"May You Have a Strong (-Typed) Foundation"[1]
Why Strong-Typed Programming Languages Do Matter

Nicola Tomatis[a,d], Roberto Brega[b], Gabrio Rivera[c], Roland Siegwart[d]

[a]BlueBotics SA
PSE-C
CH-1015 Lausanne
nicola.tomatis@bluebotics.com

[b]LogObject AG
Technoparkstrasse 1
CH-8005 Zurich
roberto.brega@logobject.ch

[c]ETH Zurich
Institute for Information Systems
CH-8092 Zurich
rivera@inf.ethz.ch

[d]EPF Lausanne
Autonomous Systems Lab
CH-1015 Lausanne
roland.siegwart@epfl.ch

***Abstract*—Programming efficient and reliable code can be considered a non-trivial task, as it requires deep understanding of the problem to be solved along with good programming skills. However, software frameworks and programming paradigms can provide a dependable infrastructure upon which better programs can be written and deployed. This allows engineers to focus mainly on their task, while relying on the underlying run-time environment for taking care of low-level programming issues, such as memory allocation and disposal, typing consistency and interface compliance.**

In this paper, we argue that strong-typed programming languages and paradigms offer a valid support for the production of reliable programs. Aware of the challenges of formal measurement metrics for code quality, we present the benefits of strong-typing by considering a practical application: The design and implementation of RoboX, a tour-guide robot for the Swiss National Exhibition Expo.02. The example is extremely well suited for such a discussion, since complex mechatronic applications can be considered critical systems—i.e. systems whose failure may endanger missions, lives and society—thus their reliability has to be made a prime concern.

I. Introduction

In the pioneering era of calculating machines, computers were programmed using primitives, which were able to directly instruct the underlying processor architecture. These instructions were either coded as sequences of digits—the machine language—or, more conveniently, in a human-readable version—the assembly language—which would be translated to machine language by a so-called *assembler*. Machine and assembly language are sometimes referred to as the first two generations of programming languages.

In the sixties developers began to realize that programs written in assembly language were tedious to write, the produced code was tied to a particular processor architecture and their engineering was extremely error-prone, as the semantic gap[2] was too large.

The third generation of languages represented the dawn of the high-level languages. They contributed to the reduction of the semantic gap by taking care—through compilers and interpreters—of the mapping between language constructs and the underlying instruction set.

The past thirty years witnessed many new languages and paradigms, such as *imperative* or *procedural languages* (e.g. C, Oberon), *logical* (e.g. Prolog), *functional* (e.g. ML) and *object-oriented* (e.g. C++, Java, Eiffel, Oberon-2). The new paradigms also introduced new features such as the strong-typing. Examples of strong-typed programming languages can be found in the Ada, Java and Oberon programming languages. C and C++ are sometimes described as strongly typed, they are indeed weakly typed.

The need for a safe type system in programming languages was a logical consequence derived from a set of requirements, coming from developers looking for rapid development-time and ease of maintenance, without having to pay to incur in performance penalties.

A milestone achievement for the definition of the notion of type may be identified in the work of Cardelli and Wegner [7]. They argue that types, much akin to mathematics formulæ, impose constraints which help to enforce correctness, that is, type-bound programming languages impose constraints on the objects' interaction, thereby preventing objects from an inconsistent interaction with other objects. Cardelli and Wegner metaphorically depict types as a set of *clothes* that protects an underlying untyped representation from arbitrary or unintended use. In other words, types provide a *protective covering* that hides the underlying representation of the information, while constraining the way objects interact.

As explained in [15], a type system is an instrument that can be used for enforcing rules in the application domain and detecting their violations. The type system of a programming language can be characterized as strong or weak, and the type checking mechanism as static or dynamic. Strong-typing is a strict enforcement of type rules because all types are defined and known at compile-time. Type checking ensures that the operands of an operator are of compatible types. A type check is static if it occurs before run-time and remains unchanged throughout program execution, while it is dynamic if it occurs (or can change) during the execution of the program. A type error is the application of an operator to an operand of an inappropriate type. A programming language is strongly typed if type errors are always detected. Hence, the most important advantage of strong-typing is that it allows the detection of the misuses of variables that result in type errors. Strong-typing catches more errors at compile time than weak-typing, resulting in fewer run-time exceptions.

The real benefits of strong-typing have been often contested. We contribute to the dispute by presenting the experiences we gained with the design and implementation of *RoboX*—a tour guide-robot for the *Swiss National Exhibition Expo.02*. The analysis of the software failures encountered by the robot's system, let us argue that strong-typed programming languages and paradigms offer a valuable support for the production of reliable code.

The paper is structured as follows: Section II, briefly describes the hard–real-time operating system XO/2, with its features and its support for the strong-typed programming language Oberon-2. XO/2 is the operating system chosen for the implementation of the robot's real-time requirements. Section III presents the whole project: The architecture of

1. The title quotes, nearly verbatim, Bob Dylan's "Forever Young" (Columbia, 1974). Likewise, it shall be taken as a wish and, concurrently, as a recommendation.
2. The difference between the complex operations performed by high-level language constructs and the simple ones provided by computer instruction sets. It was in an attempt to try to close this gap that computer architects designed increasingly complex instruction set computers.

RoboX, its requirements and characteristics. Section IV dissects the structure of the RoboX software development. Section V documents the results that have been collected during the whole period of activity. Aware of the difficulties in defining strict measurement metrics, we compare and evaluate the numerical outcomes in Section VI. Section VII wraps up our experience, with some considerations on the features of strong-typed programming languages vs. the weak-typed ones.

II. XO/2 Hard–Real-Time Operating System

XO/2 is an object-oriented, hard-real time system software and framework, designed for safety, extensibility and abstraction [6]. It is written in, and designed for the object-oriented language Oberon-2 [19]. 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.

Safety, as commonly used, is a rather general notion of "the system does what it should, and does not what it should not". A more formal separation of what is perceived as "safety" is required in order to analyze in greater detail how safety can actually be achieved and supported.

Szyperski [23] separated safety in the more technical terms of *safety*, *progress*, and *security*. These terms can be summarized as follows: Nothing bad happens, the right things do (eventually) happen, and things happen under proper authorization (or potentially bad things happen under proper supervision). All three interact to make a system safe in broader sense.

Safety can be enforced statically or dynamically. In some cases it is in fact possible to statically detect safety violations (or security or progress) by means of (simple) formal verification performed off-line. In other cases, safety needs to be enforced through supervised execution: If something potentially unsafe is detected, execution is stopped and proper countermeasures are taken. XO/2 addresses safety concerns through the deployment of several distinct mechanisms.

Memory safety is solved with the symbiotic effort of the programming language, which takes care of the static type-safety and a run-time enforcement system, which supervises memory accesses while isolating spurious fetches through run-time mechanisms, such as the light-weight sandboxing approach, described in [5] or the automatic real-time compatible garbage collector presented in [4].

Progress is handled by an earliest deadline driven scheduler (EDF), which allows the application programmer to specify the task's execution priority by means of its timing constraints, i.e. its *duration*, its *deadline* and its *period*. The scheduler tests new tasks for admission in the task set upon task creation, while continuously monitoring the application run-time for constraints violations.

A novel approach enables the application programmer to perform a reliable estimate of the task's duration—its worst-case execution time—by harnessing the performance monitoring hardware of the underlying processor architecture, as disclosed in [8].

Several complex mechatronic applications have been developed and deployed on top of XO/2, in several university projects and commercial products, ranging from automated anæsthesia devices [11] to industrial manipulators [13], from endoscopy devices [25] to service robots [3], [24].

III. RoboX Project

Robotics was a very successful project presented at *Expo.02*—the Swiss National Exhibition in Neuchâtel. Its goal was to convey the feeling of increasing closeness between human and machine. Visitors were able to interact with up to eleven autonomous, freely navigating tour guide robots.

Both the typical highly dynamic environment of an exhibit, and the high expectations anticipated by the visitors, imposed various constraints on the robot's design and control. This led to the mobile platform to be specified as follows.

- The navigation shall perform with full autonomy and with a high degree of reliability in an environment designed around human beings, crowded with visitors and without the help of any artificial landmarks for the localization.
- Safety shall be treated as a prime concern for humans, furniture and the robots.
- The robots shall interact with visitors by means of a bi-directional and multi-modal interface, comprised of speech—English, German, French and Italian—facial expressions, face tracking, visual clues to convey emotions—through icons on LED matrix—input buttons and robot motion, interpreted as a gesture.
- The robots shall require minimal human intervention and ease of maintenance.

The appearance of the robot was designed in collaboration with industrial designers. This co-operation yielded RoboX, the mobile robot platform shown in figure 1.

The robot is composed of a navigation base and an interaction turret. The control system was designed by keeping in mind that the safety of both humans and the robot have to always be guaranteed and it is composed of a CompactPCI rack containing two processor boards sporting, respectively, an Intel Pentium III and a Motorola PowerPC 750. The latter is connected by the PCI backplane to an analog/digital I/O card, a Bt848-based frame grabber, an IndustryPack encoder module and an IndustryPack high bandwidth RS-422 interface. Furthermore a Microchip PIC processor is used as a redundant security for the system.

The navigation software is deployed on top of the hard–real-time operating system XO/2 [6] running on the PowerPC. This processor has direct access to the camera looking at the ceiling, the two SICK LMS laser range finders, the tactile plates and the main drive motors. It communicates with the interaction PC through Ethernet via an on-board hub.

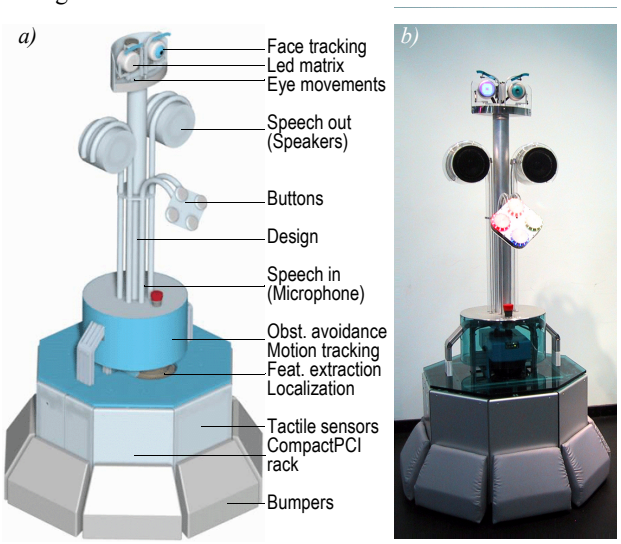

Figure 1: *a) Functionality of the tour guide robot RoboX. b) An image of RoboX 9.*

The interaction software runs on top of Windows 2000 on the embedded PC. This allowed integrating commercial off-the-shelf (COTS) software for speech synthesis and recognition, while making scenario development easier. The PC has direct access to the eye camera, the pan-tilt eyes and eyebrows controller, the input buttons, the two loudspeakers and the microphone. Both CPUs are connected via radio Ethernet (IEEE 802.11) to an external computer for supervision purposes, such as the monitoring of its stati on a graphical interface, which serves as a data aggregator.

IV. RoboX Software

The robot embeds both an Intel Pentium (PC) and a Motorola PowerPC (PPC) system. The software has been designed by taking into account the features and characteristics offered by the two embedded systems. Having being designed for complex real-time mechatronic applications, XO/2 was the natural choice for controlling the low-level hardware and the time-critical tasks. Contrastingly, the functionality requiring the COTS components has been implemented on the Windows machine because of their wider availability (e.g. *MBrola* for speech synthesis, small FireWire cameras in the robot's eye, vision libraries, etc.).

The design of the software that operates on each of the eleven robots was started at the end of year 2000. Even if the specification of the functionality was very hard due to the lack of references for a project of this kind, two milestones were defined at the beginning: Navigation and interaction. For the navigation the team could rely on the research of the Autonomous Systems Lab, EPFL, while for the interaction little experience was readily available. However, after various attempts, the basic functionality for the interaction was laid down and encompassed the navigation, the speech synthesis, the eye movements, the face tracking, the feedback buttons, people detection, speech recognition and the LED matrix.

The development of two prototypes started January 2001; The software development began April 2001. The team was composed of three special interest groups (SIGs): Robot prototyping and integration (three persons), navigation (five persons) and interaction (six persons). The navigation and interaction SIGs were responsible for the software implementation under supervision of a computer scientist. The navigation team was led by an electronics engineer and comprised two computer scientists and two microengineers (one of each was a student). The interaction team was led by an electronics engineer and incorporated four microengineer.

The code developed by the *Robotics* team checks out at 1376 KB of compiled dynamically-linked executable for the navigation and 1703 KB for the interaction. The effort can be quantified in five[1] man-years for the navigation part and six man-years for the interaction part. Table 1 shows some notable data regarding the RoboX software development.

	Navigation	Interaction
Team [persons]	4	6
Total work [man-years]	4 + 2 (reuse)	6
Micro-eng. [man-years]	1.5	5
Electronics eng. [man-years]	1 +1 (resuse)	1
IT eng. [man-years]	1.5 + 1 (reuse)	0
Compiled code [KB]	1376	1703

Table 1: *Overview of the Teams' Working Power and Knowledge.*

1. Some code reuse—roughly checking out at two man-years—should be taken into account for the navigation software. It is the result of prior research of the Autonomous Systems Lab, EPFL.

Both teams shared a wealth of similar issues, namely hardware design, proprietary periphery devices, control-loop software, low-level drivers, software in-the-large, mathematics models, algorithmics and man-machine interaction. It has to be noted that the navigation portion of the software is the only one that can be deemed critical to persons and the environment, thus adding to its complexity.

V. Results

The whole 159 days of operation—from May 14th to October 20th, 2002—are available for statistics. Every day and during the whole opening time (9:00 AM to 9:00 PM), six to eleven freely navigating tour-guide robots have given tours on the approximately 320 m^2 surface of the exhibit. At the end of the exhibit, the robots *served* more than 680'000 visitors for a total of 13'313 hours of run-time. In order to perform the task, they travelled 3'316 km for a total moving time of more than 9'415 hours. This yields a mean displacement speed of 0.098 m/s.

Run time	13'313 h
Movement time	9'415 h
Travelled distance	3'316 km
Average speed	0.098 m/s
Failures (total / critical / uncritical)	4'378 / 4'086 / 292
Critical failures (PC / PPC / HW)	3'216 / 694 / 98
Critical software failures (PC / PPC)	3'216 / 190
Visitors	686'405

Table 2: *Failures survey and statistics.*

The wealth of information recorded by the system monitors allowed us to study the behavior of the robots, to understand their operational-time and, most notably, to recognize the reasons for the failures.

On the one hand, we deem a failure *non-critical*, when an exceptional condition cannot hinder the robot in performing its current task. For instance, it is not considered a critical failure if the robot stops sending images streamed over HTTP to the supervision computer.

On the other hand, we deem a failure *critical*, when an exceptional condition forces the robot to interrupt its activities and wait upon human intervention. The abrupt failure of the scenario controller or mishap during obstacle avoidance can be considered examples of critical failures. A particular class of critical failures can be identified in those failures requiring a restart of either the Pentium or the PowerPC system. They are explicitly handled by our taxonomy, as they require more time before the robot can return to its normal operational state.

From table 2 it can be seen that the non-critical failures represent only a small portion of the total amount of failures (6.7%). Since they do not substantially reduce the capabilities of the robot, they are not considered in our analysis, which will focus on the failures deemed critical or requiring a reboot.

The mean time between failure (MTBF) of the whole robot (PC, PowerPC and hardware) was, during the first three weeks, 1.41 hours. Subsequently, the MTBF has been 4.02 hours.

The decreasing failure rate depicted in figure 2, can be tracked down to the rapid improvement of the software quality on the PC, which took place as soon as the robot was deployed under real-world conditions.

Another interesting chart is depicted in figure 3, where all the critical failures coming from the navigation software (PowerPC system) are displayed. During the first three weeks, errors in the safety-critical tasks were treated by the security controller, and could sometimes require a reboot in order to restart the trapped task. This has been partly addressed in order to avoid rebooting, thereby allowing a much faster handling of the ex-

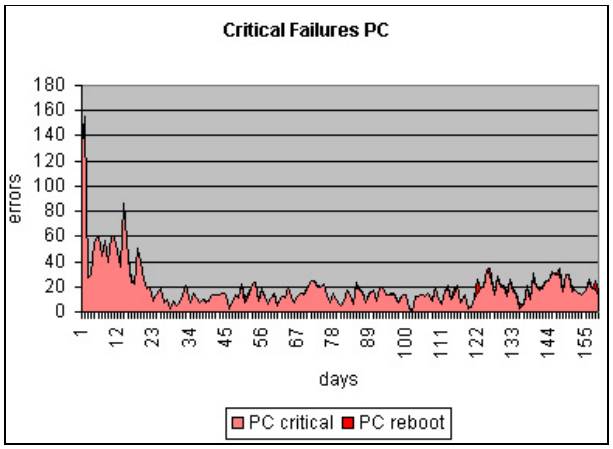

Figure 2: *Critical Failures on the Pentium (PC) System.*

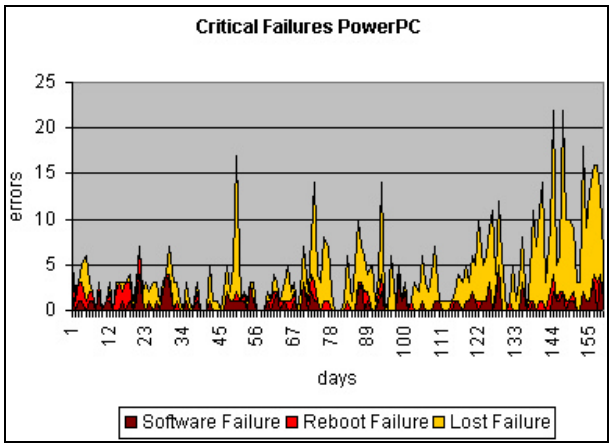

Figure 3: *Critical Failures on the PowerPC (PPC) System.*

ceptional condition. Figure 3 reports failures of the localization system—*lost failures*[1]—, which also require manual intervention, but are not directly related to the quality of the software. They cause 504 errors and thus represent the 73% of all of the critical failures on the PowerPC system. These failures, while being very interesting from a robotics point of view, are negligible for our analysis of the software quality.

The MTBF for the PowerPC system has been measured to lie between 10 and 80 hours. By taking into account the software errors only—and not the lost situations—the MTBF over the whole period checks out at 70.1 hours.

By comparing figure 2 and figure 3, we can derive that the amount of errors manifested by the interaction software (3'216 failures) is one order of magnitude larger than those from the navigation software (190 failures, without taking into account the 504 lost situations).

While the navigation software could be considered more ready for prime time—due to the partial reuse of code, checking out at approximately 200 hours cumulated run-time for some navigation modules—the difference becomes negligible as the first three weeks of operation already yielded 1'686 hours of cumulated run-time.

It may be interesting to analyze the particular issues experienced by the three major components, i.e. the PowerPC system, the Pentium system and the robot hardware.

The typical failures encountered on the PowerPC can be summarized as follows:

- Navigation: In some cases, the robots were unable to determine their location and thus notified a *lost* situation.
- Obstacle Avoidance: The process deadline was missed. Since XO/2 is a hard–real-time operation system, the missing of a deadline is a potentially dangerous situation, thus the process is stopped and an exception is thrown.
- Mission halted: In several cases, various modules encountered semantics issues, such as *divisions by zero*, *array indexes out of range*, *run-time type checks*, *assertions*, etc.

Conversely, the typical errors experienced by the Pentium system were:

- Fatal errors: The application **.exe* trapped and was closed by Windows.
- Memory errors: Memory errors resulted from a plethora of different causes: Spurious pointer references; pointer arithmetic overflows; arithmetic operations on non-initialised, null, or invalid pointer; read/write operations through non-initialised or null pointers; procedure calls through non-initialised, null or invalid pointers; wrong type casting; array references out of declared bounds and non-initialised array index. Furthermore, memory leaks and dangling pointers have harmed the reliable run-time of the application.
- FireWire issues: Problems related to the FireWire interface could result in the freezing of the whole operating system. This is due to the Windows 2000 architecture, which departed from the pure micro-kernel approach of Windows NT 3.x and allows driver software to run in the same address space of the NT kernel.

The robot hardware has seen the following faults:

- BreezeNET wireless ethernet interface (IEEE 802.11): Failures, denial of services, jamming have intermittently undermined the functioning of the networking hardware.
- SICK LMS laser scanner: Each robot was equipped with two SICK LMS laser scanners, which, sometimes, had transmission, reading or calibration failures. This flaw was strictly related to a high temperature in the environment.

VI. Discussion

By looking at the classes of errors listed above, the following aspects are easily brought to attention: On the one hand, the errors exhibited by the two different platforms do not share anything in common; On the other hand, the frequency of error conditions on the embedded PowerPC platform is an order of magnitude smaller than that of the Pentium system.

This difference may be related to several different reasons: In fact, the complexity of the tasks to be programmed for the two systems, the specific characteristics of the systems themselves and the programming experience of the single engineers could all have contributed to make one system more reliable than the other. Nonetheless, it would be interesting to understand which differences played a major role in this particular case, in order to infer generally valid rules for future projects.

We started by evaluating the software quality. This first step required us to find a metric, which would be able to identify and quantify the peculiarities of the code driving RoboX.

This question is not new in the field of computer science. The assertion that the *lines-of-code* metrics offers a rough measure of code and does not measure its content at all, fostered computer scientists to devise new models for defining such characteristics. The following paragraphs serve as a short survey of their work.

In the seventies, researchers such as Boehm [2], Barbacci et al. [1], Deutsch and Willis [9], Evans and Marciniak [10] tried to establish a hierarchical relationship among a set of quality

1. The vast majority of the failures in the localization software are due to visitors or untrained personnel, who fiddled with the robots without disconnecting the motors from the amplifiers. This caused large errors in the odometry not taken into account in the models and, consequently, failures in the localization.

measures. They took into account categories such as *performance*—i.e. how well programs function—*maintenance*—i.e. how easily programs can be corrected—*adaptation*—i.e. how easily programs can evolve or migrate—and *user satisfaction*—i.e. how well programs meet the users' requirements. A drawback of this approach is that a *real* quantitative aspect of the measurements is still missing.

Other researchers tried to measure the program's complexity directly gathered from the source code. In doing this, they stressed computational complexity. As defined by Halstead in [12], the complexity is extracted from the amount of operators and operands in the program. However, the Halstead measure did not receive unanimous acceptance: The critics ranged from "*unreliable"* (Jones [14]) to "*one of the most valid measures of maintainability"* (Oman [20]). The Halstead measurement metrics are based on four scalar numbers derived directly from a program's source code, namely the amount of distinct operators (n_1), the amount of distinct operands (n_2), the total number of operators (N_1) and the total number of operands (N_2). From these numbers, five measures are derived, namely the program length ($N = N_1 + N_2$), the program vocabulary ($n = n_1 + n_2$), the volume ($N = N(\log_2 n)$), the difficulty ($D = (n_1/2)(N_2/n_2)$), and the effort ($E = DV$).

These measures are quite simple to calculate once the rules for identifying operators and operands have been determined. But as Szulewski noted in [22], establishing these rules could be quite a difficult task. The Halstead measures have been criticized for several reasons, among them is the claim that they are a weak estimate for they measure a lexical and textual complexity rather than a structural and logic flow complexity.

To overcome such a limitation other proposals were considered. A relevant one is the *cyclomatic complexity*. Introduced by Thomas McCabe in 1976 [18], it measures the number of linearly-independent paths through a program. This measure provides a single ordinal number that can be compared to the complexity of other programs. As one of the more widely-accepted software metrics, it is intended to be language independent. On the one hand, as presented in [26], cyclomatic complexity describes a methodology for software testing and related software complexity analysis techniques. On the other hand, it has also been extended to encompass the design and structural complexity of a system, as described in [17].

The cyclomatic complexity of a program is calculated from a connected graph of the code, which represents the topology of the control flow. The complexity number is the sum of the number of edges in the graph with the number of connected components, subtracted by the number of nodes in the graph. In order to be able to actually weigh these elements, McCabe established a counting convention[1]. The complexity number is generally considered to provide a stronger measure of a program's structural complexity than the one provided by counting lines of code.

A large number of programs have been measured and ranges of complexity have been established. The resulting calibrated measure can be used in development, maintenance and re-engineering situations to develop estimates of risk, cost, or program stability. A low cyclomatic complexity contributes to a program's understandability, while studies show a correlation between a program's cyclomatic complexity and its error frequency.

It has become common practice to combine measures to fit the specific program environment. Thus, many measures are to some degree complementary. Oman in [21] presents a very comprehensive list of code metrics that are found in maintainability analysis, and orders them by degree of influence on the maintainability measure being developed in that effort, e.g. lines of code per program, lines of comments per program, variable span per program and lines of data declarations per program.

The main drawback of these technique—as highlighted in the survey work of Marciniak [16], which encompasses the various software-complexity measures and integrates them into a common framework—relies in the questionable variables that eventually make their way into the models. Therefore, instead of relying on some metrics, which would get polluted anyway by subjective interpretation or ad-hoc weighting, we decided to rely on our "qualitative" perception of the problem space.

The problem spaces addressed by the two systems show a similar degree of complexity. Where the PowerPC system extracts precise geometry features of the environment through laser scanning, the Pentium system gains cursory perception of the surroundings, by means of a video camera. Similarly, where the PowerPC system moves through a finite state machine for its jobs' handling, the Windows box follows an automata for implementing the man-machine interaction.

As a side effect, the complexity of the implementation of the two sub-systems can be considered roughly similar. This stems from the fact that both parts need to process the environments through geometry models, while handling unexpected situations during their run-time. Incidentally, a superficial code-review reveals a similar number of lines-of-codes, roughly the same amount of basic-blocks and a comparable complexity of their execution-paths. Both systems present a hefty share of mathematical computations.

As a last remark, it can be noticed that the two sub-systems have been programmed by two separate teams of engineers, who were similarly qualified for the task: Their background and experience—as mentioned in Section IV—can be considered comparable, their profile can be qualified as high-level.

By taking into account the slightly larger amount of code-reuse, we were expecting the PowerPC to perform more reliably than the Pentium system—by a factor of 2. Surprisingly, we were confronted with a system, which was *16-times* more reliable than the other, thus surpassing our most optimistic expectations. While not disputing the differences between the two tasks, we argue that they alone cannot be deemed responsible for such a difference in their respective run-time reliability. Therefore, other clues have to be found elsewhere.

The single noteworthy distinction that differentiates the two platforms can be found in the way they address safety with respect to memory. Whereas the PowerPC code was created by using a strong-typed programming language and it was run on top of an operating system that provides a plethora of run-time mechanisms aimed at enforcing safety and semantic correctness—such as a automatic memory reclamation and run-time typing information—nothing similar can be found on the Pentium system, whose code was programmed in the loosely-typed C++ programming language and was run on top of an operating system that does a poor job of enforcing memory safety—besides running applications in separate address spaces.

We strongly believe that the advantages of a strong-typed programming language and safety-enforcing run-time system can be measured in a genuine improvement of the code quality and a real improvement of its dependability: Programs are more reliable, because several errors are already caught at compile-time, while the run-time system enforces type and memory safety during the execution.

We appreciate that this argumentation has some weaknesses: We found ourselves in the impossible situation to formally prove this theory, as all of the studies on code quality metrics serve only the task of showing how difficult is to find a universal taxonomy, an all-encompassing framework that satisfies every requirement, covers each aspect and evaluates all.

1. This convention can be considered the weakest link in the model, as it introduces a subjective and questionable measure variable.

However, empirical evidence shows that *there is* a difference in the systems' reliability, and this discrepancy cannot be entirely due to the diversity between the two platforms, for such a simplification would contradict several measurement metrics. Furthermore, a non-negligible clue towards the correct interpretation is given by the breed of errors observed on the Pentium system, which had mostly to do with memory faults, such as dangling pointers, out-of-range fetches and memory leaks.

VII. Conclusions

In this paper, we considered the design and the deployment of a service robot as a practical application domain to substantiate the advantages of using strong-typed programming languages and paradigms in the development of safety-critical applications. This application domain is particularly well-suited to the discussion, for mechatronic applications can be considered critical systems—i.e. systems whose failure may endanger missions, lives and society—thus making their reliability a prime concern. Our results reinforce the conviction that strong-typed programming languages and paradigms play a major role in enforcing system safety.

The commonly used argument against languages that are type-safe is the inefficiency of the produced code. This misconception can easily be refuted. In the case of static type checking all restrictions are computed by the compiler: There is, therefore, no overhead in the code to be executed. Additionally, static typing allows for more aggressive utilization of optimizations such as reaching-definitions, and common subexpression elimination. Whereas each use of an aliased variable, as allowed in unsafe languages through pointer-arithmetic, forces the optimizer to invalidate all of the assumptions previously taken. 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 compensates the penalty paid in the execution time. In fact, there is no trade-off for letting a type violation happen during run-time.

The RoboX project represents a milestone in the field of mobile robotics: For the first time ever, eleven interactive mobile robots were operating for a relative long period in a work-space shared by human beings. We analyzed the results of 159 days of operation at the *Robotics* exhibition within Expo.02 and considered the amount and the nature of software and hardware failures. Notwithstanding the obvious differences between the two architectures driving the robot—i.e. a Motorola PowerPC and an Intel Pentium systems—and the difficulties in finding a comprehensive measurement metrics for code quality, the analogies in the code complexity on both platforms and the similar know-how of the whole development team led us to the conclusion that the reason for the *1-to-16* error ratio between the PowerPC and the Pentium should be ascribed to the features offered by programming languages and run-time systems that support strong-typing.

Lacking a formal proof, a rationale may be found in the words of Prof. Niklaus Wirth, who summarized the advantages of strong-typing by advocating that "the basic principle behind the concept of strong-typing relies in the introduction of redundancies, which verify the consistency of the code operating on the data—akin to parity checking. This consistency check should be performed, when possible, at compile-time: That is the reason of the keyword strong".

Acknowledgements

The *Robotics* project at *Expo.02* is the result of a very strong team effort: The Autonomous Systems Lab at the Swiss Federal Institute of Technology, Lausanne (EPFL) and various people from the academia and the industry contributed to its successful realization. The project has been funded by Expo.02 and EPFL. The eleven robots were produced by BlueBotics SA, a spin-off of the Autonomous Systems Lab, EPFL.

References

[1] M. Barbacci, M. H. Klein, Th. H. Longstaff and C. B. Weinstock. *Quality Attributes (CMU/SEI-95-TR-021)*, Pittsburgh, Software Engineering Institute, Carnegie Mellon University, 1995.

[2] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M. J. Merritt. *Characteristics of Software Quality*, New York, North-Holland Publishing Company, 1978.

[3] R. Brega, N. Tomatis, K. Arras, and R. Siegwart. The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study of XO/2 and Pygmalion, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Takamatsu, Japan, 2000.

[4] R. Brega, F. Wullschleger. A Personal Robot for Personal Robot Programmers—The Role of Automatic Storage Reclamation and Programming Languages in the Lifetime of a Safe Mechatronic System. In *Proc. of the IEEE International Conference on Advanced Intelligent Mechatronics*, 8–11 July 2001, Como, Italy.

[5] R. Brega. Safety ~~vs.~~ Speed. In *Proc. of the MSy'02 Embedded Systems in Mechatronics*, October 3-4, 2002, Winterthur, Switzerland.

[6] R. Brega. *A Combination of System Software Techniques Aimed at Raising the Run-Time Safety of Complex Mechatronic Applications*. Dissertation ETH Nr. 14513, Zürich, 2002.

[7] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, 17(4), pages 471–522, December 1985.

[8] M. Corti, R. Brega, Th. Gross. Approximation of Worst-Case Execution Time for Preemptive Multitasking Systems. In *Proc. of the ACM SIGPLAN LCTES'2000, Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 18, 2000, Vancouver B. C., Canada, Lecture Notes in Computer Science 1985 (LNCS 1985), Springer Verlag.

[9] M. S. Deutsch and R. R. Willis. *Software Quality Engineering: A Total Technical and Management Approach*, Englewood Cliffs, Prentice-Hall, 1988.

[10] Michael W. Evans and John Marciniak. *Software Quality Assurance and Management*, New York, John Wiley & Sons, Inc., 1987.

[11] Ch-W. Frei. *Fault Tolerant Control Concepts Applied to Anæsthesia.* Dissertation ETH Nr. 13599, ZH, 2000.

[12] M. H. Halstead. Elements of Software Science, *Operating, and Programming Systems Series,* Volume 7, New York, Elsevier, 1977.

[13] M. Honegger and A. Codourey. Redundancy Resolution of a Cartesian Space Operated Heavy Industrial Manipulator. In *Proc. of the International Conference on Robotics and Automation*, Leuven, Belgium, 1998.

[14] C. Jones. Software Metrics: Good, Bad, and Missing, *Computer*, 27(9), pages 98–100, September 1994.

[15] O. Lehrmann-Madsen, B. Magnusson and B. Möller-Pedersen. Strong Typing of Object-Oriented Languages Revisited, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP),* pages 140–150, ACM Press, New York, NY, USA, October 1990.

[16] J. J. Marciniak. *Encyclopedia of Software Engineering*, pages 131–165, New York, John Wiley & Sons, 1994.

[17] Th. J. McCabe and C. W. Butler. Design Complexity Measurement and Testing, *Communications of the ACM*, 32(12), pages 1415–1425, December 1989.

[18] Th. J. McCabe and A. H. Watson. Software Complexity, *Crosstalk, Journal of Defense Software Engineering*, 7(12), pages 5–9, December 1994.

[19] H. Mössenböck, *Object-Oriented Programming in Oberon-2*, Springer Verlag, 1995.

[20] P. Oman. HP-MAS—A Tool for Software Maintainability, *Software Engineering (#91-08-TR)*, Moscow, Test Laboratory, University of Idaho, 1991.

[21] P. Oman. and J. Hagemeister. Constructing and Testing of Polynomials Predicting Software Maintainability, *Journal of Systems and Software*, 24(3), pages 251–266, March 1994.

[22] P. Szulewski, et al. *Automating Software Design Metrics*, Rome, Air Development Center, 1984.

[23] C. A. Szyperski. *Insight ETHOS: On Object-Orientation in Operating Systems*. VDF, 1992.

[24] N.Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K.O. Arras and R. Siegwart. Designing a Secure and Robust Mobile Interacting Robot for the Long Term. *IEEE International Conference on Robotics and Automation*, Taipei, Taiwan, 2003.

[25] V. Vuskovic, M. Kauer, et al. Method and Device for In-vivo Measurement of Elasto-Mechanical Properties of Soft Biological Tissues. *Machine Graphics & Vision International Journal*, 8(4), 1999.

[26] A. H. Watson and Th. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. 1996.