Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection

Jecel Mattos de Assumpção Jr * Sergio Takeo Kufuji †

Laboratorio de Sistemas Integraveis - University of Sao Paulo - Brazil

Abstract

Pure object oriented programming languages (like Self) present a very abstract view of the hardware to the programmer, and so are unlikely candidates for system software development. The Merlin operating system adds a reflective structure to Self that allows the programmer to move cleanly between different abstraction levels to get around this problem, but this introduces other difficulties in the form of circular definitions. This paper shows how a step by step introduction of reflection into the system can handle this apparently impossible situation.

1 The Problem

The Merlin Project[AJ93] was created to make computer systems simple enough to be used by novices, yet powerful enough not to be quickly outgrown by them. So it was natural to choose a simple and powerful language like Self[US87] to implement this object-oriented operating system. Most systems offer different tools, languages and even ideas to such groups as novices, power users, application programmers, system programmers and kernel hackers. This creates great barriers that limit the users' growth as their needs expand. So Merlin uses the Self language at all levels, arranging tools and ideas in a simple path of progressive disclosure. This ideal contrasts with the current Self implementation from Sun, which consists of 80 thousand lines of C++ code on top of a traditional OS written in C. Even the first step in the implementation of Merlin, the tinySelf interpreter, is mostly written in C. So our problem is how to evolve this into a system which is almost entirely described in terms of itself.

*jecel@lsi.usp.br

†kofuji@lsi.usp.br

2 Why?

Developing an application in a dynamic, integrated system like Smalltalk or Self is wonderful, but delivering the final program is nothing short of a nightmare. It is not hard to see why this is so: the application is actually just a small extension of the very rich programming environment. The ability to reuse all the resources that come with the system is one of the reasons why developing with these languages is so productive. The problem, of course, is that the lack of a clear separation between the application and the development system means that everything must be sent to the clients in order for it to work on their machines.

Smalltalk has had a great success in the last few years in the "mission critical" applications niche. Unlike those in the shrink-wrapped software market, these developers can afford to tell their clients to dedicate a computer (and a hefty one at that) to running a single program. The price of each sale is normally high enough that the cost of including a copy of the development system (typically in the thousands of dollars range) with each delivered system is not a significant factor. To reach a greater audience, however, the problems mentioned above must be addressed.

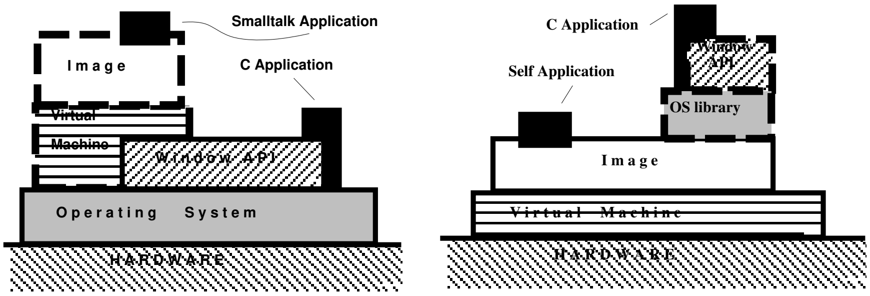
Self, like Smalltalk, is implemented in two parts: the Virtual Machine (VM) and the Image. The Image is basically a memory dump of all the objects in the system, including the full development environment, class libraries and the user's application. The VM is a normal program that makes all these objects run. Self 4.0's [1] VM is over 2MB in size, but the typical Image is over ten times as large. However, only a small fraction of all those objects might actually be needed by a given application so a solution is to "strip" the Image of this excess. Ole Agesen's extractor[AU94] for Self is currently the best example of this approach. If the application does use a large number of objects this solution will not help very much, but it would be a huge program in any language anyway.

Running several extracted applications at the same time would still be a problem as there would be multiple copies of common objects as well as a copy of the VM for each program. One solution would be to merge the applications into a single Image, but developers are not normally happy about distributing the source code (few Smalltalk implementations can handle binary object filing yet). If more than one VM must be loaded, it might be a good idea to move most of its functionality into dynamic linkable libraries which most operating systems bring into memory only once.

Figure 1 shows that a new viewpoint of the problem might lead to a better solution. A lot of the overhead mentioned so far could be eliminated by running the VM on the bare hardware - absorbing the operating system into the language implementation. This is not a new idea: most early Smalltalk implementations were exactly like this. They were rather conventional in their

[1] the Self 4.0 implementation for Sparc machines as well as all the papers of the Self group can be found by anonymous FTP to `self.smli.com`, or via the web at `http://self.smli.com`

use of disks organized in simple file systems. Simply filling the whole secondary store with the Image (and making good use of the virtual memory hardware in modern processors) would eliminate many complications by extending the object oriented world a step further. These persistent object stores[SKW92] are an area of active research, but are already an essential component of recent object oriented operating system designs[TYT92].



Current Situation **An Alternative**

Figure 1 – Implementation Approaches

The first objection to such an approach is that dynamic languages are too slow to be used in system level software. Discussions of "Adaptive Compilation" elsewhere[HU94][AJ94] show that this need not be true, so we will not consider the issue further. Another objection is that the proposed system is "monolingual" and unnecessarily restricts the programmer's freedom. Figure 1 shows how other languages, like C, might be integrated into such a system. The overhead will seem very large to the user (the boxes with the dashed lines in the figure - a POSIX and X11 emulation libraries, for example), just as the overhead of running Smalltalk or Self seems to be such a problem today. In any system there is a language (or class of languages) that is privileged (shell, shell-like and C in Unix), so it is just a matter of choosing which one.

One enormous advantage to this solution, which was chosen for the Merlin Object Oriented Operating System[2], is that everything that is in the Image part can be changed while the system is running. You can change the code that draws a window in an editor that uses that very code, for example. The danger of this should be obvious: a small mistake might render the system unusable, and so it would be impossible to recover from such a change. A good logging/backtracking mechanism is therefore a top priority. Given that, the productivity of "living inside" the system that is being developed can be tremendous, which might explain why the Smalltalk of the 70s had a more

[2]see `http://www.lsi.usp.br/~jecel/merlin.html` on the World Wide Web for more information about Merlin

advanced development system than most C++s of the 90s.

This utopia is ruined by the box labeled "Virtual Machine" at the bottom right of figure 1. Any change to that part of the system will send us right back to the C/Unix/gdb primitive programming environment and will require a tedious change/compile/link/reboot cycle to test it. The solution is to move functions from the VM into the Image to reduce the cases for which this problem occurs to the bare minimum. This will expose much more of the implementation to the system itself, which we call "reflection". All operating systems must deal with themselves in some way, and so are reflective. The difference is that most systems do this in an ad hoc way that is of no use to the programmers. In Merlin we propose to reduce the virtual machine by "adding reflection" in a structured way.

The next part of this paper explains how the final, reduced virtual machine will work. Next, message passing is described to show how reflection can add to the flexibility of the system. The fifth section shows how the system will evolve by adding reflection one step at a time. Finally, some conclusions about bootstrapping a reflective system are presented.

3 Reducing the Virtual Machine

Stanford/Sun Self includes objects called "mirrors" which allow programs to include "structural reflection". The programming environment, for example, uses this to add or remove slots from an object, or to allow the programmer to edit the code for a method. "Activations" give the program a structured view of the execution stack, allowing a certain amount of "behavioral reflection" [CMDM92]. Most of the implementation is hidden in the VM, however, and can only be accessed in limited ways through a set of primitive functions. An example of this is that the VM uses "maps" to optimize memory and time by factoring common elements for several objects. As the language defines each Self object to be independent, the VM makes this totally invisible to the user. When this information is needed (as in [AU94]) it must be laboriously recreated.

At the "base programming" level, an object in Merlin looks like a simple set of slots. Each slot has a name and a value. Sending a message with a slot's name makes the object fetch the corresponding value and execute the code (in the case of a method value) or simply return the value as the final answer. A special "assignment primitive" value has the effect of changing another slot. If an object "obj" has a slot "x" with a number as a value, and a slot called "x:" with the assignment primitive as its value, we could write "obj x: (obj x + 1)" to increment this number, for example. If an object's slot has no corresponding "assignment slot", then it is considered a "constant slot" as its value cannot be changed (at least not by normal message passing). A "map", then, is an implementation trick to save space - many objects differ only by the values of their non constant slots and should only store these while everything else is factored away into an invisible map.

Merlin's reflective structure is based on that of Apertos[Yok92]. Each object is associated with a "meta-space" which defines an execution environment for it. In Apertos, the meta-space is implemented by objects called reflectors. A meta-space also contains meta-objects which can be shared with other meta-spaces as needed. Merlin combines the notions of maps and reflectors into a single map/reflector object. In fact, whenever two or more logically distinct objects seemed to always go together, they were merged into a single object (even though this is not a very good thing to do in an initial design). The combined map/reflector object holds the information about the slots of an object (and objects cloned from it), the values of its constant slots and information about the meta-objects contained in this meta-space. The meta-objects can be found in meta-slots, which can be variable or constant in the same way as normal slots.

It is possible to change an object's slots (which would be its map) without affecting its meta-space (the reflector) by sending it the message _Define: (if this were not the case, programming exclusively at the base level would not be possible). The opposite is also possible by using the message _Migrate: (this operation moves the object to another meta-space - this might have the effect of changing its behavior or it might simply mean that the object is transferred from one node to another in a network). Even so, the combined map/reflector gives a more refined idea of "type" for the system so that the compiled code for a method for a normal binaryTree, for example, might be different from the code for a real-time binaryTree,

The message passing model in Merlin is a simplification of the "wait by necessity" model[Car90] where each message send returns a future object as an answer and allows the sender to proceed immediately. Every object is active and executes one message at a time from a queue. When a message returns a result, this replaces the associated future so all happens as if this result had been returned originally instead of the future when the message was sent, maintaining full compatibility with a sequential execution of the code. This means, of course, that a method must never "see" future objects, so it is blocked if it tries to send a message to one (when the value is returned the method is unblocked and the message send continues). Sending a message, therefore, results in the creation of two objects: the future and the message object to be placed on the receiver's queue. When the message is ready to be executed, a context (or activation) must be created and the contents of the message copied into it. Combining futures, messages and contexts into a single object can save this useless copying, though it wastes memory as contexts are larger than messages. Figure 2 shows some messages waiting to be executed by an object.

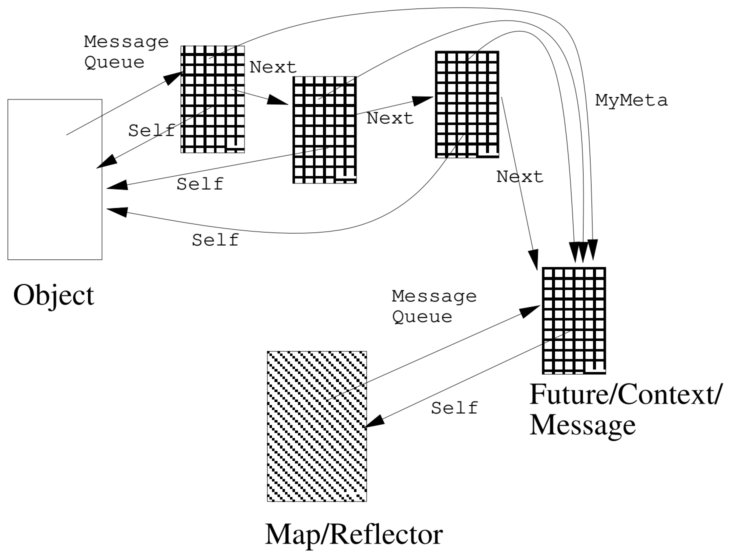


Figure 2 – Merlin's Reflective Architecture

A surprising detail in the figure is that the first three messages point to the object via their "self" slots, but the fourth one does not. The fourth future/message/context object is actually the current context of the map/reflector of the object. This is used as a sentinel to mark the end of queue. If the object's "message queue" meta-slot pointed directly to this context, then the queue would be empty and the object would be in the idle mode. In this case, the message queue points to the currently executing message/context and to two pending messages. If the current method needs to do a meta-operation (currently only sending a message and returning a value are possible meta-operations, unlike Apertos which has a richer set of primitives), it simply saves some minimal state in the current context object and loads from the object indicated by the "myMeta" slot. An "invokedBy" slot in the new context saves the value of the previous context so it may be restarted later.

A careful inspection of the figure will reveal that when the object's queue is emptied, it will "look exactly like" its map/reflector - both will have the same future/message/context in their "message queue" slots. They are, however, distinguished by use: objects are invoked by sending messages to them, map/reflectors by starting a meta-operation. Maps are objects too, and so have their own map (the "map map" object) as do the future/message/contexts. Each shaded object in the figure is the root of a structure like the one that is shown. Infinite regression is avoided with some circular references (the map of the "map map" is itself, for example).

The value of the "message queue" meta-slot must be contained in the first field of every object (except for small integers and floating point numbers), but all other information is defined by the map/reflector and may be changed. Some interesting meta-slots are memory (defines storage and garbage collection for the object), owner (to allow protected access to the object), mailer (to send messages) and dispatcher (handles incoming messages).

There is a very special object, called the "CPU object", for every node in a distributed system. It is basically what is left of the virtual machine after we have reduced it as far as possible. This object allows controlled access to resources such as the TLB in the MMU, the register set, the interrupt table and so on. The compiled method cache can also be found here, so that the CPU object is the only place in the whole system where machine language instructions are used.

4 Messages and How They Work

In Self message passing is at the bottom: everything that happens is initiated by a message send. The following pseudo-code shows what happens when a message is to be sent in Merlin. We suppose that the message receiver, the arguments and the selector have been stored in some predefined place (some registers, for example).

```
Sending The Message:

 1: switch to reflector/map ( actually, the mailer object )
 2: if receiver is future
        block and call scheduler
 3: else
        make new future/message/context object
 4:     if receiver is in other session
            trap to "kernel"
 5:     else
            put future/message/context in receiver's queue
 6:         switch back to calling context
```

Step 1 shows the transition from normal computing to meta-computing as described in the previous section. The rest of the code runs in the context of the mailer meta-object. The next step checks to see that we are not trying to send a message to a future object. In step 3 the future/message/object is created and loaded with the correct values. The execution environment is divided into several "sessions" (not described here) and sending messages between them requires extra protection. A normal message is simply inserted into the receiver's queue for later execution, and the sender is allowed to continue (with the newly created future as a temporary answer).

In the final, fully reflective, implementation of Merlin the mailer object will be written in Self so this code can be changed by the programmer. This might be done, for example, to be able to send messages with an associated timeout. Another possibility is to have a sender that avoids queuing a message if another one of the same type (like "refresh" in a window system) is already in that queue. The problem is that Self works by sending messages, so it seems we have a circular definition. This turns out to be false, however, unless a mailer is its own mailer, which is not normally the case. Even that case is possible if the compiler can inline away all message sends implicit in its code (this is very tricky to guarantee, but can be done).

Once the future/message/context is in an object's queue, we have the situation of figure 2 and the following happens:

```
Receiving The Message:

 7: object returns from previous message: move next
    future/message/context to the ready queue
 8: start dispatcher object ( inside the receiver's
    reflector/map )
 9: if meta-message
        patch context and switch to the right meta-object
10: else
        lookup the message selector
11:     start slot "reflector" apply code
```

Step 7 is actually the end of the execution of the previous message (unless the receiver was idle when this message was sent - in that case this step would be done as part of step 6). Eventually the scheduler will load this context into the processor, and its initial PC points to code that loads the dispatcher object. The following steps are actually executed inside the dispatcher object.

Reflective messages (indicated with names that start with the underscore character) are handled as a special case in step 9. Step 10 does the actual message lookup using the language's inheritance rules (once again, this may be changed by creating new dispatcher objects) and the result of the search is slot which has its type-dependent code executed (this code can also be changed by the user to create new types of slots). The last two steps are very close to what Moostrap[MC93] does, so we will not go into details here.

If a system were implemented as shown here, it would be so slow that it would not be usable at all. Fortunately, most message sends can be short-circuited by inlining by the compiler and/or with the compiled code cache. In fact, unless the programmer actually does create a customized message passing sequence, the ability to do so will not have a noticeable impact on performance. This should be the case even for messages to non local objects (which cannot be inlined) as they are so expensive to begin with.

5 Can we get there from here?

The system, as described so far, would be quite interesting except for one small detail: if it were running, it would work fine, but otherwise there would be no way to start it due to the mutual dependency of many components (including many not mentioned here). This is a classic "bootstrap" problem. If A depends on B, but B needs A to be working in order to run, we can introduce C that works exactly like B but does not need A to work. C can be made to run all by itself, and A can now run too as C looks exactly like the B it needs. We can then start B in parallel with C (as A is running this is possible) and make them switch places. We can drop the now useless C and get the system we originally intended. An example of this would have A = timed explosions, B = inertia and C = electric motor and battery in a car.

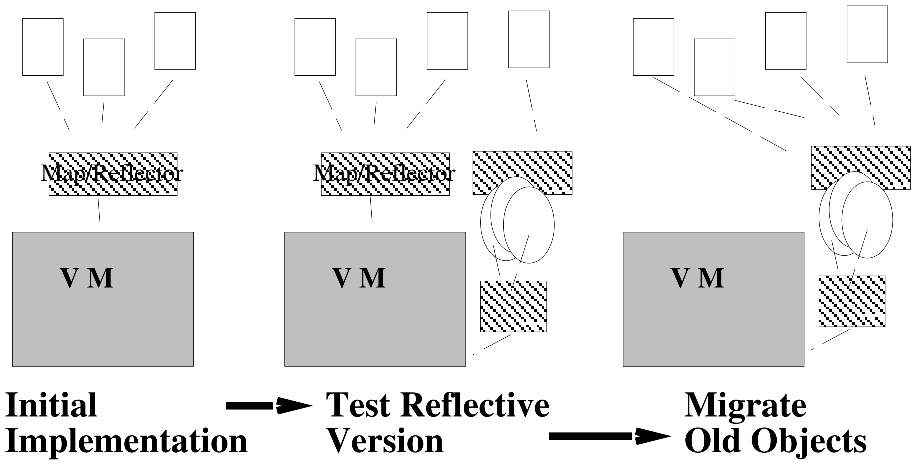


Figure 3 – Adding Reflection

Reflection can be introduced into a system in a step-by-step process to ensure a cleanly designed interface[Ste93] and to solve mutual dependency problems. This second aspect is shown in figure 3. The system in the left already presents a reflective interface to the three objects in the form of a map/reflector. This reflector, however, is a thin layer on the monolithic virtual machine - it does not actually use Self meta-objects to do its job. In the first step (middle of the figure), a deep implementation of this reflector is created with all the needed meta-objects (shown as ellipses in the figure). Note that these meta-objects are themselves objects and also have a map/reflector (which is shown as having a "shallow" implementation, but it does not have to be that way). We have extended the Image by one more layer (at least). While all this is being debugged, the system still works perfectly as all the old objects still refer to the old reflector. Once we are satisfied that the functionality of the new reflector is exactly the same as the old one, we can migrate the old objects to the new meta-space and eliminate the old reflector from the system. The last step would be to delete the now unneeded functionality from the VM to reduce its size. Note that just the last step, and maybe the first, requires a lengthy change/reboot/test cycle, and there is not much that can go wrong when simply deleting code.

The first step is the creation of a Self interpreter (we will call it "Tiny Self") in some language. We used C in the Linux system to avoid dealing directly with hardware initially. C++ would have been a better choice but was not available in some of the platforms to which we intend to port this work (this has changed since we started, however). From this starting point, the following steps will be needed to achieve the Merlin system as described in this paper:

1 – replace the parser: the parser is the part of the system that reads a textual description of a Self object and creates this object in the memory system. Replacing the original, C-based parser with one written in Self will not really enhance the system's reflection capabilities and will eliminate the possibility of "loading from an empty world" rather than from a populated Image, but it is an easy test of the bootstrapping method described above.

2 – replace the garbage collector: this is the first real test of dealing with circular definitions as the Self objects that will replace the simple C memory manager and garbage collector will need to have their memory managed as well. The second part of this step is to replace the file based secondary storage interface (Image stored in "snapshot files") with a true persistent object store.

3 – write the compiler(s): the interpreter is an unacceptable performance bottleneck, but is also simple and easy to debug. The system is designed to work with one or more compilers in a dynamic, adaptive compilation scheme. Since all the system resources are accessible to Self programs using the reflective interface, there is no reason not to write these compilers in Self. Since the compiler's code itself will be running on the slow, but debugged interpreter, the compiled code can be tested in a very controlled way. Only when confidence in the compiler's output is very high will it be allowed to try to compile critical parts of the system, including itself. Eventually, the compiled code will be stable enough to be included (just selected methods and objects) in the Image. This will allow most remaining C code to be replaced.

4 – replace the interpreter: in the last step, the interpreter could be rewritten in Self (the code would be compiled and stored in the bootable Image) or it could simply be discarded as the compilers would be sufficient to keep the system running. This might not be a good idea, however, as it would render the system unusable in case an obscure compiler bug pops up in the future.

At this point the system will be running with the originally planned architecture, but the techniques described above would still prove useful in porting the system to other machines. Machines can differ in two ways: different IO and memory systems (both the old Apple Macintosh computers and the Commodore Amiga used Motorola 680X0 processors, but had very little else in common, for example) or they may use different processors (a PowerPC based computer designed around the PReP standard, for example, has exactly the same I/O structure as most modern Pentium AT clones). In the normal case, in fact, both differences are present, but we will deal with them one at a time. If only the drivers are different, the new ones can be written in the host machine generating a "fat Image" which should be able to boot on the target machine. If something goes wrong, the drivers must be changed in the host and the process repeated. When the system is running on the target the old drivers could be discarded, but this will not be done so a single Image can be shared between the two machines (on a network, for example).

The interesting case is porting the system to a different processor. A different CPU object must be created and a new back end for the compilers must be written. When this is done, a special "cloner" process creates a bootable Image with the target machine's native code. Bringing up the system on that computer is a simplified version of the bootstrapping process described above.

The special CPU object is not very large - the rest of the system is in the Image. An interesting topic for future research would be to create a variation of the cloner using technology from the extractors. This could make the generation of very small and specialized bootable Images, suitable for embedded systems, possible. If the persistent store system is not needed for a given application (or the graphical user interfacer, or the garbage collector, or even the compilers) it would not have to be included. It is hard to evaluate just how small a system could be produced for such specialized applications without actually doing it, but it does show some interesting possibilities for the approach chosen for the Merlin Operating System.

6 Conclusion

Developing an operating system with a dynamic object-oriented programming language is quite a challenge, but the results are worth it. The best approach is to start with a reduced version of the system and to introduce reflection in a step-by-step fashion by replacing each non-reflective part of the system. The result will be an environment where applications written in dynamic languages will be distributed as efficiently as they are developed, removing one of the largest barriers for the acceptance of such systems.

7 Acknowledgments

The Self group at Stanford, then at Sun created a fantastic language for which I am extremely grateful (even if it does not seem so from my attempts to "improve it"). Laboratorio de Sistemas Integraveis at USP and my parents have made this work possible. I would also like to thank the reviewers who made such

valuable suggestions - this paper would be a lot less readable without them.

References

[AJ93] Jecel Assumpcao Jr. O sistema orientado a objetos Merlin em máquinas paralelas. In *Anais do V SBAC-PAD*, pages 304–312, Florianópolis, SC(Brazil), September 1993. In Portuguese.

[AJ94] Jecel Assumpcao Jr. Adaptive compilation in the Merlin system for parallel machines. In *WHPC'94 Proceedings of the IEEE/USP International Workshop on High Performance Computing*, pages 155–166, São Paulo, SP(Brazil), March 1994.

[AU94] Ole Agesen and David Ungar. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *OOPSLA'94 Conference Proceedings*, pages 355–370, Portland, OR, October 1994. Published as SIGPLAN Notices 29(10), October, 1994.

[Car90] Denis Caromel. Concurrency and reusability: From sequential to parallel. *Journal Of Object-Oriented Programming*, 3(3):34–42, Sep/Oct 1990.

[CMDM92] Pierre Cointe, Jacques Malenfant, Christophe Dony, and Philippe Mulet. Etude de la réflexion de comportement dans le langage Self. In *Premières Journées Représentation par Objets (EC2)*, pages 213–224. La Grande Motte, 1992. in French.

[HU94] Urs Hölzle and David Ungar. A third-generation SELF implementation: Reconciling responsiveness with performance. In *OOPSLA'94 Conference Proceedings*, pages 229–241, Portland, OR, October 1994. Published as SIGPLAN Notices 29(10), October, 1994.

[MC93] Philippe Mulet and Pierre Cointe. Definition of a reflective kernel for a prototype-based language. In *Proceedings International Symposium on Object Technologies for Advanced Systems (ISOTAS'93)*, 1993.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[Ste93] Patrick Steyaert. A two-staged introduction of reflection: A computational characterisation of reflection based on open implementations. In *OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, 1993.

[TYT92] Takao Tenma, Yasuhiko Yokote, and Mario Tokoro. Implementing persistent objects in the Apertos operating system. In *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems*, 1992. also SCSL-TR-92-015 of Sony Computer Science Laboratory.

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *OOPSLA'87 Conference Proceedings*, pages 227–241, Orlando, FL, October 1987. Published as SIGPLAN Notices 22(12), December, 1987.

[Yok92] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *OOSPLA'92 Conference Proceedings*, 1992. also SCSL-TR-92-014 of Sony Computer Science Laboratory.