error occurs, we would rather loose one piece of information rather than congest the link with retransmissions of outdated and useless data.

3. Although we need the link to be fast in order to get timely data, we do not want it to flood the main CPU with interrupts; however, we want to be told immediately when new data can be processed.

Constraint 1 made us use Ada streams. Although we have not been needing it in practice because x86 and SH4 processors share the same endianness and word size, we knew that cross-platform stream attributes were available in GNAT by replacing the `System.Stream_Attributes` package with the one coming with GLADE, GNAT's implementation of the Distributed Systems Annex[19].

Because of constraints 1 and 2, we chose UDP as the transport layer. Using IP ensures interoperability with the developers' computers, and UDP offers a packet-based mechanism without retransmission.

Constraints 2 and 3 made us develop a custom high-speed full-duplex synchronous serial link protocol called *twonet* on which PPP (point-to-point protocol) could be run to provide IP and UDP communication. This serial link can transmit data in both directions with speeds up to 15Mb/s, although we are currently using it at 1Mb/s. Also, *twonet* is packet-based, and will signal an interrupt to the main CPU only when a full packet (up to 256 bytes) has been correctly received and is ready to be retrieved. The overhead has been kept low at 3 bits per packet, so that the link can sustain a 99, 86% efficiency rate at full load. Two send buffers and two receive buffers in each direction ensure that packets can be sent back to back, since a packet can be prepared while the previous one is still being transmitted, and a received packet can be processed while the next one is being received.

We considered using the Distributed Systems Annex for high-level communications between the boards, but the implementations available for our version of GNAT were either not Ravenscar compatible or much too heavy for our limited environment. Instead we chose to implement a new stream type, inherited from `Ada.Streams.Root_Stream_Type`, augmented with a `Flush` operation. This lets us send whole data blocks once they are ready; symmetrically, if a packet is received by the other board, it knows that the packet is complete, correct, and ready to be processed. Also, we check the size of the outgoing data at flush time to ensure that it will fit into a *twonet* packet even after the subsequent UDP, IP and PPP encapsulations.

From a language point of view, only a single record type is ever used in each direction. But thanks to Ada discriminated records, this type can contain many different kind of data while preserving strong typing capabilities.

The choice of an IP based protocol makes it easy for a developer to interact with the embedded boards using a WiFi USB dongle. Moreover, by keying in the adequate routing table information, one board may act as a gateway (through PPP over the *twonet* serial link) for the other, thus allowing a single WiFi interface to be used to access both boards. Figure 10 summarizes the various layers used for communication.

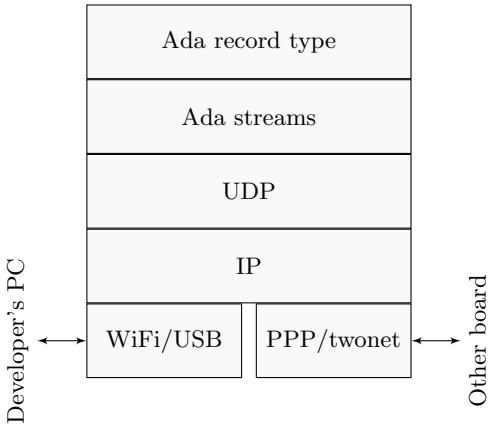

Figure 10: Protocol stack

5.3 Offloading costly operations

Four cameras require a lot of operations to process incoming data in real-time. To decrease the CPU load and the memory bandwidth, we decided to offload some picture analysis functionalities into the FPGA located on each board.

5.3.1 Hardware real-time image processing

The first image processing operation implemented in the FPGA is a picture down-sampling. The algorithms we use to locate key points or objects on the battlefield could not accommodate the original camera resolution. This step could easily be implemented in software, but it would require a lot of memory accesses to retrieve the image and store the down-sampled result. Doing it in the FPGA allows us to do it on the fly as individual pixels are sent by the video cameras, without ever storing the original pictures into the memory.

The second hardware-backed operation consists into tagging each pixel with its appropriate color within a predefined reduced set (*e.g.,* red, green, blue, yellow, black, and white). While this may look like an easy task, this information cannot be obtained in a reliable way from the pixel hue and saturation values alone. The use of cheap video cameras and the presence of different lightning environments make it a complex operation requiring a manual calibration.

To calibrate the pixel tagging system, the cameras take various pictures of the match environment. Those pictures are tagged using a dedicated program by a human operator: significant color areas are drawn using a mouse and identified. From this data, the program computes the best shapes corresponding to each identified color in a three-dimensional color space. The result is then uploaded into the FPGA every time the system starts, and each pixel is analyzed on the fly right after the down-sampling process. The resulting image is mapped directly into the Ada process space; reading a pixel value is as simple as getting an element from a global array.

For each video input, both image processing operations can be individually turned on or off depending on the needs. Other hardware algorithms such as straight lines detection using Hough transforms[17] have also been implemented and have been used in the past to help localize the robot using checker patterns drawn on the match table. Nowadays, our current localization system based on odometers and fixed position beacons is precise enough and requires less processing power and FPGA cell space.

5.3.2 Video overlay

Although video-related algorithms are first experimented on the developers' machine using test pictures, it is often useful to see how the algorithms behave on the real platform. In order to do so, we have added a video output interface to our boards. A PAL (European TV standard) composite signal is generated by the FPGA by combining the camera inputs and an overlay buffer.

This memory mapped overlay buffer is accessible from Ada and allows the main program to draw shapes on the screen on top of the acquired pictures. For example, it can be used to display the pixel colors computed by the pixel tagging system, to draw a line around the identified objects, or to display the path the robot plans to follow to reach its next key position.

As can be seen in figure 1, an additional memory is directly connected to the FPGA and is used to store both raw and processed pictures. Using the video output does not put any additional pressure on the SH4 memory bus since it is only used to transmit overlay drawings.

Additionally, it would be possible to add text on the screen by embedding some character fonts in the FPGA, or to integrate a general purpose shape drawing hardware library. However, we have not encountered such a need yet.

6. ON THE FIELD

During the year preceding the robotics cup, every team builds an environment similar to the one that will be used during the contest. This usually consists into a 2m×3m table equipped with various equipment and other game elements. Building the exact environment with the right materials would be very costly; wood and aluminum substitutes are frequently used instead. As a consequence, sensors calibrated in this environment will probably not work out of the box on the contest site and will need to be recalibrated.

For example, optical presence sensors may be affected by the difference in materials reflexivity, or by the extreme brightness caused by the lights of the television crews. Grippers may need to handle objects more firmly or more gently, or to pick them up from a slightly different angle. The first hours spent on the contest site are used by the teams to adjust their devices and make sure they function as expected in the final environment. Also, each team looks carefully at other robots and tries to guess and study their varopis strategies. For those reasons, we needed to be able to efficiently test existing features and add new ones in no time while on the field.

6.1 Interactive robot manipulation in Forth

When a new peripheral is added to the system, or when a new strategy is designed, it may be useful to be able to interactively test them with various parameters without modifying, rebuilding and rerunning the whole program. For this reason, we wanted to add an interpreter which would let us call various subprograms and manipulate their input and output values.

Forth is an interactive stack-based language with a very simple syntax[21], which can easily be extended into a domain-specific language (DSL). Forth is commonly used in robotics because of its conciseness, its small footprint[11, 10, 8], and its extensibility. The interpreter lets you define new *words* (the Forth name for *subprograms*) that can be used immediately after their definition and are indistinguishable from predefined words. If needed, user-defined subprograms can even assume total control over the parser and create a whole new syntax.

Reusing a well-known and modular language was more attractive than writing a specialized DSL interpreter[14] which would need to be modified continuously to accomodate the robot changes. This is why we chose to implement a Forth interpreter in Ada. We briefly considered reusing one of the many portable Forth interpreters[7], but those are usually written in C. We did not want to add yet another interface to maintain.

Forth uses a reverse Polish notation: words in the input lines are executed from left to right, and any non-existing token is assumed to be a number, parsed, and put onto the data stack. For example, the expression "`17 5 - 10 *`" will let 120 on the stack: 17 and 5 are put onto the stack, "`-`" takes two numbers from the stack, subtract them and put the result (12) on the stack, 10 is put on the stack, "`*`" takes two numbers from the stack, multiply them, and put the result on the stack.

The Forth interpreter is linked with the main Ada program and interacts with the robot devices. For example, closing the first of the two grippers can be performed by entering the "`1 close-gripper`" sentence interactively. The interpreter is compatible with the Ravenscar profile and uses a well-defined interface to call Ada subprograms, which are seen as primitives of the Forth kernel. It is also possible to manipulate the Forth stack from the Ada side, to extract arguments to pass to subprograms or to put results back.

Let us assume that to grab an object lying on the game area the robot needs to first open the gripper, then go forward so that the object enters the gripper, then close the gripper to pick up the object. We want to test interactively by how much the robot should move to get the best result. In order to avoid tedious repetitions, we can interactively define a new "`pickup`" word which will take a gripper number and a distance:

```
: pickup tuck open-gripper forward close-gripper ;
```

We can now test whether a 35 millimeters forward move works for the first gripper by issuing the "`35 1 pickup`" command (see figure 11 for an explanation of how it works internally). If it is obviously not enough, we can then attempt a 40 millimeters move with "`40 1 pickup`", then try with the second gripper with "`40 2 pickup`". With a single line (the "`pickup`" word definition), we have extended the domain-specific language with a new functionality.

6.2 Relaxation of Ravenscar restrictions

As teams study each others strategies, being able to adapt the robot behavior in a very short time is a crucial advantage. The contest lasts three days and two nights, and each team may have as little as one hour between successive rounds to design, code and test a new strategy.

This year, for the first time, we chose to relax the Ravenscar profile restrictions while on the field: new strategies could be written using the full Ada language, but the lower code layers could not be changed. Should a bug be discovered in the core routines, the fix would have to compile with the full Ravenscar restrictions set.

Relaxing those restrictions was not an easy decision to make, as it sacrifices the integrity of some parts of the software to obtain faster results. Afterwards, we are happy with this choice that proved very useful in practice; we were able to adapt to the next opponent supposed strategy between the rounds. The "`select ... then abort`" construct was used to try a strategy and immediately fall back to another one if the first one was not working or became inefficient because of an opponent unexpected move.

Many of constructs forbidden by Ravenscar, such as timeouts, may be implemented differently while using Ravenscar[4]. However, the time needed for doing so would have

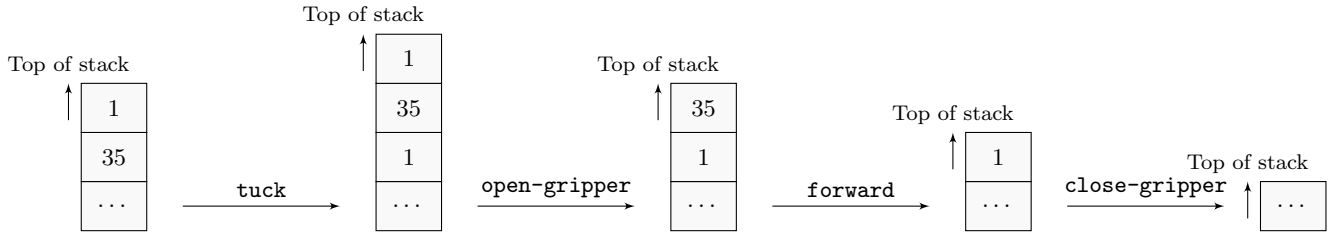

Figure 11: The inner working of "`35 1 pickup`"

prevented us from working on urgent strategy changes that could not be done without looking at the other teams behavior first.

7. CONCLUSION AND FUTURE WORK

The first conclusion is that our current languages combination works well for this kind of task, as our rank gets better every year. In 2009, about 200 teams have entered the competition, and we ended up in the 9th position. The points we lost during the matches were not due to faulty software; our lack of mechanical skills is badly showing, and we lost one decisive match because of a last-minute gripper mechanical problem which could not be fixed by the software alone.

Also, nobody in our organization seriously talks about replacing Ada by anything else. Using the Ravenscar profile during the whole development phase leads to very clean and well-defined software modules, some of them being reused painlessly year after year. For the last two years, we have not taught a single Ada class to newcomers; they learn Ada casually from old-timers as they go, and end up after one year of coding with very solid Ada skills. Most newcomers are freshmen, and this experience makes them choose Ada classes during their stay at Télécom ParisTech in order to reinforce their theoretical Ada knowledge and see other aspects of the language.

We try as much as possible to use Free Software solutions, so that the organization members can easily duplicate our setup either at home or in other organizations. We have contributed back the GNAT port for Linux on SH4 as well as the Forth interpreter written in Ada[20]. We also encourage our members to create tutorials on the organization web site to share their knowledge; several ones have already been written, including one on Ada[23] (in French).

Complementing Ada with other languages such as Forth and Verilog brought us the flexibility we needed for interactive testing and tightly coupled software and hardware processing. We have the feeling that using those two languages did not endanger the system solid Ada foundations: Verilog drives hardware peripherals and is controlled directly from Ada without the need for a specialized Linux kernel device driver, and the Forth interpreter is written in Ada and is compatible with the Ravenscar profile.

We will continue to participate to the French robotics cup and we are currently examining the way to get an even better and more efficient software base. Our two most promising leads again involve Ada.

In the future, we could use a standard for model-based engineering such as AADL[1]. Ocarina[15], developed by another research team at our institute, would let us partition our program into various tasks executing on the two boards and communicating through the lightweight PolyORB-HI middleware.

We also plan to move the propulsion control system out of the FPGA onto a third board based on a STM32 processor programmed in Ada. We would then get a more flexible system capable of computing complex trajectories. This would familiarize our organization members with Ada programming on bare-board systems without any help from a full-fledged operating system.

8. REFERENCES

[1] As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506, November 2004.

[2] R. Barry. FreeRTOS Reference Manual - API Functions and Configuration Options. eBook, 2009.

[3] A. Burns. The Ravenscar Profile. *ACM Ada Letters*, 4:49–52, 1999.

[4] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical report, University of York, January 2003.

[5] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3 edition, 2005.

[6] B. Dupouy, O. Hainque, L. Pautet, and S. Tardieu. Overview: the SPIF project. In *Proceedings of the Reliable Software Technologies conference*, 1997.

[7] A. Ertl. A Portable Forth Engine. In *Proceedings of the EuroFORTH conference*, 1993.

[8] M. A. Ertl. Is Forth Code Compact? - A Case Study. In *Proceedings of the EuroForth conference*, 1999.

[9] B. S. Fagin. Teaching Computer Science with Robotics using Ada/Mindstorms 2.0. In *Proceedings of the 2001 Annual ACM SIGAda International Conference*, pages 73–78. ACM Press, 2001.

[10] P. Frenger. Forth as a robotics language: part two. *SIGPLAN Not.*, 32(6):19–22, 1997.

[11] P. Frenger. Robot control techniques: Part one: a review of robotics languages. *SIGPLAN Not.*, 32(4):27–31, 1997.

[12] J. A. Gutierrez, M. Naeve, E. Callaway, M. Bourgeois, C. Mitter, and B. Heile. IEEE 802.15.4: A developing standard for low-power low-cost wireless personal area networks. *IEEE Network*, 15:12–19, September 2001.

[13] J. Hightower and G. Borriello. Location Systems for Ubiquitous Computing. *IEEE Computer*, 34(8):57–66, 2001.

[14] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of fifth international conference*

on software reuse, pages 134–142. IEEE Computer Society Press, 1998.

[15] J. Hugues, B. Zalila, and L. Pautet. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4):1–25, July 2008.

[16] G. Jones. *Maximum PC 2005 Buyer's Guide*. Prentice Hall, 2005.

[17] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[18] B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.

[19] S. Tardieu. *GLADE: une implémentation de l'annexe des systèmes répartis d'Ada 95*. PhD thesis, École Nationale Supérieure des Télécommunications, October 1999.

[20] S. Tardieu. AForth, a Forth interpreter in Ada. `http://www.ohloh.net/p/aforth`, 2009.

[21] Technical Committee X3J14. *Forth*. American National Standards Institute, Inc., March 1994.

[22] D. E. Thomas. *The Verilog® Hardware Description Language*. Springer, 3 edition, May 1996.

[23] Télécom Robotics. Tutoriel Ada. `http://www.telecom-robotics.org/ada`, 2009.