# Design, Implementation, and Evaluation of the Resilient Smalltalk Embedded Platform

Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund

*OOVM A/S*
*Ellevej 2*
*DK-8310 Tranbjerg J*

Toke Eskildsen, Klaus Marius Hansen, Mads Torgersen

*Computer Science Department, University of Aarhus*
*IT-Parken, Aabogade 34*
*DK-8200 Aarhus N*

---

**Abstract**

Most microprocessors today are used in embedded systems and the percentage of microprocessors used for embedded systems is increasing. At the same time development of embedded systems is very resource-consuming among other due to the lack of support for incremental development and for support for dynamic servicing and upgrading of deployed systems. This paper introduces the design and implementation of the *Resilient System* for embedded systems development which has as a design goal to support exactly this. Programs are written in a dialect of Smalltalk and executed on a compact, efficient virtual machine running on embedded systems. Programmers may connect to running virtual machines and service, monitor, or change the running systems. Furthermore, we present an evaluation of the Resilient platform in relation to the design goals through a case study of two development projects which successfully used the platform.

*Key words:* Virtual Machines, Smalltalk, Evaluation, Case Study

---

*Email addresses:* jakob@oovm.com (Jakob R. Andersen), lars@oovm.com (Lars Bak), steffen@oovm.com (Steffen Grarup), kasper@oovm.com (Kasper V. Lund), darkwing@daimi.au.dk (Toke Eskildsen), klaus.m.hansen@daimi.au.dk (Klaus Marius Hansen), madst@daimi.au.dk (Mads Torgersen).

## 1 Introduction

More than 90% of microprocessors produced today are used in embedded devices, ranging from dishwashers, automobiles, to mobile phones. The embedded industry each year spends more than 20 billion dollars developing and maintaining software in these products [1]. Development of software for embedded devices has traditionally been very cumbersome: source code is compiled and linked on a development host, whereupon the resulting binary image is transferred as a whole onto the actual device, typically into flashable memory. If the source code is changed, the entire process must be restarted. Making a change effective can thus easily take several minutes, severely limiting development productivity. This is problematic in an industry where software development and testing already comprises more than 50% of R&D budgets [1].

Another problem facing the embedded industry is the lack of serviceability. Software content in embedded devices doubles every two years, making exhaustive testing virtually impossible [2]. Deployed products will inevitably contain software errors, leading to expensive recalls. As an example, the telecommunications industry currently spends as much as 8 billion dollars each year fixing faulty handsets [3]. Debugging and profiling is sometimes supported during development, typically through instrumentation of code, making a debuggable system larger and slower than a non-instrumented version. For this reason such support is removed in deployed devices, making later error diagnostics exceedingly difficult.

Similarly, software content in embedded devices often requires updates in the form of upgrades. In a static software model, this is commonly accomplished by downloading and re-flashing the entire binary image. In a more dynamic software development model where component code can originate from several locations, such updates become extremely hard to administrate.

Finally, source code is typically highly platform-specific, and written in unsafe, low-level programming languages such as C or assembler. As a result, reusability is limited and software development requires a large degree of expertise regarding the particular target platform and its low-level details.

The Java™ 2 Micro Edition (J2ME) has been proposed as a solution to some of these issues [4]. It comes with a debugging interface, but this is again commonly removed due to space concerns. Furthermore, the dynamic code loading model in J2ME is severely limited compared to its Java 2 Standard Edition (J2SE) counterpart. Finally, the runtime environment for J2ME requires more memory than what is available on most low-end embedded systems.

Future pervasive computing scenarios in which a large number of embedded devices are communicating will require a much more flexible software model, allowing for dynamic and changing functionality over time [5]. In the face of this,

the *Resilient System* has been designed and implemented. The design goal for the Resilient System is to have an always running serviceable platform for embedded devices with the following characteristics:

**Incrementality and serviceability:** The platform should support incremental programming enabling rapid development and full product serviceability.

**Accessibility and reuse:** Furthermore, it should be accessible to non-expert programmers, by being based on a safe, high-level programming language with easy support for reuse of platform-independent code.

**Low resource consumption:** Finally, the platform should be compact enough to be useful on low-end embedded devices, including system-on-chip (SOC) solutions, yet efficient enough for most real-time applications.

In this paper we present the design of the Resilient platform as driven by these goals. Furthermore we evaluate the extent to which this was successful, based on experiences from two development projects using the Resilient platform, both conducted in a cooperation between academia and industry.

### *1.1 Contributions*

The expected contributions of this paper are:

- to introduce the innovative design and implementation of the Resilient System which provides incremental development, dynamic upgrading, and serviceability in embedded systems development and
- an evaluation of the Resilient System in terms of its design goals and of the relevance of the design goals to the development of two embedded systems.

### *1.2 Paper Structure*

The rest of this paper is structured as follows: Section 2 presents the design and implementation of the Resilient System in terms of its associated programming language (Section 2.2), embedded platform (Section 2.4), and programming environment (Section 2.5). Next, Section 3 presents our evaluation of the Resilient System through the conduction of two projects; the “Digital Speakers” project (Section 3.1) and the “LIWAS” project (Section 3.2). Section 3.3 discusses the experiences of using the Resilient Systems and finally Section 4 concludes.

## 2 Design and Implementation of the Resilient System

### *2.1 Overview*

The complete Resilient System comprises a development environment running on a PC and a running Resilient Virtual Machine deployed on an embedded system or run locally. Figure 1 shows a deployment view of the Resilient System using a Unified Modeling Language (UML; [6]) deployment diagram. The Resilient Program

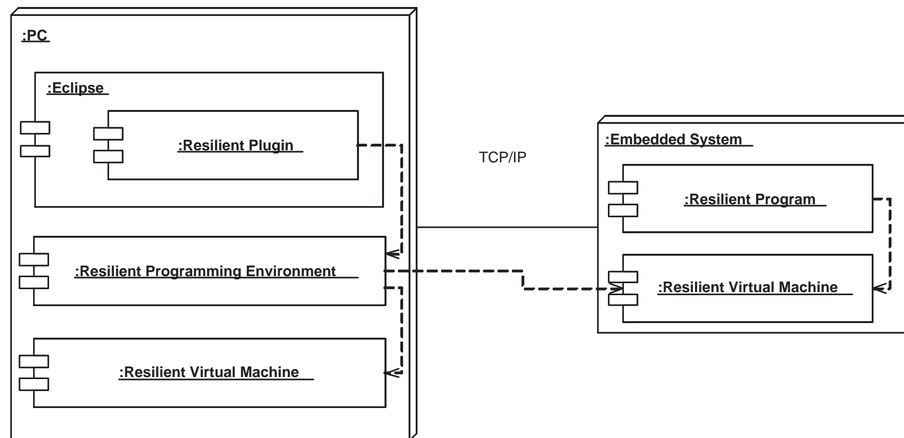


Fig. 1. Deployment Overview of the Resilient System

Environment, accessed using Eclipse with Resilient plugins, communicates with the running Resilient Virtual Machine using any available network connection.

The virtual machine connects to the underlying operating system (if available) in order to establish network connections. If the virtual machine is running without an underlying operating system, a TCP/IP networking stack running on the virtual machine is used.

In order to create snapshots for booting and deployment, the programming environment uses a local virtual machine. Due to the platform independence of the Resilient code, the local virtual machine is independent of the target platform for the snapshots.

Having created an initial snapshot, the developer transfers it to the target hardware and starts the virtual machine. Incremental development is performed by connecting to the virtual machine from Eclipse, which controls debugging as well as updating of classes and evaluation of code on the target virtual machine.

Changes to the running system are not persistent, so the creation and transfer of

a new snapshot is necessary if the changes are to survive rebooting of the virtual machine.

The following sections describe the design of the Resilient System in detail.

### 2.2 *The Resilient Programming Language*

The Resilient programming language is a dialect of Smalltalk [7] designed for simplicity, compactness, and performance. Smalltalk was chosen for several reasons: Smalltalk is a simple, dynamically typed, object-oriented programming language; everything is an object and behavior is described as message sends between objects; Smalltalk has proven ideal for supporting incremental program modification in that a programmer can freely modify a program without the need for recompiling and restarting the application; and most Smalltalk systems use a snapshot model allowing the same program execution to survive for years.

The Resilient programming language differs from Smalltalk in the following ways:

- Full syntax for classes is provided.
- Last-In-First-Out (LIFO) blocks are enforced.
- An atomic test-and-store statement for synchronization is introduced.
- A namespace hierarchy for modularization of libraries is introduced.

Traditionally, Smalltalk systems have forced programmers to use the integrated programming environment for all program manipulations. Methods are the unit of manipulation and for that reason a full syntax for classes does not exist. We introduce a full syntax for classes to allow programmers to use standard tools for program manipulation and source control management. The class syntax has been inspired by the syntax for Self [8]. The example below shows the source code for `Mutex`, a class that implements a simplified lock structure.

```
Mutex = Object (
  | owner |
  "acquire the lock prior to evaluating the
   block and then release the lock"
  do: [action] = (
    ["repeat the atomic test and store until it succeeds"
      owner ? nil := Scheduler current
    ] whileFalse: [ Scheduler yield ].
    action value.
    owner := nil
  )
)
```

Most object-oriented systems provide high-level synchronization mechanisms as part of the programming language [9] or as prebuilt data structures [10]. Instead, we have introduced a very low-level and simple synchronization mechanism; an atomic test-and-store statement. Advantages of this approach are minimal virtual machine support and a very flexible building block for implementing a wide range of high-level synchronization mechanisms. The above `Mutex` example uses the atomic test and store statement to busy wait for exclusive access. In the example, the current thread is computed, and then we atomically compare and do a conditional store (`owner ? nil := Scheduler current`). The instance variable owner is compared to nil, and if they match the current thread is assigned to it. The boolean result of the statement indicates whether the atomic test-and-store succeeded.

In order to support independently developed program parts, a solution for preventing name collision has been introduced. Resilient provides a simple form of namespaces. Any class can act as a namespace. For instance, the class describing elements in a list resides inside the List class. Classes in two different namespaces will therefore not be subject to name collisions.

### *2.3 Typed LIFO Blocks*

Creating blocks in Smalltalk has always been a potential source of performance problems. Blocks might survive the lifespan of the creating activation forcing the underlying implementation to heap allocate not only the blocks themselves but often also the method invocations in their scope. This is expensive and also stresses the memory management system. In Resilient, we have restricted blocks to be last-in-first-out (LIFO), which means a block cannot survive the creating activation. This allows Resilient to stack allocate blocks, thereby eliminating most costs associated with block creation.

To guarantee this behavior, we have introduced a type declaration for blocks: square brackets around a formal parameter specifies that it is a block. An example is the parameter `action` in the above `Mutex` class. This separation between objects and blocks makes it straightforward for the byte-code compiler to statically enforce LIFO behavior, by preventing blocks from being stored in objects or used as return values.

The graph in figure 2 on the following page shows the execution time of a simple, recursive, block-intensive micro-benchmark on a number of Smalltalk virtual machines. The benchmark constructs a linked list and uses blocks and recursion to compute its length:

```
Element = Object (
  | next |
```

```
  length = (
    | n |
    n := 0.
    self do: [ :e | n := n + 1. e ifLast: [ ^n ]. ].
  )

  do: [block] = (
    block value: self.
    next do: block.
  )

  ifLast: [block] = (
    next isNil ifTrue: [ block value ].
  )
)
```

It follows from the implementation that the micro-benchmark allocates at least one block-context per level of recursion, and that the non-local return in the `[ ^n ]` block must unwind at least as many contexts as the length of the linked list.

The graph shows that the execution time is linearly dependent on the recursion depth for all virtual machines. It also shows that enforced LIFO blocks makes our virtual machine almost 78% faster than the virtual machines for Squeak and Smalltalk/X, when it comes to interpreting block-intensive code. Better yet, our interpreter outperforms the just-in-time compiled version of the Smalltalk/X system by more than 16%.

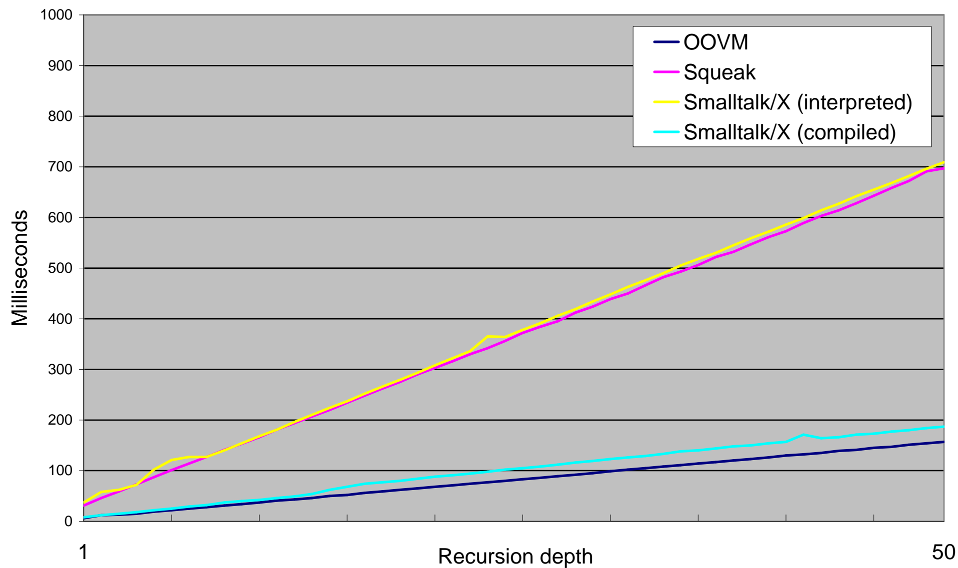


Fig. 2. Execution time for block-intensive micro-benchmark

Of course this approach has some language implications. On the negative side, the purity of the standard Smalltalk "everything is an object" credo is somewhat hampered with two static types rather than one. In practice the generality of standard Smalltalk blocks is rarely used, but there are a few (we know of two) common situations where it would be natural to store blocks inside objects in order to dynamically parameterize these objects with behaviour. One example is sorted collections, which should be parameterizable with the comparison operator to use for the sorting. The other example is Graphical User Interface (GUI) widgets like buttons, which should be able to store a callback function for when the button is pressed. In neither case can Resilient LIFO blocks be used, and one must instead apply the "function object" technique from, e.g., Java Comparators, where a full object is supplied, implementing the desired behaviour (comparison or callback) as a method.

From a language point of view it should be noted that general blocks themselves are not without problems, especially in light of the non-local return mechanism. Non-local return makes the block return to its creating context rather than its calling context, but that is meaningful only when the creating context is still on the stack, and otherwise gives rise to a runtime error (after possibly having peeled apart the whole stack in search of the missing activation). This undesirable situation is naturally prevented in Resilient, so the added static typing of blocks does in fact have an error-preventing as well as an efficiency-related effect.

LIFO behaviour also means that blocks can never be transferred between concurrent threads, avoiding the similar issue of what to do in the case of non-local returns to a different stack. All in all we think that this language restriction has important semantic advantages along with the efficiency gain, and that the loss of expressiveness is a minor problem in practice.

### *2.4 The Embedded Platform*

The embedded platform is based on a small object-oriented virtual machine. All software components are compiled to safe, compact bytecodes and executed on top of the virtual machine. The compactness makes it possible to fit the virtual machine, core libraries, device drivers, TCP/IP networking stack, and user applications in less than 128KB of memory.

The embedded platform can be configured to run directly on hardware without the need for an operating system. This accommodates for the most resource constrained devices, onto which it is impossible or impractical to fit a full operating system. However, it is also possible to run the embedded platform on top of existing

embedded operating systems, such as Embedded Linux [1] or Symbian OS [2]. This option is useful in projects that depend on existing applications or device drivers.

Software components running on top of the virtual machine are mostly platform independent. The virtual machine bytecodes themselves are independent of the hardware on which they run, so that it is possible to build applications that run on any device equipped with the embedded platform. However, the virtual machine allows device drivers to be implemented on top of it, and therefore some software components may depend on external input/output devices available only on some hardware platforms. The example below briefly illustrates how a sample device driver for the Intel StrongARM general-purpose I/O (GPIO) module might be implemented:

```
GPIO = Object (
  | io |

  initialize = (
    io := Memory at: 16r90040000 size: 16r20.
  )

  setOutput: pins = (
    io longAt: 16r08 put: pins.
  )

  clearOutput: pins = (
    io longAt: 16r0C put: pins.
  )
 )
```

The example also illustrates the most common way of providing device access, viz., by mapping device address spaces as part of system memory space. Access to memory-mapped I/O is provided through external memory proxy objects (here through `Memory` objects). The virtual machine makes sure that are driver cannot allocate a proxy that refers to the object heap, thereby possibly corrupting the heap. Interrupt requests from devices are reified as *signal objects* by the Resilient system software; device drivers consequently handles interrupt requests at the level of these objects.

Software components running on top of the virtual machine are fully serviceable through a reflective interface in the embedded platform. The reflective interface allows the programming environment running on the developer's personal computer to inspect the state of the running system and possibly change it. Section 2.5 gives

[1] http://www.embedded-linux.org
[2] http://www.symbian.com

more details on the connection between the programming environment and the embedded platform.

Most of the reflective interface is implemented in the Resilient Programming Language itself as a component running on top of the virtual machine. The virtual machine provides hooks for manipulating breakpoints and threads and for inspecting and changing classes, methods, and objects. The communication protocol for reflection is handled by a service implemented entirely in the Resilient Programming Language. The reflective interface is always available, even on devices deployed at customer sites. The result is that developers can debug, profile, and update running code on any device that runs the embedded platform.

The embedded platform guarantees predictable response times. It supports scheduling of interrupt handlers and time critical tasks. The non-disruptive, real-time garbage collector handles all resource management in the background. Furthermore, the virtual machine prevents user code from performing malicious operations, and as such it provides a secure, device-independent platform for real-time software execution and delivery.

### *2.5 The Programming Environment*

The Resilient programming environment is written as a plug-in to the Eclipse extensible IDE [3]. The Eclipse framework provides well-known methods and abstractions for managing projects and browsing and editing source code. The Resilient plugin provides the Resilient-specific components, such as source code compiler, debugging, and the reflective connection to a running embedded platform. Because the IDE is based on the industry-standard Eclipse framework, developers are able to transfer existing knowledge and practices of software development into the realm of embedded development.

An important part of the Resilient development model is the ability to connect the IDE to a running embedded platform. When connected, the IDE is able to inspect and make changes to the object model on the running platform. The connection is made using the network stacks present on the embedded platform, such as standard TCP/IP. The reflective interface on the embedded platform allows the IDE to inspect, pause and terminate running threads, inspect and modify objects in the object heap, and update code on the running platform.

The reflective interface is key to the Resilient way of developing software. Since developers can evaluate chunks of code on the target device, and upload changes while the system is running, it is possible to immediately see the effects of changes

[3] `http://www.eclipse.org`

in source code. Debugging is also easier, since it is closely coupled with the source code development.

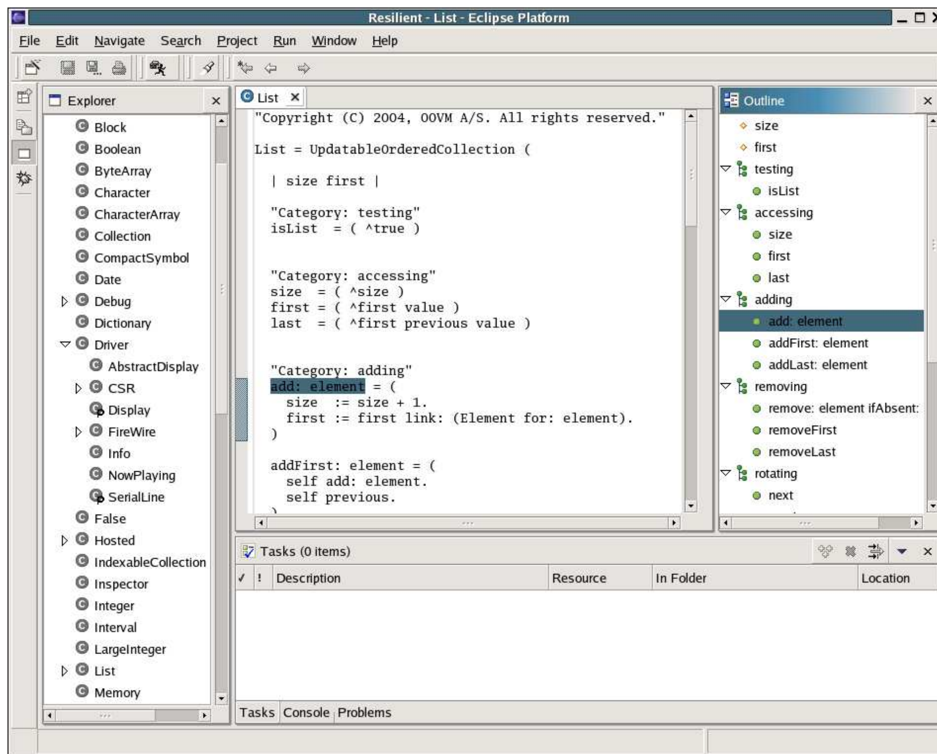


Fig. 3. Resilient IDE with a sample project open

Classes in Resilient are organized in namespaces, which may be nested, providing a means for grouping of related classes. The Resilient IDE, shown in Figure 3, offers advanced features for navigating and searching the source code.

## 3 Evaluation

This section relates the experience of two research projects – both including academic as well as industrial participation – in which the Resilient platform was used. One project was aimed specifically at evaluating the platform, whereas the other uses Resilient as a vehicle for industrial prototype development. We first introduce the projects and then discuss experiences of using the Resilient System.

### *3.1 The "Digital Speakers" Project*

The purpose of this project was to evaluate the Resilient platform in an industrial setting, targeting as a test case the next generation digital speakers of Bang and Olufsen (B&O) [4], which are to be connected using the IEEE1392 Firewire standard.

The virtual machine was ported to a development board with a number of Firewire-related hardware components. A prototype "speaker" was set up by connecting the board to a Firewire source (in this case a laptop) and a digital audio destination (in this case a stereo). In a production speaker, the Firewire source would be a B&O stereo and the digital audio destination would be a digital amplifier and a number of speaker units, boxed up with the board as an active speaker.

A programmer with no previous experience in embedded software was employed to develop a full test application ranging from low-level platform-specific Firewire driver code to a high-level platform-independent webserver component.

### *3.2 The LIWAS Project*

The LIWAS project is a three year research and development project aiming at producing a framework, *Ex Hoc*, for hybrid communication in sensor networks [11]. The initial use scenario is a network of communicating sensors for measuring road condition, i.e., whether the road is dry, icy, snowy, or rainy. This scenario has applications in traffic safety as well as for resource consumption in connection to road maintenance.

The sensors will be deployed in cars and on stationary road signs along roads. This will allow for ad hoc networking among cars [12] in combination with centralized communication through either Internet-connected road signs or mobile gateways in the form of mobile phones in cars. The communication will, e.g., allow cars to be warned by passing cars if the road conditions are problematic further ahead.

The current status is that the first communicating mobile sensor system with a very simple dissemination protocol is built and tested. Figure 4 shows a deployment view of Ex Hoc. The major part of the software on the *Mobile Unit*, viz. the *Communication System* is written using the Resilient System. It currently runs on a CerfCube [5] board with an XScale processor.

The *Sensor System* is mainly low-level microcontroller code written in C. The *Sen-*

---

[4] `http://www.bang-olufsen.com`
[5] `http://www.intrinsyc.com`

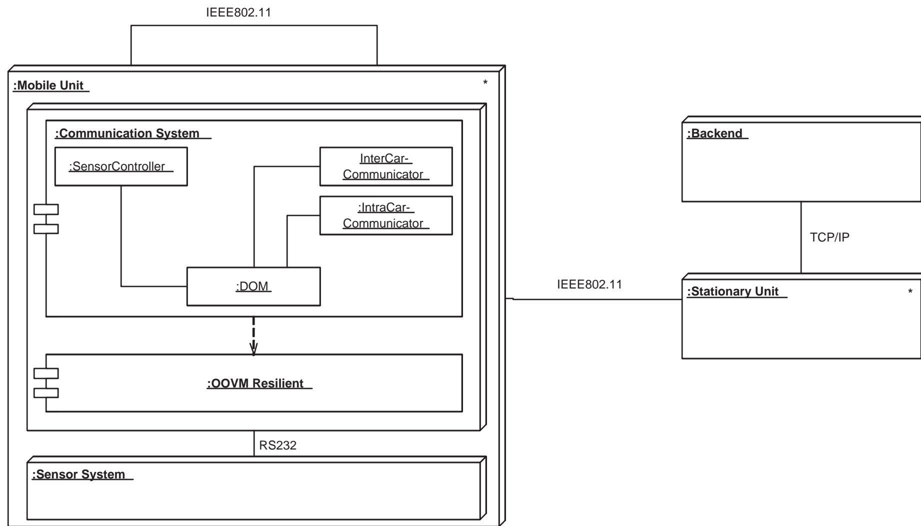


Fig. 4. Deployment View of Ex Hoc

*sor System* interfaces to the various sensors needed for reliably classifying road conditions. The *Sensor System* delivers measurements 36 times per second (which is equivalent to each meter at 130 kph), each of which needs to be classified by the *Domain Object Model (DOM)*. The ad hoc communication puts real-time constraints on the system as well since among others classifications need to be communicated among cars passing each other at high speed. The *Stationary Unit* contains software analogous to that of the *Mobile Unit*.

The implementation of the communication system of the sensors is done using the Resilient System by two programmers with little experience in embedded software. The development is done in collaboration with the private company LIWAS A/S [6] which is also responsible for the necessary hardware construction.

### *3.3 Experiences*

This section presents and discusses experiences from the "Digital Speakers" and LIWAS projects in relation to their use of the Resilient System as a basis for implementation.

[6] http://www.liwas.com

### *3.3.1 Batch and Incremental Development*

The "speaker" application was the first major application to be developed on the Resilient platform, and the project took place while the platform was still being developed and thus suffering from a number of bugs and flaws. One of these turned out to be a major show stopper: a problem with the serial line meant that for some time the platform could not survive receiving incremental program modifications. Thus we were temporarily forced into the batch-oriented style more conventionally used for device programming, and inadvertently ended up being able to compare it with the incremental development model of the Resilient system.

The incremental mode of work proved a huge benefit to productivity. The ability to query and modify the state of the running device (by sending code to it – at the time the debugger was not yet functional), and to modify the running code for programming or test purposes gave rise to a very tight development loop. Compared to desktop development environments the immediacy and integration of the code-test-run cycle on the Resilient platform was found to be more in line with dynamic systems such as conventional Smalltalk than the sopmewhat slower turnaround of static systems such as Java and C++. The experiences in the LIWAS project concur with this.

### *3.3.2 Dynamic Update in Development*

An example of the power of serviceability comes from the LIWAS project: part of the prototyping work has been to equip a trailer with among others light reflection, road temperature, dewpoint, and air temperature sensors. This was done to collect as much data as possible on as varied road conditions as possible in order to be able to create a suitable classification algorithm. While using the trailer with measurement collection and classification controlled by software written in the Resilient System, it was possible to connect to the running Resilient System with a PC running the Resilient programming environment. This was very useful among others in deploying the system, since it was possible to monitor the system, see what was wrong if anything, and possibly fix the problem online.

One problem encountered during development of the LIWAS application was the inability to update the running code for a thread handling serial communication. By refactoring the code to a tight loop containing a method invocation, we sidestepped this limitation. This experience lead to the standard practice of factoring all looping threads.

A probable future use of the incrementality and serviceability in the LIWAS project will be the dynamic update of the classification part of deployed systems. Currently, we are testing two different strategies of classification: one based on a physical model and one based on a statistical classification. It may possibly turn out that the best strategy for classification will change after the LIWAS system has been

deployed in a number of test installations and that the type of change is one that requires major code changes which cannot be parametrized in any single algorithm. The Resilient System contains the main functionality for this, but probably defining a component model (which enables developers to make deployable, versionable collections of classes and objects) will be necessary on top of this.

### *3.3.3 The Resilient Programming Language*

Within the resources of the projects we did not have the opportunity to directly compare programming in Smalltalk with solving an equivalent task in a traditional embedded programming language like C or C++. We have however written both low level (e.g. the FireWire driver in the Digital Speakers project) and high level (e.g. the webserver in the Digital Speakers project) code.

At the low level, Smalltalk lacks the ability of C-like languages to directly map declared datastructures onto the very specific bit layout of data in memory which is often characteristic of low level programming. The abstraction mechanisms of Smalltalk are strong enough, however, that it is straightforward to represent the individual data components symbolically by means of accessor methods.

In the “speakers” application, the construction and decoding of FireWire packages was as simple and elegant as any C version, the bit manipulation localized to a few package-specific methods. For parts of the FireWire code, we did indeed have C source available for comparison. In conclusion, writing driver software and related low level programming presents no special challenges in the Resilient Programming Language.

At the high level, the use of object-oriented abstractions is a well documented advantage, which was also heavily leveraged in the project. A similar effect could probably have been achieved with the more traditional C++. However, apart from being far from platform independent, C++ as a compiled language inherently lacks the possibility of incremental update.

The combination of platform independence and object-oriented abstraction meant that the whole web server component, first developed on top of Linux, could be migrated to the evaluation board with only one minor source code adjustment before the code was running again. Thus, the on-the-board development philosophy is balanced by the possibilty of writing extensive parts of an application off-board if necessary. Moreover, this suggests the possibility for reuse of large amounts of application code across different platforms, something which is rarely seen in the embedded world.

In the LIWAS project, a “proof-of-concept” prototype was first developed on a Resilient System running without operating system. Using Smalltalk, it was possible to implement a special-purpose serial driver which hooked into the running Ex Hoc

system with miminal overhead. The system was later ported to Embedded Linux for further prototyping since a large number of drivers had to be used including Bluetooth, USB, and IEEE802.11 drivers. Except for the driver part, there was little code that had to be removed or replaced. Currently, we are primarily developing directly on the CerfCube, but also developing on a version of the Resilient System running on Microsoft Windows XP without any code changes being necessary between the setups. The end goal is that the system (i.e., drivers) will be fully implemented on a version of the Resilient System running without an operating system.

#### *3.3.4 Interpretation and Performance*

With a purely interpreted architecture, certain performance characteristics are to be expected. Bytecode representations of code are generally compact, whereas compiled code is several times more memory consuming. This is the reason for having Just-In-Time (JIT) compilers in many systems in order to compile only the most used code. That code will be doubly represented however, and more importantly in a small system, the JIT compiler itself takes up considerable space. But compiled code undoubtedly does run a lot faster.

The running “speakers” application, including the VM itself, webserver, firewire driver, TCP stack, etc., fitted comfortably within the 128K RAM available on the development board. There were no real-time constraints in the functionality, and therefore little opportunity to evaluate efficiency. Engineers at B&O estimated that the Resilient VM might have a hard time keeping up with e.g. real-time filtering of audio streams, at least on affordable hardware. For this purpose, interfacing to compiled or assembler code would be crucial for the performance-intensive parts of the code.

The LIWAS application on the other hand is inherently a real-time application in which the system has to handle a set of measurements 36 times per second and make a classification based on this for each set of measurements. Our current implementation clearly supports this also with the statistical classification. And even though the CerfCube board has a large amount of RAM available, only little is used, and the application would be able to be ported to a device with 256K RAM.

## 4 Conclusion

### *4.1 Incrementality and serviceability*

Incremental development with a smooth and tight integration of coding and running has always been a hallmark of Smalltalk and related dynamic programming

languages. A major contribution of the Resilient development model is to make this manageable on an embedded system, by separating out the reflective parts of the running program onto a separate IDE on a different platform.

Experience from both development projects shows that the Smalltalk programming feel does indeed carry over to the embedded world, where it is an even greater relative advantage, because the traditional batch-style alternative here is so much more costly than in a desktop environment.

Where the update of running code in a desktop Smalltalk environment is a possibility, in Resilient it is almost inevitable. This brings to focus some of the standard problems in this area, most notably the fact that code has to be structured in anticipation of later changes.

### *4.2 Accessibility and reuse*

Traditional embedded programming tends to require a great deal of platform specific expertise. Not only do programs have to be written in platform-dependent dialects of assembler, C or C++, but the tools used for testing and debugging are highly platform-specific and often even require special hardware. With the Resilient system, the execution semantics as well as the development tools are independent of the execution platform, as long as a virtual machine has been ported to it. Of course, drivers etc. must be implemented specially for each device, but the object-oriented abstraction mechanisms of Smalltalk help encapsulating and isolating these parts. Thus, code reuse across platforms becomes a real possibility.

The projects have confirmed this situation. Programmers with no embedded experience or device-specific knowledge have found it easy to develop applications on the system, and cross-platform reuse has also been employed successfully. Compared to standard Smalltalk, the LIFO restrictions on blocks have proven a nuisance at some points, but generally the language restrictions of the Resilient Programming Language have not been too inhibitive.

### *4.3 Compactness and efficiency*

Being something as unusual as a fully interpreted bytecode-based embedded platform, the Resilient system emphasises size constraints over speed considerations. The bytecode format, memory layout, and virtual machine itself are all optimized for compactness. That said, of course a lot of devotion has gone into running code as fast as possible. The block limitations are an example of that.

From especially the “speakers” project we can conclude that a good deal of software

fits onto a rather small device, although very low-end devices will be out of reach for the Resilient model. As for speed, we have no measurements, but can only conclude that in both projects it was fast enough for our purposes.

**Acknowledgements**

The academic part of this work has been supported by the software part of the ISIS Katrinebjerg competency centre [7]. We thank the industrial participants in the projects which evaluated the Resilient System.

**References**

[1] Wind River Systems, Inc., Investor Relations Presentation (September 2003).

[2] Wind River Systems, Inc., Changing Software Development in the Electronics Industry, Prudential Securities Technology Conference (October 2002).

[3] K. Belson, Beware of the Worm in Your Handset, The New York Times Technology Section (November 28, 2003).

[4] R. Riggs, A. Taivalsaari, M. VandenBrink, Programming Wireless Devices with the Java$^{TM}$ 2 Platform Micro Edition, The Java Series, Addison-Wesley, Reading, Massachusetts, USA, 2001.

[5] ISTAG, Scenarios for Ambient Intelligence in 2010, Tech. rep., European Commision - Community Research, ISTAG: European Commission's Information Society Technologies Advisory Group (February 2003).

[6] OMG, Unified Modeling Language specification 1.5, Tech. Rep. formal/2003-03-01, Object Management Group (2003).

[7] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts, USA, 1984.

[8] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Proc. of the OOPSLA-87: Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, FL, 1987, pp. 227–242.

[9] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison Wesley, 1997.

[10] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, K. Nygaard, Abstraction mechanisms in the beta programming language, in: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press, 1983, pp. 285–298.

[7] http://www.isis.alexandra.dk

[11] K. Hansen, T. Eskildsen, L. Kristensen, K.-D. Nielsen, R. Thorup, J. Fridthjof, U. Merrild, J. Eskildsen, The Ex Hoc Infrastructure - Enhancing Traffic Safety through LIfe WArning Systems, in: In Proceedings of Trafikdage 2004 (11th Danish Conference on Traffic Research), Aalborg, Denmark, 2004.

[12] E. Royer, C.-K. Toh, A review of current routing protocols for ad hoc mobile wireless networks, IEEE Personal Communication 6 (2) (1999) 46–55.