Highly configurable operating systems : the VVM approach

Ian Piumarta[1,2], Bertil Folliot[1,2], Lionel Seinturier[2], Carine Baillarguet[1]

[1]INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

[2]Université Paris VI, Lab. LIP6, 4 pl. Jussieu, 75252 Paris Cedex 05, France

Ian.Piumarta@inria.fr, Bertil.Folliot@lip6.fr, Lionel.Seinturier@lip6.fr
Carine.Baillarguet@inria.fr

1 Introduction

Applications, especially distributed ones, have become more and more complex during the last decade. One cause of this is the increasing number and complexity of system components — for communication, fault tolerance, mobility, replication, and so on. At the same time, development environments (operating systems, software communication busses and programming languages) are changing rapidly and place stringent requirements on developers. Our response to this challenge is a multi-language, hardware-independent execution platform called the *virtual virtual machine* (VVM) [FPR98]. The essential characteristic of this platform is dynamic extensibility and reconfigurability to suit the needs of each application.

After a brief description of the VVM project in Section 2, this position paper provides some insights into the capabilities of the VVM in terms of its reflective features (Section 3) and its application to the problems communications busses (Section 4), reconfigurable/fault-tolerant embedded systems (Section 5), and interoperability of mobile code (Section 6). Finally, Section 7 concludes.

2 The VVM project

The VVM is an environment dedicated to dynamic extensibility and reconfigurability — the ability to specialise every level of both system and execution environments to specific application needs. This specialisation can be at an arbitrarily fine grain, and is reversible. The principles are to use language techniques such as virtual machines, reflection and dynamic code generation in the kernel of the system — and to achieve a much tighter integration with the application than in related solutions such as Synthetix [PW93], Spin [BSP+95] or Apertos [Yok92]. Our objectives are therfore to maximise extensibility and minimise "kernel" size; to push the philosophy of Exokernel [EKO97] (and micro-kernels in general) to the limit.

We propose a single environment supporting target applications (or application components) built in almost any "bytecoded" programming (or scripting) language. We make no assumptions about the origin of these applications. Applications are "typed" with an appropriate execution model. Each application type corresponds to a virtual machine description called a VMlet. VMlets are loaded on demand, whenever a new application type is encountered.

We use the term "virtual machine" in both the language and system sense. Agents or applets that travel over a network will almost certainly be written in a bytecoded language; in this case the appropriate VMlet will define an execution engine suitable for interpreting the application itself — a virtual machine in the language sense. Software communication busses and other middleware components will define system services possibly destined for use with fully compiled (native) applications — their VMlet defines new virtual machine functionality in the system sense, extending the abstraction of the physical resources imposed by the native operating system. In some cases (embedded systems, for example) it is appropriate for the VMlet to define both the execution engine and the system abstractions that enable applications to coexist and access local resources.

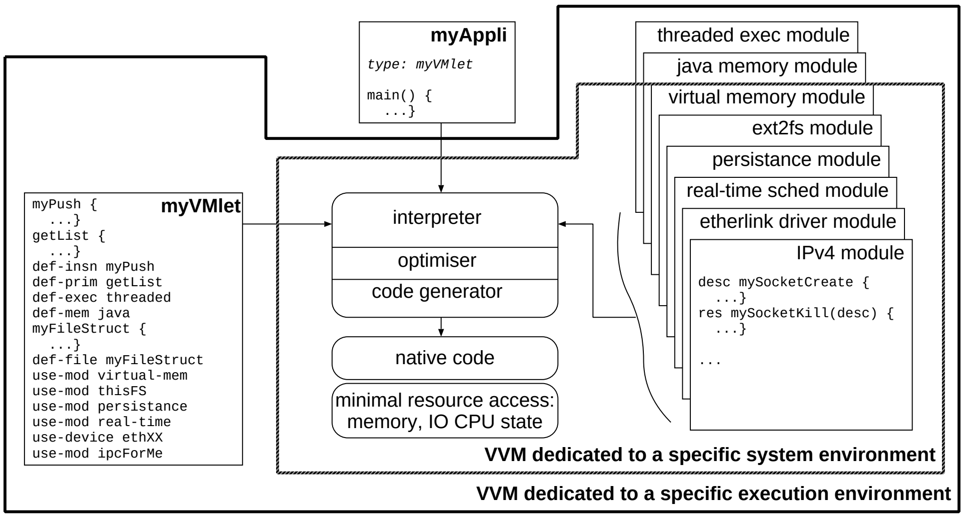

Figure 1: An example of the VVM architecture applied to execution of bytecode applications. Each application (or application sub-component) is "typed" with a VMlet specification, which defines the characteristics of the execution environment required by the application. The VM-let extends the execution environment with the specific requirements of the application. These extensions can equally concern with language (virtual instruction set, application memory model, etc.) and system (host operating system interface, filesystem and network services, etc.) components. Other uses of the VMM (for scripting in middleware applications, for example) would use far fewer modules.

2.1 Architecture of the VVM

The VMM architecture is modular, with individual reusable components described by VMlets. In practice, many system and execution components are "generic", and can be represented by a (possibly parameterised) VMlet. A usable system environment or execution engine is typically composed of various predefined VMlets (appropriately parameterised) plus any specific interconnecting "glue".

Figure 1 illustrates the VVM architecture.

3 Reflection and aspect-orientation

VMlets are imperative (not declarative) specifications that are executed when the VMlet is loaded. One side-effect of their execution is to modify or extend the VVM's functionality.[1] VMlets are encoded as compact bytecoded programs and hence any VVM will include at least one bytecode execution engine, for interpreting the VMlets themselves.

Executing a VMlet when it is loaded permits a large degree of both flexibility and security. The VMlet can introspect on and reason about the environment into which it is loaded, modifying its behaviour (and consequently the changes it makes to the VVM's functionality) appropriately. Conversely, by restricting visibility of

[1] Compare this with executing a PostScript program, whose side-effect is to leave visible strokes in a framebuffer.

the local environment and access to the intercession mechanisms VVM functionality, a VVM can ensure that VMlets do not compromise local safety or security concerns when loaded.

Besides providing a "meta" representation of a given system, reflection is also supposed to achieve separation of concerns for applications where many functionalities need to be simultaneously managed. The difficulty of this task stems mainly from the fact that most of the time the code implementing these functionalities ends up being tangled; for example, in a given software component some piece of code for mobility may be intertwined with some other piece of code dealing with replication. Distributed applications in particular, where the number of separate functionalities tends to be high, are therefore difficult to develop and maintain. One of the current trends is towards clear separation of concerns using aspect-oriented programming (AOP) [KLM+97]. This field, pioneered by languages such as D and AspectJ [LK97], permits each concern (aspect) to be tackled separately — leaving programmers to focus their attention on one problem at a time.

Of course, the fact that aspects may interfere (for example, the anomaly between synchronization and inheritance [MY93], or the non orthogonality between replication and migration when both are considered simultaneously) remains one of the major difficulties of AOP that is still being investigated by the research community. The goal of the project presented in this paper is not to tackle this problem, but rather to show that the VVM can be used as a "weaver constructor" for any given weaving model (AspectJ-like, D-like, etc.). The natural way to do this is to consider each weaving model as implementedd by a VMlet. To achieve this goal, we believe that the VVM has two main advantages. First, the reflective features of the VVM allow any (abstract) instruction to be modifed "under the feet" of the running VM. For example, data accessors/modifiers can be redefined seamlessly when replication is to be introduced. Second, weaving is a highly dynamic process that reconfigures a base level program with some new aspects. This reconfiguration process is precisely that VMlets are designed to do. The parameters of this reconfiguration, i.e. where and when aspets are to be applied (the definition of so called "crosscut actions" in AspectJ), can be defined by specialising this VMlet. Based on these principles, we are in the process of evaluating an implemention of such an aspect-oriented programming style using an early prototype of the VVM (the RVM [RVM]).

4 Communication busses

The support of heterogeneous devices and communications protocols has clearly been identified as a need for current applications. Existing micro-kernels (Flux, L4, Exokernel, etc.) and virtual machines (JVM, etc.) provide a partial solution to this problem. Nevertheless, each individual solution does not always scale well to all existing devices. Instead of being bound to only one "operating mode", the VVM provides a way to design several VMs, each of them tailored to the needs of a particular device. The VMlet specification language is then the foundation on which the communication protocols between these VMs can be implemented.

Even within existing standards, recent work on middleware systems has identified real needs for reconfigurability to support mobility of objects and/or code, dynamically-configurable or "generic" proxies, specialisable connectors for replication or security, and so on. Reflectivity has been identified as one of the most promising approaches to solving these problems [BC00], but current solutions are either too static (metaobject protocols, aspect weaving, or other compile-time reflection) or too inefficient (placing interpreted scripts in the communications path, for example).

The VVM provides the mechanisms to optimise both dynamicity and execution. VMlets are compiled to native code for execution, and can also be given access to the code generator to instantiate arbitrary behaviour as native code. Scripts represented as VMlets or otherwise passed to the native code generator have all the benefits of dynamicity offered by interpreted solutions, but run with the same efficiency as would fully-compiled static code.

A VVM-based solution does not require the use of a bytecoded application language or the use of explicit scripts. The VVM can be present as an application library, used to instantiate native implementations of communications components (loading and running VMlet specifications under the control of the application) that are connected to other (static) parts of a traditional middleware bus, such as CORBA.

5 Embedded systems, reconfigurability and fault-tolerance

Scientific satellites represent an extreme case of embedded system, posing severe problems of reconfigurability for both fault-tolerance and communication optimisation. High-energy protons regularly damage system components (regions of main memory in particular) requiring the system to reconfigure itself to use only the remaining undamaged resources. Limited communication bandwidths and availability (typically several brief periods of contact per day, limited to a total of 140kbits of information in the uplink) impose a highly compact format for reconfiguration commands.

In collaboration with the Observatory of Meudon, we are creating a limited version of the VMM for use aboard the Corot satellite [AB99] to be launched in 2005. In addition to the usual needs of reconfiguration for fault-tolerance, this satellite is unusual in that much of the scientific data processing will be performed aboard before transmitting the results to the ground station. Since physical resources are severely limited, the on-board systems operate in many different configurations depending on the mission phase (data capture, processing, telemetry, and so on). Capture and processing algorithms are based on theoretical models, but must be adaptable based on conditions actually experienced during the flight — including the uploading of entirely new algorithms.

The Corot VVM [FCP$^+$00] resembles a scripting engine, controlling the disposition and connectivity of system resources at a given instant. Reflective facilities permit the VVM to modify the operational parameters of the satellite systems, and to replace parts of itself or the image processing software as required. New configurations are determined using ground equipment, optimised to identify only those "components" that require modification, and then transmitted to the satellite as scripts for execution by the VVM. Being able to optimise the behaviour of the VVM-based configuration engine as easily as the processing software is essential for minimising the amount of data uploaded.

6 Mobility and interoperability

We have completed an early prototype of the VVM, called the *reflexive virtual machine* (RVM) [RVM], which is capable of modifying its own instruction and primitive sets at runtime (Figure 2).

To demonstrate its effectiveness we have implemented VMlets for both the PLAN [HKM+98] and ANTS [WGT98] active networks. When loaded these VMlets extend the RVM execution environment with packet encoding/decoding facilities,

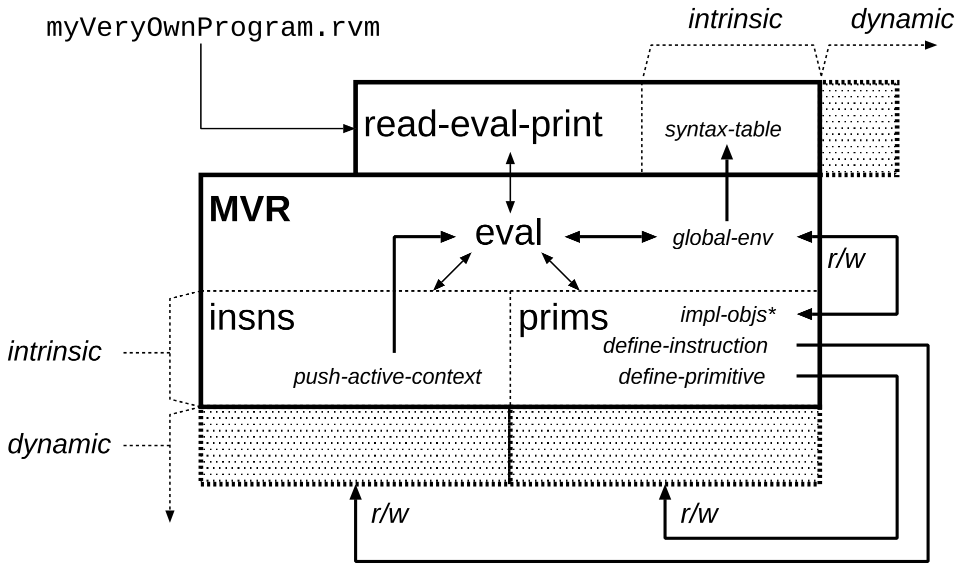

Figure 2: The Reflexive Virtual Machine implements a Lisp-like interactive language. A high degree of introspection and intercession permits the small set of intrinsic instructions and primitives to be extended dynamically by the application, which can also modify the syntax and basic execution mechanisms to suit a particular domain. There is no separate specification language: the *define-primitive* and *define-instruction* primitives take application-level functions as specifications for new functionality. The RVM can transform itself, for example, into an active network router at "boot" time. Low-level IO can be provided by a micro-kernel or (for example) a specialised kernel built using OSkit, allowing RVM-based applications to be fully "standalone".

extend the RVM language with efficient routing table data structures, extend the RVM bytecode set with new bytecodes that will appear in active packet headers, and then redefine the intrinsic RVM execution engine to consider network packets as "executable objects". The corresponding VMlets are about two orders of magnitude smaller then the original implementations of these active networks (counted as bytes of source code).

We are currently extending these VMlets to coexist, in order to experiment with "bridges" between two different active networks. Another topic of interest is "active active networks", where "control" packets contain VMlets which are passed directly to the RVM for execution — possibly fundamentally altering its underlying behaviour. This makes possible a "meta" active network, where new active network models (including those of PLAN and ANTS) can be loaded and unloaded as needed.

7 Conclusion

The Virtual Virtual Machine is an execution environment which is dynamically extensible and tailored to application needs. The reflective features of the VVM provide mechanisms for introspection on the current execution environment, and intercession to modify it. Active specifications maximise the potential of reasoning with the results of introspection before interceding. Within the context of interpreted languages, bytecodes can be added or redefined dynamically without stopping the applications. Adaptive middleware is easily supported by the ability to load different communications "personalities" on top of a common substrate.

A very limited prototype, called the RVM, has been implemented and used to dynamically construct various system and language facilities, including active networks. We are in the process of evaluating the RVM's potential as a vehicle for dynamic aspect-oriented programming.

References

[AB99] M. Auvergne, A. Baglin, et al. *Du coeur des étoiles aux planètes habitables, les enjeux de Corot.* Journal des Astronomes Français, No. 60, pp. 27–34, 1999.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, and al, *Extensibility, safety and performance in the SPIN operating system*, SOSP, 1995.

[BC00] G. Blair and R. Campbell. *Proc. Reflective Middleware 2000.* April 2000. `http://www.comp.lancs.ac.uk/computing/RM2000/`

[EKO97] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr, *Exokernel: an operating system architecture for application level resource management*, Proceedings of the 6th workshop on Hot Topics in Operating Systems, May 1997.

[FCP+00] B. Folliot, D. Cailliau, I. Piumarta and R. Bellenger. *PLERS : Plateform Logiciel Embarqué Reconfigurable pour Satellites – application au satellite Corot.* In Proc. RenPar'2000, June 2000.

[FPR98] B. Folliot, I. Piumarta and F. Riccardi. *A dynamically-configurable, multi-language execution platform.* In Proc. 1998 SIGOPS European Workshop, 1998.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, et al, *PLAN: A Packet Language for Active Networks*, International Conference on Functional Programming (ICFP), 1998.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin. *Aspect-Oriented Programming.* In Proc. ECOOP'97, LNCS 1241, pp. 220–242, June 1997.

[LK97] C. Lopes and G. Kiczales. *D: A Language Framework for Distributed Programming.* Technical Report SPL97-010 P9710047, Xerox Palo Alto Research Center, February 1997.

[MY93] S. Matsuoka and A. Yonezawa. *Analysis of inheritance anomaly in object-oriented concurrent programming languages.* Research Directions in Concurrent Object-Oriented Programming, pp. 107–150, MIT Press, 1993.

[PW93] C. Pu and J. Walpole, *A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design*, Technical Report OGI-CSE-93-007.

[WGT98] D. J. Wetherall, J. V. Guttag and D. L. Tennenhouse, *ANTS: A toolkit for building and dynamically deploying networks protocols*, IEEE OPENARCH'98, San Francisco, CA, April 1998.

[Yok92] Y. Yokote, *The Apertos Reflective Operating System: The Concept and Its Implementation*, OOPSLA 1992. Sony CSL Technical Report SCSL-TR-92-014.

[RVM] `http://www-sor.inria.fr/projects/vvm/`