Reflective and Middleware Features of the VVM

Bertil Folliot(1)(2), Ian Piumarta(1), Lionel Seinturier(1)

(1) Univ. Paris 6, Lab. LIP6 - Thème SRC, 4 pl. Jussieu, 75252 Paris cedex 05, France
(2) INRIA, Projet SOR, Domaine de Voluceau, 78153 Le Chesnay, France

Email: Bertil.Folliot@lip6.fr, Ian.Piumarta@inria.fr, Lionel.Seinturier@lip6.fr

1 Introduction

During the last decade, applications, especially distributed ones, have become more and more complex. One of the reasons of this fact, is that the number of system and network functionalities (for communication, fault tolerance, mobility, replication, etc.) to be included, has constantly increased. On the other hand, development environments (operating systems, software communication buses, programming languages) are rapidly changing, and put some stringent requirements on developers. Our response to this challenge is a multi-language, hardware independent, execution platform, called the virtual virtual machine (VVM). The genuine characteristic of this platform is to be dynamically extensible and adaptable to each application's needs. After a brief description of the VVM project in Section 2, this position paper provides some insights on the capabilities of the VVM in terms of reflective features (Section 3), and communication bus (Section 4). Finally, Section 5 concludes.

2 Virtual Virtual Machine

We propose a single environment supporting target applications built in almost any "bytecoded" programming 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.

3 Reflective features of the VVM

A prototype "Recursive Virtual Machine" (RVM) [FPR98] has been built. The RVM is a virtual machine for a Scheme-like language, in which new primitives and instructions can be specified "recursively" in terms of the intrinsic instructions. Much of the implementation level is visible to the programmer. The possibilities of this openness are demonstrated with user-level implementations of lightweight processes, synchronisation facilities, full upward continuations, user-defined syntax, OO extensions (classes with message passing and inheritance), etc. The two basic characteristics of a reflective system are to be able to reason about itself by accessing (feature known as introspection), and to modifying (intercession) its internal structure. They are straightforwardly achieved in the RVM, by allowing a program to access the instruction set of the VM, to dynamically define new instructions, and to redefine existing ones. By this way, the reflective capabilities of the RVM are used to adapt a given VM to a new environment.

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 mainly stems from the fact that most of the time, the code that implements these functionalities ends up being tangled: i.e., in a given software component, some piece of code for mobility may be intertwined with some other piece of code dealing for instance, with replication. Applications, especially distributed ones where the number of functionalities to take into account may be rather high, are then hard to develop and to maintain. One of the current trends in software engineering is to achieve a clear separation of concerns with some aspect-oriented programming (AOP) techniques [KLM+97]. This field, pioneered by languages such as D [LK97] or AspectJ [LK97], suggests (1) to tackle each functionality (i.e., each aspect) separately, and (2) to write a so called aspect weaver to compile them into the application. Most of the time, the weaver provides some reflective features on the programming language. Even if the aspects are not always completely orthogonal (see for instance, the anomaly between synchronization and inheritance [MY93]), it is assumed that programmers will then, be able to focus their attention on one problem at a time. We believe that the VVM provides a clean testbed platform for AOP, where each system aspect (for instance mobility or replication) could be implemented as a dedicated instruction set. Unless the code tangling issue still remains to be solved at the application level, the support of new aspects comes throught some new instructions appended to the VM, and the weaving process is provided by the VMlet.

4 The VVM as a communication bus

The support of heterogeneous devices and communication protocols has clearly been identified as a need for current applications. Existing microkernels (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 every existing devices. Instead of being bound to only one operating mode, the VVM provides a way to design several VMs, each of them taylored to the needs of a particular device. The VM description language is them the foundation on which the communication protocols between these VMs can be implemented.

5 Conclusion

The Virtual Virtual Machine is an execution environment which is dynamically extensible and tailored to application needs. Reflective features of the VVM are provided through mecanisms that allows to retrieve the current instruction set of a VM, and to add or redefine existing instructions. Middleware features of the VVM are provided by the ability to load different communication personalities on top of a common ground (i.e., the architecture of the VVM).

A prototype version, called the RVM, has already been implemented. We are in the process of evaluating it to carry out work on aspect-oriented programming.

6 References

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

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP'97, LNCS 1241*, pages 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. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.