Towards Active Applications: the Virtual Virtual Machine Approach.*

F. Ogel, G. Thomas, A. Galland, B. Folliot, I. Piumarta, C. Baillarguet

July 11, 2003

Abstract

With the wide acceptance of distributed computing a rapidly growing number of application domains are emerging, leading to a growing number of ad-hoc solutions that are rigid and poorly interoperable. Our response to this situation is a platform for building flexible and interoperable execution environments called the Virtual Virtual Machine. This article presents our approach, the architecture of the VVM and some of its primary applications.

1 Introduction

Traditional operating systems, such as Unix or Windows, seem a poor match for the needs of emerging applications. They rely on a "one-size-fits-all" philosophy that exports a fixed set of abstractions and applies the same policies to every application. These policies, embedded in the system, are designed to match the needs of most "standard" applications and are fixed at the time the system is designed. This philosophy does not sit well with emerging applications whose needs are difficult (or impossible) to anticipate in advance.

Many applications are therefore penalized by rigid, *a-priori* resource management choices (such as scheduling and memory paging algorithms) and would perform better if given different ones. A classic example is the interaction between a system's memory pager and a DBMS:[1] the application, or even the DBMS itself, may have sufficient knowledge to *tune* the memory paging algorithm (which is often a static, general-purpose strategy such as LRU).[2]

This observation led to the development of several flexible operating system architectures. Their goal was to give applications a way to specify their needs and then to adapt resource management accordingly. A similar observation has been made at other "levels" within the execution environment: virtual machines, middleware and languages. Ad-hoc solutions to increase flexibility have been

*This work is partially funded by the european project IST COACH (2001-34445).
[1]DataBase Management System
[2]Least Recently Used (is first evicted from the cache).

proposed, as within middleware: although distributed environments as CORBA and EJB are becoming de-facto standards, they are still complex and dedicated to *client-server-based* applications. Moreover, they are not intended to adapted or specialized to other application domains or architectures. Hence the proliferation of ad-hoc solutions, each one being dedicated to a given application domain, poorly reusable and interoperable with each other.

Examples can be found with new embedded systems/devices (mobile 'phones, smartcards, etc.), with Quality of Service (QoS) management (real-time, fault-tolerance, etc.), and (in general) with the adaptation of *any* service according to some unforeseen need, constraint or event.

As long as an artificial "red line" exists between language and system, flexibility in execution environments will remain limited. Operating systems, even flexible ones, face a barrier preventing them from addressing *any* language issues (such as interoperability or flexibility). At the other extreme, virtual machines (and other language-based approaches to flexibility) consider the operating system as a rigid, closed *black box*, thus precluding any adaptation of their underlying abstractions or services. Between the two we find middleware, which is limited both by the rigidity of the underlying operating system and by its lack of knowledge of application internals.

We therefore propose a new, systematic approach for constructing adaptable, specializable and interoperable environments based on language and system *reconciliation:* the *Virtual Virtual Machine* (VVM) [13]. The *VVM* is both a programing and an execution environment, whose goals are: (i) to facilitate adaptation of language and system aspects according to a given application domain, such as smart cards, satellites or clusters; (ii) to provide dynamic extensibility, allowing "on the fly" modification to the execution environment (adding protocols, hardware support, algorithms or even "bug correction"); (iii) to provide a common language substrate on wich to achieve interoperability between different languages/applications.

The remainder of this paper starts by presenting related works in Section 2. The *VVM* approach and its prototype are described in Sections 3 and 4, respectively. Section 5 presents the application of the *VVM* to the development of active Java applications, followed by conclusions and perspectives in Section 6.

2 Related Work

Although flexibility in execution environments is an active research area, most projects focus on one specific area: operating systems, virtual machines or middleware.

The rigidity of existing operating systems, as discussed in Section 1, has led to many flexible operating system projects. Their objectives are to give control over resource management back to the application. In order to achieve this level of flexibility, system strategies and services are organized as librairies or modules and presented as system extensions. Depending on the granularity of the kernel (exo, nano, micro or monolithic) these extensions are loaded into

the kernel or linked with the application. The two main approaches for flexible operating systems are therefore: (i) extensible kernels, that support loading of application code (extensions) in the kernel through a fixed extension protocol while enforcing security properties to preserve the integrity of the kernel and thus of the entire system; (ii) minimal kernels that export a reduced set of features and therefore push the implementation of both resource management and system services up into the application level. (Any application is then linked with dedicated abstractions and services.)

Obviously, only the minimal kernel-based approach allows a completely flexible operating system. Whereas minimal kernels do not impose any restrictions on extensions, extensible kernels rely on both technical constraints (use of a specific language, compiler, certification tool) and semantic constraints (rigid security rules, trade-offs involving integrity vs. flexibility, the model imposed by the extension protocol). Nonetheless, even minimal kernel-based operating systems address only system aspects (mainly resource management) and thus do not support any language aspects (such as interoperability and language specialization).

Middleware has similarly evolved towards dynamic flexibility. The most rigid middleware has been enhanced with ad-hoc features, such as the POA[3] and interceptors in CORBA [28]. New-generation middleware exhibits more flexibility: from dynamic specialization of a generic ORB (as with *Jonathan* [11]) to the reification of the ORB to support dynamic adaptation/reconfiguration (as in *OpenORB* [7] or the *2K* [20, 29] distributed system).

The *XVM* [17] project defines a monolithic, extensible virtual machine, thus directly applying the extensible kernel-based approach to virtual machines. Although an application can redefine primitives and basic mechanisms (such as dynamic compilation or the arithmetic operators), the global architecture of the virtual machine is still imposed, along with various low-level mechanisms, in order to enforce strict security rules that ensure the virtual machine's integrity.

An important trend for specialization is the use of *Domain-Specific Languages* (DSL). Whereas the traditional reflexive approach defines a general framework, in which application developers can express the application domain's semantics and modify some aspects of the execution environment, the DSL approach relies on languages restricted to a given application domain, and thus exposes only high-level, domain-specific abstractions to developers. This approach is used in many application domains ranging from device drivers (*Devil* [25], *GAL* [31]) through active networks (*Plan-P* [30]) and schedulers (*Bossa* [21]) to Web caches (*WebCal* [26], *CacheL* [6]).

The main drawback of DSLs, not considering the time-comsuming acquisition of any domain-specific knowledge, is the lack of dynamism. Indeed, the purpose of this approach is to specialize an execution environment according to an in-depth knowledge of the application domain's semantics. This domain-specific knowledge can also be used for code verification and formal proof. Nonetheless, once specialized, the execution environment and the application are not in-

[3]Portable Object Adapter.

tended to undergo further adaptation. Reconfiguration according to unforeseen events or the emergence of new protocols is limited to what was anticipated during the program conception/implementation: the DSL itself and its framework (compiler, interpreter, etc.) are not dynamically flexible.

Because this approach aims at constructing ad-hoc solutions (dedicated to each application domain) dynamic flexibility does not appear to be a primary concern: a specialized environment will naturally "match" an application's needs. Whereas the DSL approach provides the developer with a dedicated language and environment in which to build applications, flexible execution environments let application programmers introduce domain-specific semantics into a generic environment. As illustrated in the following section, our approach is based on a combination of the minimal kernel and DSL approaches.

By focusing only on a specific part of the execution environment, most projects exhibit only limited flexibility: operating systems are still too low-level to manipulate application semantics (and global language features), virtual machines hide resource management and low-level mechanisms, and middleware is bound to the flexibility of the underlying operating system while ignoring high-level language concerns. The artificial separation between language and system [18] leads to a *fragmentation* of the execution environment into independant and rigid layers.

Our proposal merges system and language into a "meta" execution environment that allows *direct* manipulation and definition of *any* element in the execution environment.

3 The Virtual Virtual Machine

Most modern distributed applications and environments are composed of complex and heterogeneous interacting components. Dealing with this heterogeneity raises severe obstacles to interoperability.

The virtual machine approach is a step in the right direction, allowing inter-system operability and portability, and promoting mobility/distribution with a compact code representation and intrinsic security mechanisms. But they are still dedicated to specific application domains. Let's consider Sun's Java Virtual Machine: it corresponds to an application domain where there is large amount of available memory, limited access to the underlying system and few quality-of-service guarantees.

The appearance of new application domains, with widely differing characteristics, implies new virtual machines to match the new requirements (for a given architecture, as with *JavaCard* [1] for smartcards, the *KVM* [2] for mobile telecomms, or constraints such as fault-tolerance or real-time operation in *RT Java* [14]). This proliferation of "ad-hoc" virtual machines breaks the interoperability advantages inherent in the VM approach.

If virtual machines are a step in the right direction but remain far too rigid, why not "virtualize" them? Hence, instead of developing a new virtual machine for each new application domain, a specification is dynamically loaded into a

"virtualized" virtual machine (the VVM). This specification describes a virtual machine dedicated to a given application domain.

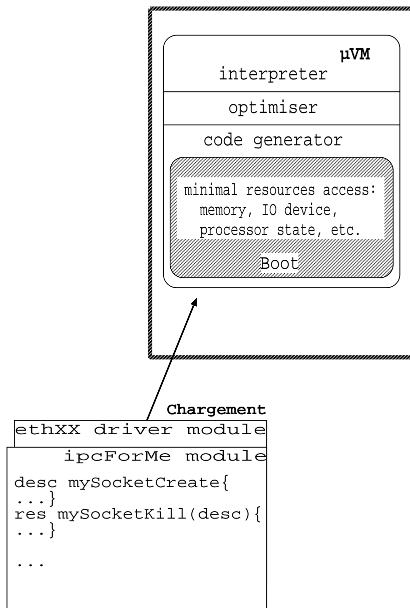

Figure 1: A minimal execution environment.

The inter-application isolation that results from using virtual machines leads to inefficient resource utilization and poor interoperability. Indeed, there can be as many running virtual machines running on a single machine as there are applications: *Java*, embedded in the Web browser, the *Emacs* editor, *PostScript* and *PDF* (Portable Document Format) viewers, *Tcl/Tk* in graphical user interfaces, and so on.

Using a single generic/meta virtual machine (the VVM) allows a factorization of several low-level mecanisms,[4] found in practically all virtual machines. This factorization results in a more efficient management of resources: (i) less memory used, thanks to the elimination of many redundant components; (ii) resources managed by a unique environment being shared in a more efficient way.

Moreover, this factorization results in a common language substrate at the lowest level, on which interoperability can be achieved more easily at both application and execution environment levels.

A major goal of our architecture is dynamic flexibility and interoperability without sacrifying performance. To achieve this the VVM relies on two principles: (i) minimality in the "core"; (ii) coupling of language and system components at the lowest level.

[4]Such as virtual processor or syntax-tree optimisations.

Minimality in the core is required for complete flexibility. Indeed, to be fully flexible, an execution environment must be free of any rigid, predefined absractions. The VVM therefore defines an environment as minimal as it is generic.

Flexibility in execution environments frequently concentrates on adapting functional aspects to particular needs and constraints. Nonetheless, adapting language aspects cannot be done without coupling language and system at the lowest level. Interoperability appears as a direct consequences of this coupling.

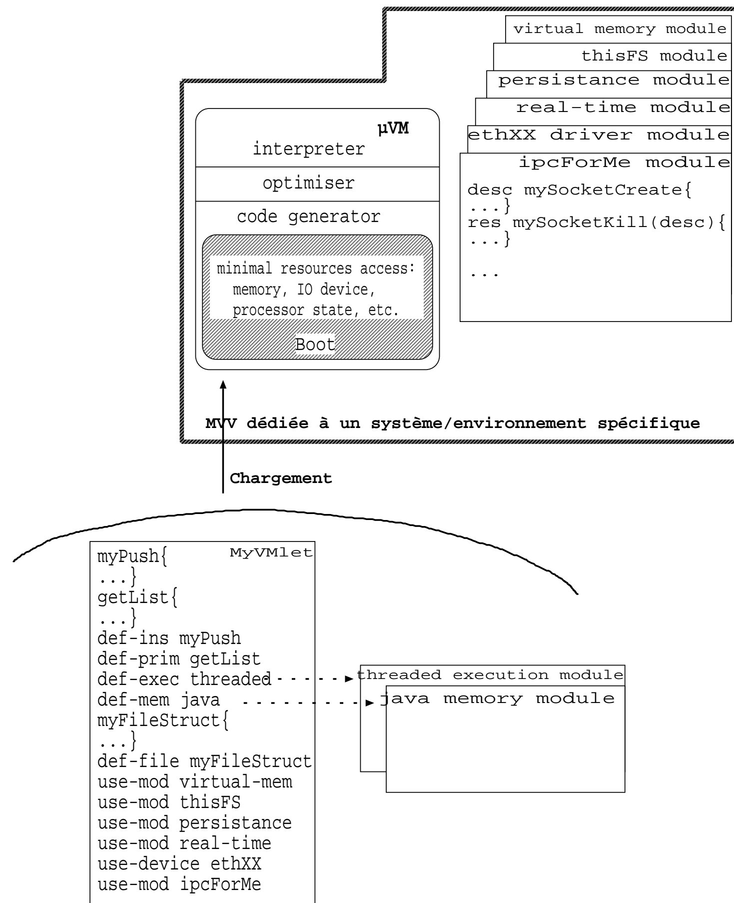

Figure 2: Dynamic construction of a dedicated execution environment.

Figure 1 represents an "empty" VVM: it is a minimal execution environment,[5] which defines neither resource management models nor abstractions.

[5]In fact, it is an execution environment dedicated to the construction of dedicated execution environments.

This "basic" environment is composed of: (i) an Hardware Astraction Layer (HAL)[6] responsible for reifying physical resources in a policy-neutral way; (ii) a generic virtual machine initially composed of a single, reflexive dynamic compiler that allows "on-the-fly" construction of arbitrary compilation chains (from lexer/parser through to native code generator).

This minimal execution environment is then extended or specialized according to a given application domain (as with a DSL-based approach). This adaptation consists of loading an "active specification", called a *VMlet*, that describes a dedicated execution environment as illustrated in Figure 2. This specification directly extends the VVM with new dedicated primitives, operators, abstractions for resource management, or language support for dynamic compilation. Hence the VVM, while remaining a generic virtual machine dedicated to execution environment construction, becomes a native, domain-specific execution environment too, offering the same semantics as (and comparable performance to) any other native, hand-coded, domain-specific execution environment.

By loading VMlets into the VVM, both functionnal and semantic aspects can be adapted. Interoperability can be defined at the VVM level, through the use of an intermediate code representation that defines a common language substrate on which to achieve data and code sharing between both applications and VMlets.

An application is "typed" with a *VMlet-Id*, used to find a suitable "loader". The application is therefore loaded and executed in an appropriate, specific environment that includes domain-specific abstractions, drivers, services and semantics, as illustrated in Figure 3.

4 A Flexible Execution Environment

Faced with the continuing emergence of new application domains, each having its own needs and constraints, we propose an architecture for a flexible platform supporting the dynamic construction of adaptable execution environments.

This architecture is structured as a set of components and interfaces, based on the *ODP Reference Model* [19]. Each component exports one (or more) interface(s) that are considered "access points" to the component. Figure 4 illustrates a component X exporting three interfaces (named *Interface1*, *Interface2*, and *Interface3* respectively). Exporting multiple interfaces allows a component to partition its functionality or to reify different aspects of its semantics, such as the level of security, trust, or QoS.

Interaction between components relies on the concept of *bindings*, which are reified through *binding factories*. Reified communication channels between components become flexible and can be dynamically adapted, for example: dynamically replacing a standard procedure call with a remote invocation in response to the migration of a component, or with a remote group invocation for dynamically-replicated components.

[6] Or a "Kernel Abstraction Layer" when running within a kernel such as Linux.

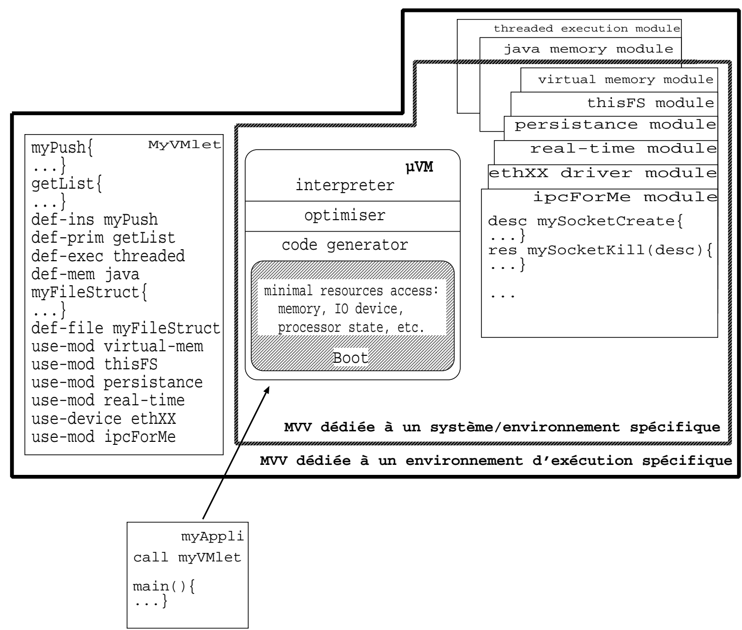

Figure 3: Loading an application.

As illustrated in Figure 5, the core of the platform is a minimal execution environment composed of two macro-components. *THINK*[7] [12] is a component-based HAL developped at *France Telecom R&D*. Physical resources are reified in a policy-neutral way, that is, without imposing any semantics. This macro-component can also be defined as a kernel extension within a "traditional" operating system, such as a Linux kernel module granting access to the kernel's implementation. The second macro-component is a dynamic compiler representing an "empty" VVM. In order to store compiled code (as well as meta-data kept from the compilation process) the dynamic compiler relies on a memory allocator that can range from a simple physical memory allocator to a sophisticated, MMU-based component; no particular model is imposed or required.

4.1 The Hardware Abstraction Layer: THINK

Whereas traditional kernels are virtual machines defining a set of software abstractions and services (such as filesystems, processes, and memory management), THINK is a library reifying physical resources,[8] (such as physical memory, MMU, and IRQs) through a set of components and interfaces. No additional logical abstractions are defined. These components allow direct access to phys-

[7] THink Is Not a Kernel.
[8] Hence its name.

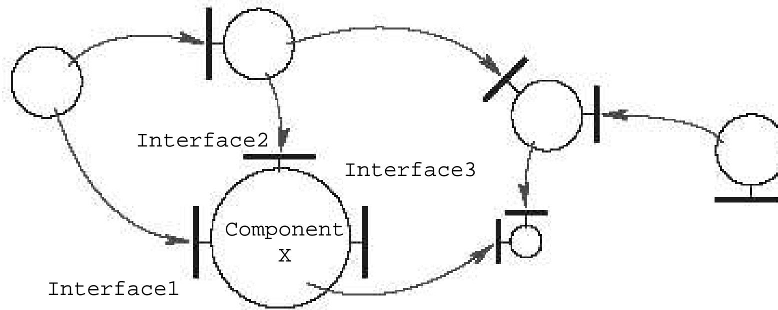

Figure 4: Components and interfaces.

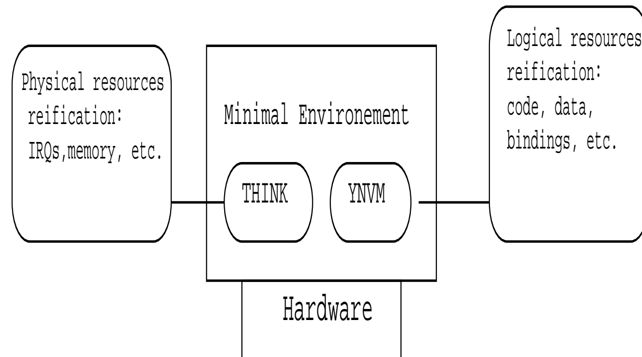

Figure 5: A minimal execution environment.

ical resources, without enforcing any control or management policies and thus offering complete flexibility.

Figure 6 shows examples of THINK interfaces for *VIA* microcontrolers and IRQs.

Although THINK provides a trading service, binding factories and several device drivers, it is only used, in the context of the VVM project, as an HAL bootloading a standalone "empty" VVM. Any additional abstractions, services or even drivers are then defined/compiled upon the dynamic compiler.

4.2 A dynamic compiler: *YNVM*

The *YNVM*[9] is our first prototype of *VVM*. Its role is mainly to support dynamic construction of dedicated virtual machines and execution environments. This implies that it is also responsible for supporting dynamic flexibility. The *YNVM*

[9]YNVM is Not a Virtual Machine.

```
interface via {
            int  start(key keymanager);
            void reset_bus();
            void reboot();
            void shutdown();
            int  get_rtc_time();
};
interface irq {
            void disable(int nr);
            void enable(int nr);
            void _register(int nr, char[] handler, char[] arg);
            void _unregister(int nr);
};
```

Figure 6: Interface definition in THINK.

is structured as a set of components and interfaces constituting an open and flexible compilation chain, as illustrated in Figure 7

The dynamic compiler relies on a garbage-collected object memory, used to store meta-data. An optional parser converts text, obtained by any input method, into an Abstract Syntax Tree (AST) stored in this object memory. A "tree compiler" then converts ASTs into instructions for an abstract stack machine, whose sematics and execution model are those of C. The tree compiler also provides meta-data reflecting the state of compiled code and applies transformation rules supplied by the *VMlets*, if any. Meta-data are organized into hierachical namespaces called *modules*. Using two abstraction levels (AST and stack machine) allows two independant and specific optimization processes. Finally, a code generator (containing a platform-specific dynamic assembler) converts the intermediate representation into concrete machine instructions. The generated instructions are not stored in the internal object memory, but rather in the application's memory. Although most expressions are read, compiled and then executed, a mechanism (called *syntax*) provides for the definition of AST nodes that are executed during the dynamic compilation process, allowing dynamic code verification and arbitrary dynamic transformation/rewriting of tree structures.

The default front-end "language" of this compilation chain is a Lisp-like textual representation of ASTs. Since the internal components of the YNVM can be accessed from the application-level, any component in the compilation-chain can be dynamically redefined; for example, to adapt the semantics of the language or to introduce new ones. An example component definition, using this default front-end language, is shown in Figure 8. This script defines a namespace (*dev.console*) that encapsulates the component definition. This definition contains the component-state (its attributes), declared through *define-state*, a default interface (*%default*) and a constructor (*new*). Since the component de-

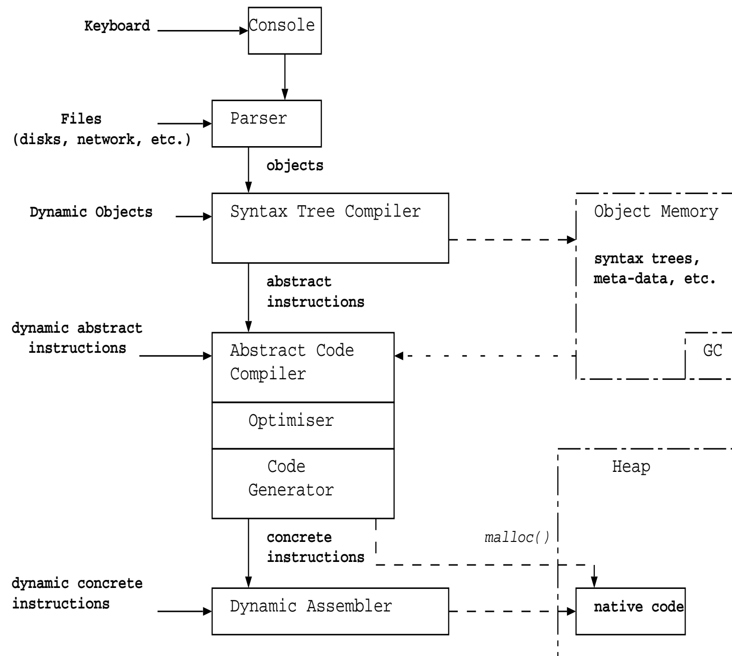

Figure 7: Structure of the *YNVM*.

```
(module dev.console)
(component.define-state fb          ;;composant framebuffer
                        x y         ;;pointer offsets
                        cols rows)  ;;range
;; defining an interface
(component.export-methods (putc    char)
                          (putcs   string)
                          (putxycs int int string)))
;; instantiation of an interface
(define %default (:component.interface myPutc myPutcs myPutxycs))
(define new (lambda(fb nbcols nbrows)
                (component.new %default 0 0 fb nbcols nbrows)))
;; allocating and using a component
(module global)
(define fb-ptr (myBindingFactory bind
                                 (myTrader lookup ``framebuffer'')))
(define myConsole (dev.console.new fb-ptr 80 25))
(myConsole putcs ``hello world !\n'')
```

Figure 8: Creation of a *console* component.

```
module(dev.console);
component.define-state(fb,          //composant framebuffer
                       x, y,         //pointer offsets
                       cols, rows); //range
// interface definition
component.export-methods(putc    (char),
                         putcs   (string),
                         putxycs (int, int, string));
// instantiation of an interface
define %default = component.interface(myPutc, myPutcs, myPutxycs);
defun new(fb, nbcols, nbrows) {
  component.new(%default, fb, 0, 0, nbcols, nbrows);
};
// allocating and using a component
module(global);
define fb_ptr = myBindingFactory
    -> bind(myTrader -> lookup(''framebuffer''));
define myConsole = dev.console.new(fb_ptr, 80, 25);
myConsole -> putcs(''hello world !\n'');
```

Figure 9: Creation of a *console* component.

pends on a *framebuffer* component, a binding between the two components has to be built, through a "trading" component (*myTrader*) and a binding factory (*myBindingFactory*).

A functionally-equivalent script based on a C++-like parser, demonstrating the flexibility of the compilation chain, is shown in Figure 9.

5 *Pomv:* a toolkit for Java virtual machines

Java has becomed a *de-facto* standard for developing distributed applications. By using a portable bytecoded representation and a virtual machine, it deals with both hardware and system heterogeneity issues.

Nonetheless, Java Virtual Machines (JVMs) are monolithic and poorly adaptable. Moreover, Sun's reference virtual machine [23] has limited support for reflection. This monolithic structure has led to the developement of many dedicated JVMs, such as the KVM [2] for mobile phones, Java for SmartCard [1], PJama [5], Gemstone [3] for persistant objects, MetaXa [24] for reflexive programming, and the Real-Time for Java Expert Group [14] that defines the API of a real-time JVM. Although such dedicated virtual machines remain close to other JVMs, their developement is time-consuming. Each platform deals with specific problems and remains rigid and poorly interoperable: a real-time JVM with persistant objects would require yet another dedicated virtual machine, whereas these two aspects are orthogonal.

Moreover, this monolithic structure does not allow dynamic extension of attribute management.[10] Let us consider JVM's bytecode verification, which is

[10] An attribute is a name followed by a meta-data struture giving information about a class,

the foundation of the Java security model [15]. It can be implemented using several techniques [22] each of which relies on a different algorithm, dedicated to a given application domain (workstation, embedded systems, smartcards, etc.). Some algorithms directly modify the application before it is loaded,[11] in order to speed-up bytecode verification. They therefore require a dedicated application loader, corresponding to the verification technique used. In KVMs, for example, a *pre-verifier* adds a PCC-like [27] proof to application bytecodes through a `StackMap` attribute that can be used only with JVMs supporting this functionality (at both the loader and verifier levels).

Another use for attributes is *escape analysis* [8] annotations. Through a static analysis of the application's bytecodes, escape analysis establishes object lifetimes and indicates stack-based allocation for certain objects, reducing garbage collection overheads.

We approach dedicated JVMs in a different way, by introducing *active applications.* An active application is composed of two distinct parts: a standard Java application and an active script that is responsible for instantiating a dedicated JVM (Figure 10). This architecture allows an application to specify the internal mechanisms it requires. Whereas traditional meta-object programming is limited to the application level, our approach allows active scripts to adapt any internal component within the execution environment.

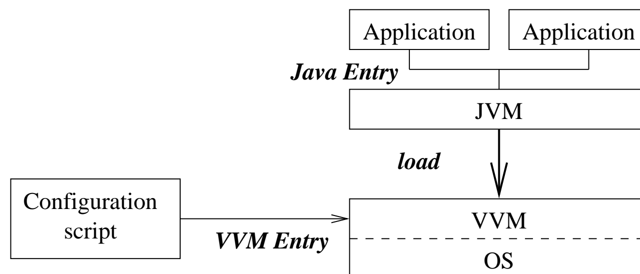

Figure 10: Active Applications

To support active applications, we developed a library called *POMV*[12] that provides several "generic" modules. These modules export well-defined APIs (allowing active applications to replace or modify their JVM's modules) as well as an adaptation API that is used to specialize module behaviors according to the application's needs, when needed. The generic modules implement Sun's specification [23] and are built on top of the YNVM; POMV is therefore a VMlet.

POMV relies on both the YNVM's adaptation mechanisms and its internal *Just In Time* (JIT) compiler, called the *Virtual Processor Unit* (VPU). Active

a method, or a field.

[11] Or rather, they modify the `.class` file encapsulating the application.

[12] Open Platform for Virtual Machines, or *Plateforme Ouverte pour Machines Virtuelles* in French.

scripts are typically written in the YNVM's default front-end language.

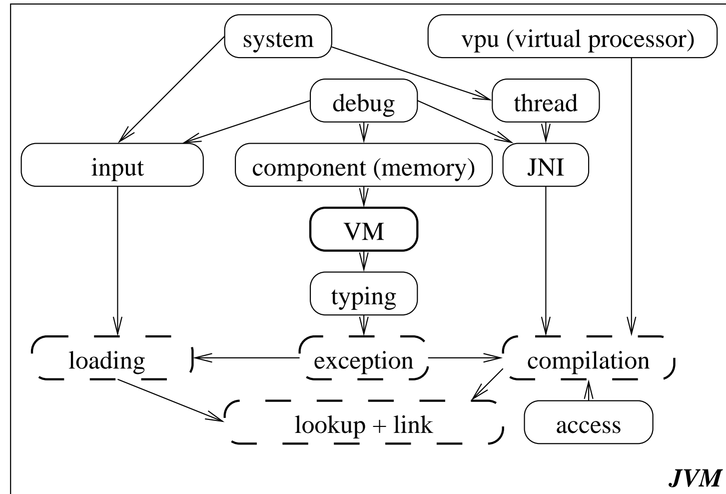

Figure 11: Generic modules in POMV

Modules in POMV are organized following a dependency-tree, as shown in Figure 11.

5.1 Naming module

Java symbols are associated with YNVM symbols: classes are represented by modules; methods and fields by symbols within modules. For example, the Object *clone() method of the java.lang.Object class is represented by the :java.lang.Object.clone_*sign* symbol, where *sign* is the method signature. This naming scheme allows an active script to interact directly with classes loaded into the JVM.

5.2 Virtual machine definition module

This module defines the basic strutures required for the assembly and execution of a virtual machine. (It is the "centerpiece" of the POMV VMlet and so replacing it may have repurcussions in many places.) A virtual machine is represented by a structure containing classes' life-cycle definitions, error management routines, and a bytecode compiler. Its interface consists mainly of:

- Input *open(char *name) opens a class description. By default it uses the underlying filesystem to find a description file.
- int (*load)(VM *, Class *, Input *) is the class loading function allowing, for example, the implementation of incremental bytecode loading over the network.

- int (*check-error)(VM *src, Object *excp) checks if an excp exception, raised by the src virtual machine, has to be caught at a particular point. By default it implements the traditional Java algorithm.
- void *(*compile)(Class *, Method *, int access) compiles the given method and returns a pointer to the resulting native code. By replacing it, the compilation process can be specialized.
- char reserved[32] bytes reserved for active applications.

Each of these function pointers is an access point to another module. An active application that modifies one of these is specializing an aspect of the virtual machine. The linker module, which cannot be adapted this way, uses a specific adaptation mechanism. This strtucture, combined with the error management and linker modules, permits the construction of virtually any object-oriented virtual machine.

5.3 Object module

Memory management relies on a portable, exact, incremental, colored mark-and-trace, inspired by algorithms by Boehm [9] and Dijkstra and Lamport [10]. References are raw pointers into memory.

Each object has an interface (stored at offset 0) that specifies the object's behavior in memory. In particular, this interface includes the function responsible for tracing objects (that is, marking all reachable objects as *live*). This function is compiled dynamically to suite each Java object type. For example, given a Java class containing only one reference and 10 integers, the associated dynamically-compiled trace function will trace only the reference. This technique not only reduces GC overhead but also reduces confusion between references and integers holding similar values.

By dynamically removing semaphores and write protection from both the memory allocator and the garbage-collector, multi-task management can be deactivated.

Java objects also have a class stored at offset 4. It is a non-typed object containing meta-data describing the object.

5.4 Classes, methods and fields

Classes, methods, and fields are untyped but garbage-collected objects, containing information required by the JVM. Each structure holds a few reserved bytes to allow more flexibility for active applications. Since these structures are necessarily bound to a given virtual machine and its associated namespace in the YNVM, many dedicated virtual machines can co-exist within a single active application.

5.5 Loading module

This module is responsible for loading Java class description files from some given input source. The predefined function allows management for previously-unknown attributes to be added dynamically. Some attributes will be managed by the virtual machine itself (such as the CODE attribute, which identifies the bytecode associated with a method); others are silently ignored.

When the load function finds an attribute, it looks for a YNVM symbol defined with the same name, in which case the corresponding function is invoked. An active application therefore extends the attribute management simply by defining a symbol and associating a management function with it.

5.6 Linker module

The linker module is responsible for binding Java symbols to methods or fields. There are 8 linking algorithms: virtual, static, and special method call, virtual and static field access, read and write, and the new pseudo-method. Since these algorithms are very similar (in terms of their adaptation interfaces) we describe only the virtual call algorithm.

The interface of the virtual call algorithm is composed of two functions: vpu-call-virtual, responsible for implementing the call on the VPU, and synt-call-virtual that converts an AST representing a method call into an AST implementing the call algorithm. Typically, the (:java.lang.Object.clone_*sign* obj) call is rewritten, using the *syntax* mechanism, to transparently implement the virtual call algorithm. By modifying these two functions, an active application can change the call algorithm.

The default algorithms are "classical". Linking is lazy, a method is not compiled until it is executed, a class is not loaded until used, and so on. To allow lazy linking, an inline call cache is used. It contains the information needed to perform the linking (such as the VVM symbol associated with the method name), the class of the last object that called the method (to verify that the virtual method is appropriate for the current object) and a pointer to a destination function. This pointer is initialized to the virtual linking function and is replaced by the address of the assembled method after the first execution of the call.

5.7 Compilation module

The compilation module has a single access point: the compile function, responsible for compiling the bytecode of a Java method. Each bytecode is compiled by calling through a 256-function switch table. In order to modify the bytecode compiler, an active application can globally modify the compile pointer in the virtual machine structure or individually modify entries in the switch table. The bytecode is compiled into the abstract instructions of the YNVM's stack-based VPU, allowing the JVM to leverage the YNVM's internal JIT "for free".

5.8 Examples

In this section we present some prototypes of active Java applications, concentrating on their associated active scripts.

The reference virtual machine

This first example of an active application instantiates a standard JVM, fully-compliant with Sun's specification [16, 23]. Basic classes required to bootstrap the JVM were taken from the GNU *ClassPath* project [4]. This active script was used to test POMV and can be used to execute any standard Java application.

The adaptable virtual machine

This second example is a remotely-adaptable JVM.

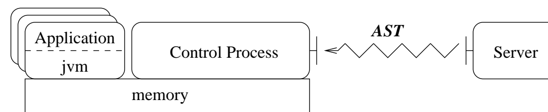

Figure 12: Remote update

The active application starts by launching a control process, responsible for receiving and executing serialized ASTs from the network, before starting the Java application itself. Reconfiguration commands can then be sent to the JVM while the application is running.

Method modification

This active application modifies Java method behavior at run-time. It relies on the previous active application: the AST sent to the control process duplicates the finalize method of the java/lang/Object class, then modifies the original. The new version prints the name of the object being destroyed, before calling the old finalize method.

Modifying the finalize method is done by modifying the description of the java/lang/Object class and by reinitializing every cache with *.finalize() calls, where * refers to any class. Synchronization is achieved through the semaphores of java/lang/Object and of the in-line caches.

This short script (42 lines of code) demonstrates how an "aspect wrapper" can be built.

Escape analysis

This example illustrates the addition and management of an application-specific attribute used to indicate the lifetime of objects allocated within a method, allowing the compiler to decide whether or not an object can be destroyed

automatically when the method returns. The management of this attribute reduces garbage collection overheads.

Firstly, the active script defines the EscapeMap symbol, which is used to read the data structure of the attribute. This structure is stored in the reserved bytes of the class structure.

It then modifies the function of the compilation module in charge of allocating the method's frame on the stack so as to allocate a structure called sea on the stack. This structure contains the addresses of the objects that should be destroyed when the method returns.

Finally, the behavior of both the object creation bytecodes and the method return bytecodes is modified. Object allocation opcodes begin by checking if the allocated object has a lifetime limited to the current method, in which case the object is directly allocated on the stack (through a call to the YNVM's version of the C function *alloca*) and marked as *to be destroyed* in the sea structure. Opcodes for method return finalize the *to be destroyed* objects by applying the finalize method to each instance marked in the sea structure.

This example demonstrates how an active application can enhance the JVM with relatively low development cost (about a hundred lines of code).

5.9 Performance evaluation

Performance measurements are still in progress, hence we present only preliminary results.

Launching a POMV-based JVM is rather slow: 3 seconds on a PPC G3 366MHz against 1 second in the case of Sun's native JVM on the same machine. This results from the dynamic compilation approach of the VVM: a POMV-based JVM is dynamically compiled *in its entirety* when launched. On the other hand, once loaded, the performance of both JVMs is similar: dynamically-compiled code exhibits performance comparable to optimized C and the indirection overheads due to the virtual machine structure are negligible.

6 Conclusion

This article presented our approach for building dynamically-adaptable execution environments. The VVM project defines a general framework emphasizing minimality of the execution environment and system/language reconciliation at the lowest level. The resulting architecture is composed of an HAL, responsible for physical ressources reification, and a reflexive dynamic compiler, used to dynamically construct dedicated execution environments. The lack of any predefined resource management, security, or programming models connected to the reflexivity of the dynamic compiler allows total dynamic flexibility.

Within the VVM project we continue to investigate a systematic approach to building flexible, adaptable, and interoperable execution environments, to free applications from the artificial limitations on reconfiguration imposed by conventional programming environments.

References

[1] http://www.javacard.org.

[2] http://www.java.sun.com/products/cldc/wp/.

[3] www.gemstone.com.

[4] Gnu classpath. http://www.gnu.org/software/classpath/classpath.html.

[5] Atkinson, M., Daynes, L., Jordan, M., Printezis, T., and Spence, S. An orthogonally persistent java. *ACM Sigmod Record 25*, 4 (December 1996).

[6] Barnes, J., and Pandey, R. CacheL: Language support for customizable caching policies. In *Proceedings of the 4th International Web Caching Workshop* (April 1999). http://www.ircache.net/Cache/Workshop99/Papers/barnes-final.ps.gz.

[7] Blair, G., Costa, F., Coulson, G., Delpiano, F., Duran, H., Dumant, B., Horn, F., Parlavantzas, N., and Stefani, J.-B. The design of a resource-aware reflective middleware architecture. In *Proceedings of Reflection'99* (Saint-Malo, France, July 1999).

[8] Blanchet, B. Escape Analysis for Java. Theory and Practice. *ACM Transactions on Programming Languages and Systems* (2003). To appear.

[9] Boehm, H., Demers, A., and Shenker, S. Mostly parallel garbage collection. In *Proceedings of the Conference on Programming Language Design and Implementation* (1991), ACM SIGPLAN, pp. 157–164. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[10] Dijkstra, E., Lamport, L., Martin, A., Scholten, C., and Steffens, E. On-the-fly garbage collection: an exercise in cooperation. *CACM 21*, 11 (November 1978), 966–975.

[11] Dumant, B., Horn, F., Dang Tran, F., and Stefani, J.-B. Jonathan: an open distributed processing environment in Java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)* (Lake District, UK, Sept. 1998).

[12] Fassino, J., and Stephani, J. THINK : un noyau d'infrastructure rpartie adaptable. In *Proceedings of CFSE 2* (Paris, France, Avril 2001).

[13] Folliot, B. The virtual virtual machine project. In *Proceedings of the IFIP Symposium on Computer Architecture and High Performance Computing* (Sao Paulo, Brasil, October 2000).

[14] for Java Expert Group., T. R.-T.

[15] Gong, L. *Inside Java 2 Platform Security : architecture, API design and implementation.* The Java Series. Addison-Wesley, June 1999.

[16] Gosling, J., Joy, B., Steele, G. L., and Bracha, G. *The Java Language Specification*, 2nd ed. The Java Series. Addison-Wesley, 1996.

[17] Harris, T. L. An extensible virtual machine architecture. In *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design* (November 1999), p. 135.

[18] Howell, J., and Montague, M. Hey, you got your compiler in my operating system! In *Seventh Workshop on Hot Topics in Operating Systems* (March 1999).

[19] ISO International Organization for Standardization. *Information Technology – Open Distributed Processing – Reference Model*, ISO/IEC 10746-(1-4) ed., 1996 — 1998.

[20] Kon, F., Singhai, A., Campbell, R. H., Carvalho, D., Moore, R., and Ballesteros, F. 2K: A reflective, component-based operating system for rapidly changing environments. In *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems* (Brussels, Belgium, July 1998).

[21] Lawall, J., Muller, G., and Barreto, L. Capturing OS expertise in an event type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW'2002)* (Saint-Emillion, France, September 2002).

[22] Leroy, X. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning* (2003). To appear.

[23] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, 2nd ed. The Java Series. Addison-Wesley, September 1996.

[24] M. Golm. *Design and implementation of a meta architecture for Java.* PhD thesis, University of Erlang, Jan. 1997.

[25] Merillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. Devil: An IDL for hardware programming. In *Proceedings of the USENIX 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)* (San Diego, California, Oct. 2000).

[26] Muller, G., Barreto, L. P., Gulwani, S., Trachandani, A., Gupta, D., and Sanghi, D. Webcal: A domain-specific language for web caching. In *4th International Web Caching and Content Delivery Workshop* (April 1999).

[27] Necula, G. C. Proof–Carrying Code. In *the 24th ACM SIGPLAN-SIGACT symposium on principles of programming Languages* (Paris, France, January 1997).

[28] OMG Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.3.1 ed., Oct. 1999.

[29] Román, M., Kon, F., and Campbell, R. Design and implementation of runtime reflection in communication middleware: the dynamicTAO case. In *Proceedings of the ICDCS'99 Workshop on Middleware* (Austin, Texas, June 1999), pp. 122–127.

[30] Thibault, S., Marant, J., and Muller, G. Adapting distributed applications using extensible networks. In *Proceedings of the 19th International Conference on Distributed Computing Systems* (Austin, Texas, May 1999), ieee, pp. 234–243.

[31] Thibault, S., Marlet, R., and Consel, C. A domain-specific language for video device drivers: from design to implementation. In *Conference on Domain Specific Languages* (Santa Barbara, California, 1997), USENIX Association, pp. 11–26.