Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems

The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion

Roberto Brega[†], Nicola Tomatis[‡], Kai O. Arras[‡]

[†]Institute of Robotics
Swiss Federal Institute of Technology Zurich (ETHZ)
CH–8092 Zürich
brega@ifr.mavt.ethz.ch

[‡]Autonomous Systems Lab
Swiss Federal Institute of Technology Lausanne (EPFL)
CH–1015 Lausanne
{nicola.tomatis, kai-oliver.arras}@epfl.ch

Abstract

Starting from a user point of view the paper discusses the requirements of a development environment (operating system and programming language) for mechatronic systems, especially mobile robots. We argue that user requirements from research, education, ergonomics and applications impose a certain functionality on the embedded operating system and programming language, and that a deadline-driven real-time operating system helps to fulfil these requirements. A case study of the operating system XO/2, its programming language Oberon-2 and the mobile robot Pygmalion is presented. XO/2 explicitly addresses issues like scalabilty, safety and abstraction, previously found to be relevant for many user scenarios.

1. Introduction

In the mobile robotics community we can observe two approaches to the subjects of self-contained autonomy and real-time. On the one hand, vehicles are used as ‘sensors with wheels’ where information processing is done either off-board or even off-board and off-line. This procedure offers advantages for the researcher; he can focus on a precise topic without taking into account further complexity due to integration. On the other hand, robots are used as fully embedded systems which provide all means to acquire, process and act on-line and on-board. These systems exhibit a degree of self-contained autonomy which is compatible with application requirements but can suffer from high complexity. The choice of hardware, but particularly the choice of the embedded operating system and programming language is crucial when deciding to face this further application-relevant complexity. In this paper we argue that, when the researcher, the student and the end-user need or wish a certain functionality, flexibility or safety of the robot, properties like real-time capability coupled with a strong typed programming language can help to fulfil these needs. We present a case study of the XO/2 operating system which runs on the mobile robot Pygmalion as an example where exigent user requirements could get translated into a system that facilitates application to a real-world robot.

1.1 Do we need self-contained systems?

Although an unspoken question among many roboticists, we estimate that the majority of mobile robot research platforms in use today could not operate in a fully self-contained mode since their algorithms rely at least partially on wireless connections to off-board infrastructure. Whereas this limitation might not be relevant for research, it will not be acceptable for many applications where operating environment size, economical aspects and safety issues do not allow off-board computing. Autonomy with respect to perception, energy and processing for fully self-contained autonomous decision-making is not an option but should be addressed in its full complexity already as a research topic.

1.2 Do we need real-time in mobile robotics?

When on-board computation is required we are typically confronted with limited computing power. Hence, meeting timing constraints becomes a problem which is *present but only hidden* when using off-board hardware of extensive processing power. In the commonly used approach, a small microcontroller of limited computational power handles the real-time aspects of vehicle control. Some simple program, hooked to a processor-interrupt, implements a trivial control loop that drives the robot’s actuators, exchanging limited information with a foreign computer through some serial wire. This can be viewed as a flexible and modular architecture approach, but, in most cases, it results in a system where the low-level controllers are treated as an untouchable black-box.

Progress in mobile robotics requires the researchers to access and improve all modules that compose the robot, from low-level real-time components to high-level reasoning systems. The performance of a mobile robot depends upon the interaction of components at all levels, and therefore these modules cannot be treated as independent units. For instance, if a researcher focuses exclusively on high level changes in an effort to improve mobile robot performance, he may be hindered by low-level behaviour that could be changed with ease if the robot’s real-time components were equally accessible. We argue that real-time capabilities with deadline-driven scheduling and a tighter integration between low-level and application modules are a welcome feature in mobile robots, allowing a safer composition and a transparent information exchange between each layer of the application software. Yet, real-time and composition capabilities are not the only pleasing qualities that a system could provide. The next section will highlight present-day problems and their requirements that roboticists face during their work with their systems.

2. Requirements From a User Point-of-View

Be the user a researcher, a student or an application engineer, they have specific requirements to the functionality of their robot. Without anticipating a choice, criteria from state-of-the-art research, education, ergonomics and applications are outlined. They represent what the users encounter in their work and what they may expect today from a modern development environment for embedded systems.

2.1 Requirements from research

This type of requirements stems from a need for flexibility and as few functional constraints as possible.

- *System scalability*. Multiprocessor-, multisensor- and multiaxis-extensions must be supported by the operating system in a coherent way. Adding degrees of freedom for mobile manipulation, adding a sensor or an extra computing unit has in the first place to be possible and then it must not results in a patchwork of component-specific development environments for the programmer.
- *Dynamic programming*. Simultaneous localization and map building or sophisticated data association techniques with multiple hypothesis-management ask, by the nature of the problem, for dynamic data structures. Map objects as well as hypotheses get inserted, fused or deleted. The explicit disposal of memory would impose artificial constraints onto these problems.
- *Target-to-target communication*. Multi-robot research requires means for unsupervised exchange of data between robots—preferably under use of standards.
- *Multi-modal user interfaces*. Interfaces of this type make use of multiple communication modalities like speech, motion, gestures, propioceptive- or visual-feedback. This asks for a system design with enough generalisation capabilities, in order to support additional serial lines, display units, haptic interfaces, A/D I/O signals etc.

2.2 Requirements from education and ergonomics

Mobile robotics is a multi disciplinary research domain that takes place mainly in the academic world. Robots are not programmed by highly specialised engineers and system integrators alone, but by researchers from various domains and students. This class of requirements originates from recognising this fact.

- *Ease of use* for academic environments. Students from different disciplines should be able to learn it in a short time. An easy to learn programming language and operating system favours an integration into courses, exercises or student competitions. The system should exhibit inherent mechanisms that protect the inexperienced user from himself (‘student-proof’).
- *Man-machine interfacin*g. The system must provide means for on-the-fly visualization of various on-board data e.g. for debugging or task supervision purposes. Possibly under use of wide-spread standards like Internet Engineering Task Force (IETF) backed protocols.
- *Code coherence*. In a complex mechatronic system, like a non-holonomic mobile robot with multiple sensors, processors and axis, the developer shall be able to consistently work in the same programming environment from time-critical low-level control to non–time-critical high-level algorithm design.
- *Fast edit-compile-run cycles*. A fast and safe mechanism to unload, edit, (cross-)compile, and dynamically link a piece of code dramatically improves the testing effectiveness of the developer.

2.3 Requirements from applications

Provided that no functional limitations hinder the development of the robot anymore, safety and economic issues mark this class of requirements.

- *Safety*. With the ever increasing presence of computers controlling critical systems—critical to missions, the environment, human lives, or the society—the safety of such systems is a prime concern. In some cases it is possible to statically enforce safety by using construction methods that simply exclude bad cases. In other cases it may be necessary to enforce safety dynamically: If something bad is about to happen, this gets detected and proper steps are taken. The attention to safety should span all aspects of the software constellation, from high-level behaviours to system-software primitives.
- *Real-time capabilities*. In most applications the shape and the kinematics of the robot is imposed by the task. For certain shapes and kinematic configurations, algorithms for obstacle avoidance, motion planning, trajectory execution and position control partially degenerate to trivial cases (e.g. for circular robots and/or holonomic kinematics). But in general, these algorithms require much computing power, the installation of user real-time tasks for position or trajectory control and, since they are safety-critical, means for their supervision. An operating system must support the researcher and the application engineer in the development and implementation of such algorithms.
- *Vehicle dynamics* and *speed*. Rarely an optimization criterion for research, operation speed can become an economic factor. Temporal imprecisions in the navigation algorithms which remain hidden at low speed be-

come apparent. The system must provide means for management of high resolution timestamps and their assignment to sensory inputs for on-the-fly implementations of these algorithms. For highly dynamic axis control, the scheduling timeslice must be sufficiently short.

- *Long-term reliability, ease of maintenance*. The life-cycle of a mobile robot usually spans over many years. The development of such a machine can take many months, during them application developer need to poke with the hardware and software specification. During the actual machine operation, maintenance of hardware and software must be guaranteed: The system should not be tied to a particular configuration of the hardware, and changes in the electronics or in the application should be as painless as possible.

2.4 Can a careful choice of the operating system help in meeting these requirements?

The requirements outlined above present a lot of design challenges: abstraction layers with well-defined interfaces, concurrence between tasks sharing different timing characteristics, blackbox-like components, dynamic behaviour, reliability, safety, and economic aspects, scalability.

It is useful to remember that the main purpose of an operating system can be summarised as that of managing system-wide resources through abstractions, while presenting well-defined interfaces to the applications. Unfortunately, most operating systems and frameworks do a poor job in supporting safety or general robot-programming patterns. The management of time as a system resource, the safe composition of software modules, the type-safety, the management of dynamic memory, all add up to a great part of the task of writing software for a mechatronic product.

There is no real reason not to have the required features embedded in the operating system. In fact, deadline-driven scheduling has already been widely recognised as a safer alternative to interrupts-driven systems [2]. Software composition is advertised everywhere: Why it cannot be implemented with the correct amount of security, like strongly checked interfaces or safe unloading. Dynamic memory management techniques, pioneered by lisp and smalltalk systems, could find their way in a embedded operating system, thus setting the applications free from dangling pointers and memory leaks—common problems that are far too frequent, awkward to track and to debug.

The difficult task of designing a software architecture for complex mechatronic systems, should not be further mixed up by these common, low-level problems. System-wide mechanisms need to be present, in order to lighten this already mighty effort. Mechatronic software-design strives for safety. Safety-driven operating systems are the most practical building-blocks for reaching this goal.

3. The XO/2 Real-Time System and Framework

XO/2 is an object-oriented, hard-real time system software and framework, designed for safety, extensibility and abstraction [3]. It takes care of many common issues faced by programmers of mechatronic products, by hiding general design patterns inside internal mechanisms or by encapsulating them into easy-to-understand abstractions. Careful handling of the safety aspects has been the criterion by which the system has been crafted. These mechanisms, pervasive yet efficient, allow the system to maintain a *deus ex-machina knowledge* about the running applications, thus providing higher confidence to the application programmer. The latter, relieved from many computer-science aspects, can better focus his attention to the actual problem to be solved.

3.1 Safety

The system sets higher standards for safety through a combination of programming paradigms and modern computer-science solutions. In order to understand this claim, a somewhat more technical definition of safety is required.

Safety, as used in the common speech, can be separated into the more technical terms of *safety*, *progress*, and *security* [4]. These terms can be summarised as follows: nothing bad happens, the right things do (eventually) happen, and things happen under proper authorisation (or potentially bad things happen under proper supervision). All three interact to make a system safe in a broader sense. A system that deploys static and dynamic enforcement of those aspects, can be said to provide a higher degree of safety. The static safety can be gained with a careful choice of the programming language, wherever dynamic safety must rely on run-time mechanisms checking that the programming invariants hold. Clearly, whenever possible, static safety should be preferred over dynamic safety.

3.2 The role of programming languages in safe systems

System components are the tools of a software engineer. The more safer the tools, the more reliable is the system. It is therefore natural to expect programming languages to help improve system safety. It is well-known that languages, or more precisely proper language paradigms and type systems, can do a lot in helping programmers to better realise their solutions. Strangely enough, despite the existence of better alternatives, a lot of safety-critical software gets implemented by means of programming languages that do a poor job at ensuring static or dynamic safety. Selling tools that try to fix the problem renders the situation even more ludicrous. Tools are used for uncovering errors that should not have been made possible in the first place, as for array indexes out of range, dangling pointers, casting

errors, and memory leaks.
The commonly used argument against using languages that are type-safe, is the inefficiency of the produced code. This misconception can be easily refuted. In the case of static safety, all restrictions are computed by the compiler (at compile-time), therefore there is no overhead in the code to be executed. When static safety cannot be enforced, dynamic checks are needed. The added safety, brought by the validation of the programming invariant at run-time, more than compensate the penalty paid in the execution time. In fact, there is no acceptable trade-off for letting a type-violation be passed.
The programming language chosen for XO/2 is Oberon-2 [5]. Oberon is a successor of Pascal and Modula, featuring strong-typing, compatibility by name and not by structure, object-orientation, and modularisation. Since the same characteristics are shared by the Java programming language, an ongoing effort aims at supporting the language, by natively compiling bytecodes at linking-time, in the native system environment.

3.3 Handling untyped operations

In the section above, it is stated that it is not always possible to ensure (type) safety statically, i.e. at compile-time. Notwithstanding the run-time checks needed for object-oriented polymorphic operations, this also holds for each potentially untyped action. Examples in this direction range from NIL-pointer dereferencing, stack overflows, and dangling references to unloaded modules.
Most of the aforementioned errors can be trapped by run-time checks emitted by the compiler. Anyway, the overhead in the execution time cannot be tolerated. Imagine checking each stack-push against the valid stack-ranges: You wouldn't invoke an activation frame anymore! A more aggressive technique, avoiding in-program checks, is by means of memory protection. This scheme, usually found on Unix derivatives, cheats the running programs (also called processes) by allocating to them different, disjunct virtual address spaces.
The major drawback of this scheme resides in the overhead that has to be paid for the reloading of the page-table and the memory management unit registers during context-switching. For a real-time system, as with XO/2, which fires the task scheduler with a time-slice quantum of 100 microseconds, this cannot be tolerated. Supported by the fact that no *explicitly programmed* untyped operation are allowed by the Oberon-2 language, we have favoured a more lightweight memory management scheme that helps in catching the possible untyped, unsafe operations emitted by the compiler, without imposing restrictions on what can be shared between programs, nor bringing an unacceptable overhead during context-switching.
The chosen scheme, makes pervasive use of the underlying memory management unit of the PowerPC architecture, by creating a single virtual address space, where virtual address ranges are allocated to the running processes. Following this method, NIL dereferencing can be mapped to an invalid virtual page. In the same fashion, stacks can be allocated with guard pages between them, thus actively guarding against stack overflows. Additionally, within a virtual address range, the stack can be allowed to grow whenever the running task needs it, without asking the programmer to explicitly demand a bigger stack at task creation. Module unloading is handled similarly: When modules are removed from the system, their virtual address range is invalidated, thus preventing dangling procedure variables to execute upon non-trusted code or data.

3.4 Automatic reclamation of dynamic memory

In a highly dynamic, object-oriented, composable system, the central knowledge of all references that exist for a particular object becomes hard to maintain as the dynamic loading of extensions augments. Even worse, it becomes impossible for a programmer to keep track of references in a safe way when the language doesn't impose restrictions on the passing and copying of references. This brings us to the sheer conclusion that in a dynamically extensible system, explicit deallocation of objects is not feasible. The failure of realising this introduces a new class of run-time problems, like dangling references and memory leaks: If the object is disposed to soon, some stale references could access the object while the same memory block is being referenced by someone else; on the contrary, late or non-existent disposal induces memory leaks, i.e. unused memory doesn't get reclaimed.
The only safe possibility for object reclamation is by means of a system-wide mechanism performing automatic storage reclamation: a so-called garbage collector [6]. A garbage collector decides upon the liveness of heap objects with their reachability, starting from a working set of global and local references. After complete traversal of the heap data structures, objects that haven't been visited by the collector's marking get disposed.
XO/2 deploys a very robust, incremental, real-time compatible *mark-and-sweep* garbage collector with object-finalisation that combines good collection performance with no memory requirements at execution time. The latter is more important when the collector is kicked by a low-memory condition, i.e. it can complete the traversal and the collection of the heap-space without demanding memory. Moreover, the proposed solution works very well in a preemptive scheduling environment, without blocking nor delaying tasks performing accesses to heap-objects.

3.5 Modularisation and separation of concerns

One of the most important design principle is the separation of concerns. This principle requires a system to be structured into subsystems, also called modules. Modules should expose an interface by exporting functions and procedures to the clients. The functionality of a module is accessed only by means of its interface; the interface can be generalized enough to hide most of the implementation details, thus establishing and guaranteeing invariants for its states and procedures. Disjoint, orthogonal modules implementing this design principle can be exchanged without invalidating clients, therefore leading to a dynamic composition of the system.

An important precondition for the realisation of this design principle is the presence of safe dynamic loading and unloading of compilation units. XO/2 provides the required safety by checking at compile time and at linking-loading time the formal interfaces against the actual ones. Only interface-compatible modules may be loaded in the system, without threatening the safety of the dynamic composition.

A different, non trivial task resides in the dynamic unloading of modules, i.e. when a module can be safely removed from the system. By means of reference counting, lexical scopes and virtual memory ranges, XO/2 can guarantee that a needed module will not be unloaded and that stale references will be trapped before execution.

The presence of safe dynamic loading and unloading in XO/2, along with very short edit-compile-run cycles, has been one of its most appreciated features. In fact, during the development of a complex application, different programmers can safely test new code modules without threatening the stability of the system and applications. It is not uncommon that an XO/2 machine will run, uninterrupted, for several days, during them application programmers actively program and test part of a software constellation.

3.6 Process scheduling

The principal responsibility of a real-time operating system can be summarized as that of producing correct results while meeting predefined deadlines in doing so. Therefore, the computational correctness of the system depends on both the logical correctness of the results it produces, and the timing correctness, i.e. the ability to meet the deadlines of its computation [7].

A real-time application can be modelled as a set of cooperating tasks. These tasks can be classified according to their timing requirements, as hard–real-time, and non–real-time. A hard–real-time task is a task whose timely execution is labelled as critical to the operation of the whole system. Consequently, it is assumed that the missing of the deadline can result in a system failure. Non–real-time tasks are those tasks that exhibit no real-time requirements (e.g. system maintenance tasks running in the background).

XO/2's real-time process manager implements a static, *earliest-deadline-first* scheduling algorithm with *admission testing*. With this algorithm, the pool of real-time tasks is statically sorted according to the their deadlines. The first one, i.e. the one with the shortest deadline, will be set for execution. This task will remain in the foreground, until its normal execution cycle is completed, or when a task characterized by a shorter deadline has been activated by the occurrence of some event, such as the expiration of a waiting period or user intervention. The process manager is also responsible for dispatching non–real-time tasks, also called threads. Since their computations can be delivered any time, threads are brought to the foreground only when no other real-time task is pending, waiting for being dispatched. The non–real-time scheduler chooses the thread to be scheduled according to its priority. Threads carrying the same priority are taken in the foreground in a round-robin fashion. *Anti-starvation* mechanisms and *priority inheritance* guarantee fairness and progress.

3.7 Other OS functionality

Our system provides a wide range of non-core functionality, like multiple-filesystem support, streams-based I/O, TCP/IP networking software, internet standard servers and clients, object-oriented databases of periphery hardware.

The robust, threaded TCP/IP stack allows a wide array of network protocols to be implemented. A standard, full-blown release of XO/2 comes with a Telnet server, an FTP server, an HTTP server, TFTP, SNTP, SMTP, and POP3 clients. Applications looking for computer-based man-machine interaction are usually realised as a set of web-pages. A client agent, like a web browser, can present on-robot information without need of post-processing, or call CGI-commands that are actually normal application entry-points, exposed as commands. Object linearisation through XML allows browsers, java-applets or custom solutions to access remote objects on the HTTP transport. Facilities like the in-system creation of GIF and JPG images are used for streaming visual information to the client.

4. Pygmalion's Software Implementation

Pygmalion is a mobile robot recently built at the Autonomous Systems Lab, EPFL for application-near research purposes (figure 1). It makes extensive use of the facilities provided by XO/2. Every critical, periodic task is implemented as a hard real-time task. The operating system and programming language guarantee both safety and security of the executed code, while ensuring progress and timing correctness through the deadline driven scheduler.

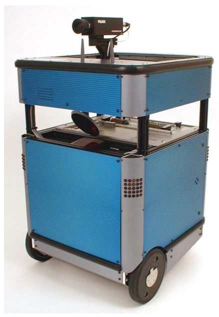


Figure 1: *The autonomous robot Pygmalion. Its controller consists of a VME standard backplane with a Motorola PowerPC 604 microprocessor, clocked at 300 Mhz. Among its array of peripheral devices, the most important are the wheel encoders, a 360° laser range finder and a grey-level CCD camera.*

When unexpected events occur, the operating system is responsible for supervising a proper counter-measure.

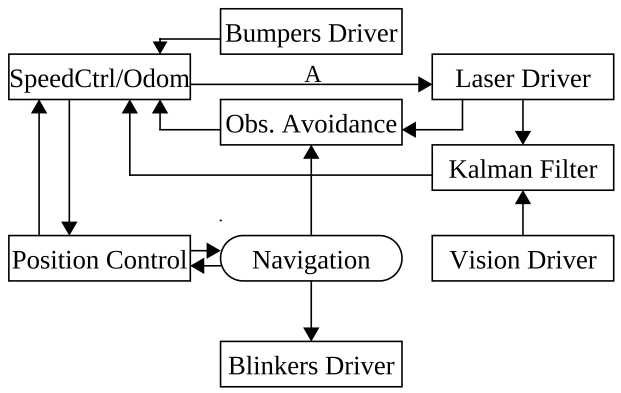
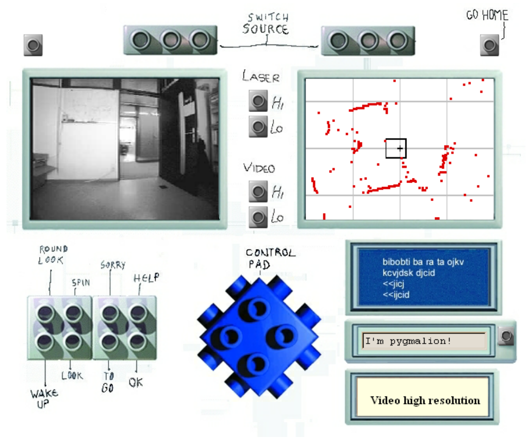


Figure 2: *The tasks constellation deployed on Pygmalion. Rectangles indicate hard real-time tasks, arrows stand for information flow. The former are the only means to guarantee calculation time for vital functions under the typical conditions of unpredictable processor load.*

The task constellation running on Pygmalion is depicted in figure 2. The bumper driver and the speed controller which calls also odometry run at 1 KHz, the position controller is scheduled with a frequency of 50 Hz. The laser scanner takes 360 ms for a complete mirror revolution and compensates on-the-fly the distortion imposed by the vehicle movement during acquisition for each arriving range reading (this explains arrow A). Obstacle avoidance and localization are vital functions and are also installed as RT task with an appropriate worst case period. The multi sensor localization cycle is trivially limited by the slow period of the laser scanner. Temporal coherence of acquisition and localization result, which are in the past, is guaranteed by means of timestamps provided by odometry, the vision and the laser driver. Forward simulation of the odometry model yield finally the actually valid position update. The navigation module, implemented as a non–real-time task, is used for global planning and mission control. This implementation has been extensively used for multisensor on-the-fly localization with vehicle speeds up to 0.8 m/s [1].

Pygmalion's web interface is shown in figure 3. It resides in the XO/2 web server and accesses the underlying application software components through CGI-calls. This seamless integration is used for visualizing on-line raw-data retrieved with both sensors, their extracted features and robots pose information by on-the-fly generation of GIF and JPG images. The interface allows also the robot to be guided, access the bi-lingual speech processor and execute a library of robot gestures for human-machine interaction.

Simultaneous localization and map building is also currently investigated. In this case, only new objects have to be explicitly allocated, the rest of the dynamic handling is left to the operating system; the garbage collector ensures that the unused memory will get automatically reclaimed.



bibobti ba ra ta ojkv
kcvjdsk djcid
<<jicj
<<ijcid

I'm pygmalion!

Figure 3: *On-robot information as accessed through a web client. The interface can control the vehicle, too.*

5. Conclusions

The need for self-contained autonomy, real-time, and safety in mobile robots cannot be quantified. Although solutions exist that circumvent these issues by means of off-board infrastructures, custom hardware that has been tailored to the task, and artificial environments, we argue that in many applications these restrictions cannot be accepted. Meanwhile, the increasing complexity of mobile robots sets higher requirements to the application software and, indirectly, to the operating system. Safe composition of software modules, type-safety, deadline-driven scheduling and automatic memory reclamation mechanisms can relieve the application programmer from many time-consuming implementation issues, while raising the safety-bar.

References

[1] K.O. Arras, N. Tomatis. Improving Robustness and Precision in Mobile Robot Localization by Using Laser Range Finding and Monocular Vision. *Proc. of the 3rd European Workshop on Advanced Mobile Robots (Eurobot 99)*, Zurich, 1999.

[2] M. Joseph (edited by). Real-time Systems: Specification, Verification and Analysis. *Prentice Hall International Series in Computer Science*, 1996, Prentice Hall.

[3] R. Brega. A real-time operating system designed for predictability and run-time safety. In *Proc. of The Fourth Int. Conf. on Motion and Vibration Control (MOVIC)*, Zurich, 1998.

[4] C. Szyperski and J. Gough. The role of programming languages in the life-cycle of safe systems. *Second Int. Conf. on Saftey Through Quality (STQ'95)*, Kennedy Space Center, Cape Canaveral, Florida, USA, October 1995.

[5] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4), 1991.

[6] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. *Submitted to ACM Computing Surveys.*

[7] F. Panzieri, R. Davoli. Real Time Systems: A Tutorial. *Technical Report UBLCS-93-22*, October 1993, Univ. of Bologna, Italy.