REFLECTIVE VIRTUAL MACHINE

Karsten Verelst*
Vrije Universiteit Brussel
Programming Technology Lab (PROG)
pleinlaan 2, 1050 Brussels
Belgium
karsten.verelst@vub.ac.be

Werner van Belle
Programming Technology Lab (PROG)
Vrije Universiteit Brussel (VUB), Brussels Belgium
werner.van.belle@vub.ac.be

Theo D'Hondt
Programming Technology Lab (PROG)
Vrije Universiteit Brussel (VUB), Brussels Belgium
tjdhondt@vub.ac.be

Abstract We claim that current day reflective architectures do not offer sufficient functionality, and that new developments in computer science push us towards a stronger reflective model: reflective virtual machines. We have witnessed these shortcomings in the application domain of mobility. Strong mobility is very difficult to implement in today's programming languages, mainly because of the inability to capture the program's computational state. Therefore we propose a new reflective architecture, the reflective virtual machine, that offers sufficient support for applications in mobility. In this paper we will first describe the basic functionality a mobile agentplatform should offer. This shall be done using a solution to the malicious host problem as a case. After identifying these needs we will introduce an interpreter, the Reflective Virtual Machine, that offers sufficient reflection, so that mobile applications can be straightforwardly implemented.

Keywords: reflection, mobility, security, malicious host problem

* Karsten Verelst is funded by the Belgium Fund for scientific Investigation (FWO)

1. Introduction

Reflection has since long proven its use. Most current day languages have at least reified their abstract grammar into the object level, thereby creating an abstraction level that allows an elegant solutions to many problems. However when we turn to new applications domains such as mobile agent systems we notice some shortcomings in today's reflective architectures.

Suppose we want to implement a strong mobile program; that is a program that has the ability to move to another location on the network at any time during its execution, even during loops and deep nested functions. To implement this application the following five steps should be taken: First the program's computational state must be captured. Then it should be serialized and moved across the network. And finally the receiver should deserialize the message and continue the execution of the transferred computation.

Although this algorithm seems simple enough it cannot be easily implemented in most popular languages today. Especially the capturing and restoring of the computational state proves to be problematic. A program's computational state usually consist of some sort of stack, a memory and the code. For a language like scheme for example this corresponds to the call stack and the environment containing variable-bindings. For a compiled language the computational state would more resemble a data stack, a handful of registers and the code. So if we want to capture the computational state, we need to obtain a copy of the virtual machine's internal stack, memory model and code. Accessing the code is not usually the problem because the abstract grammar is already reified in many languages, but capturing the stack and memory is much more problematic and often requires the programmer to maintain an explicit copy of the virtual machine's internal data structures himself.

In practice we have observed that strong mobility has been implemented in most popular languages, although the elegance and possible restrictions of the resulting code usually strongly correspond to the reflective nature of the language. So judging by this introduction we can already conclude that mobile applications require a strong reflective architecture. For an interpreter to be useful as a mobile agent platform, it will not only need to reify it's abstract grammar, but also a large part of it's internal state such as the stack and the memorymodel. In the next section we will present a solution to the malicious host problem and use this case to clearly identify which internal datastructures an interpreter should reify and what basic functionality it should offer. After this section we will demonstrate how we can implement such a virtual machine.

2. The malicious host problem:

Mobile agent security can be divided into three different fields. First there is the matter of safe communication: sending our agent to the remote host with-out anybody intercepting the code. This challenge can be easily solved using cryptography. Next there is the threat of possible viruses: malicious agents that try to infect a remote host. Although much work remains to be done here, there exist some solutions like the Java sandbox model[3] and proof carrying code [10]. The main challenge in agent security is trying to protect the agent from a malicious host. Suppose we would create an agent that visits several airport sites looking for the cheapest flight to a certain location. Our agent would be sent to a remote airport site, conduct a local database search and continue its voyage to the next airport. Now suppose that our agent arrives at a malicious airport. Since the malicious host has access to the agents internal variables, nothing prohibits him from tweaking the stored prices of previously visited airports, or even rewriting the agent's code. This is called brainwashing. Up to now few techniques have been found to effectively secure an agent against brainwashing. Data-encryption can never work because the agent needs to be able decipher the data it needs to work with. But if the agent can do this, then so can the malicious host. One possible solution to protect against brainwashing is the application homomorphic functions[9], first presented by Sander & Tschudin. The technique they present consists of encrypting the agent's input and letting the agent calculate with the encrypted data. Then afterwards we will try to obtain the unencrypted result from the encrypted computation's answer. For example, suppose that we have an agent that computes a very simple function, like an exclusive or of two numbers. Then to securely calculate the result of this function we will encrypt the input, pass it to the agent and later on try to decrypt the computed result. In our example we could encrypt the data by taking the complement of one of the arguments. Then we let the agent do its computation on the encrypted data, and later we decrypt the result by taking the complement once more. As a result, the remote host never knows what encryption scheme was used, so it can never interpret the agent's data.

Although this technique is very promising and is currently one of the few solutions proposed to solve the malicious host problem, there are still a few drawbacks inherent to it. First of all the code itself is not encrypted, so nothing prohibits the host from recoding the agent. Another major problem is loops and recursion. For a loop to be correctly executed, the loops result should be partially decrypted so that the same loop invariant is satisfied at the beginning of the loop again. This partial decryption might help the host to decrypt all data or abuse the loop's results.

To solve these drawbacks we propose a solution where this technique is not only applied to the agent's data but to the agent's code as well. This would result in creating a homomorphic agent: an agent that computes an algorithm similar to the original agent but that produces an encrypted result. Again the malicious host doesn't know the used encryption scheme and can therefore not recode the agent in a sensible way. As a simple example we can present the simple increment(x) operation and encrypt it so that it will actually compute 'x+2' in place of the original function 'x+1'. Again after the computation we can decipher the result by subtracting one from the result, but the computing platform has no idea what it is computing and can therefore not sensibly tamper with the code. In a real implementation, security can even be further enhanced by combining this encrypted program with Sander & Tschudin's original idea, resulting in an encrypted program working on encrypted data.

Next we will look at such an encrypted program as an agent, being executed by a specific interpreter. This means that an encrypted mobile agent is sent over the network as piece of code accompanied with it's own interpreter. Now suppose that we are able to redefine this interpreter at runtime. For example that we can at runtime decide that the increment function gets encrypted differently. This would offer more flexibility to the agent's encryption scheme and could even eliminate the partial decryption of a loops results by redefining the loop so it now correctly executes on the new data encryption scheme.

It is apparent that the implementation of this technique imposes very strict rules on the virtual machine. First of all, since we are dealing with mobile agents, we would like reification of the computational state, eg the abstract grammar, the stack and the memory model. Next to this, we just showed that we need a very flexible interpreter where the programmer would be able to redefine the interpreters semantics at runtime. Therefore we need a reflective virtual machine that reifies it's own primitives.

3. The Reflective Virtual Machine

We define a reflective virtual machine as a virtual machine that reifies and absorbs its entire computational state including the abstract grammar, the memory model, the environment model, the stack and its primitives. As explained above we want a virtual machine that offers as much reflection as possible to the user. This starts of course with the reflection of the abstract grammar. When a programmer inserts a program into the virtual machine, the program is first parsed and transformed into a treelike structure, the abstract grammar. Under reflection of the abstract grammar we understand that the nodes of this tree are made explicit in the language. An example of a reified abstract grammar can be found in the java.lang.reflect package. All methods in this package allow access to the internal representation of Java's first class data structures. Our reflective virtual machine must of course reflect its entire abstract grammar. This implicates that all metalevel data structures must be first class and this also implies the existence of meta-operators such as read, eval and apply. We will not elaborate further on this subject since reflection of the abstract grammar is already well understood and almost all popular languages today exhibit at least some reflective features.

Next we also wish the reification and absorption of the entire computational state. As explained above, the computational state usually exists of some sort of stack, a part of memory and the program code. To reify the computational state we must reify these three data structures. Since the program code is internally represented as abstract grammar, reflection of the code is essentially the same as reflection of the abstract grammar described above. So next on the list is reflection of the stack. Reification of the stack means that the meta level stack should be explicit and that it should preferably be stored in a object level data structure. In practice this means that the virtual machine's stack can best be created in the interpreters heap and that it is best implemented as some table or a list or whatever equivalent datastructure your programming language supports. Also follows that all objects that can be stored on the stack should again be reflected in the object level. For example, if your stack can contain debugging information, then this information must also be made explicit in the object level. Finally, to reify the entire computational state we also need to reify some part of the environment. What this environment looks like depends very heavily on the virtual machine. In case of a compiled program, this are usually some registers and the heap, while for a functional language this looks more like an environment with variable bindings, and for an object oriented language this can be the entire object hierarchy. Independent of what it looks like the environment should be reflected in the language. Again this might imply that meta level data structures need to be made explicit and that possibly new datastructures the represent this environment need to be constructed.

When these three meta level datastructures are reflected into the object level we have successfully reflected the entire computational state. Technically this means that we have introduced sufficient means of reflection to implement the mobile application presented before. However, we want to go further and also reify the virtual machine's primitives and memory model. We are already observing an evolution towards this idea in the Squeak virtual machine. Currently the Squeak virtual machine is already written in the language itself. To actually use this metacircular Squeak it is first compiled to the C programming language and then this generated C code is further compiled and the user is presented with a new virtual machine. Our aim is to continue this evolution and add more reflective properties to the language, so that our virtual machine can be rewritten at runtime.

For this to be possible, the virtual machine's primitives should be reflected into the language, or in simple words, the virtual machines should be written in the programming language itself. Under primitives we understand all functionality offered by the meta level. This ranges from natives like '+' and sqrt, to the entire eval-method. The big advantage of such a metacircular implementation is that the programmer can at runtime change the interpreter's behavior. For example nothing prohibits him from introducing new primitives or redefining the evaluation of the existing ones. Since we also consider the parser (the read-primitive) part of the reflected primitives, the language's syntax isn't statically defined anymore either. Redefining this read-native would allow the programmer to adopt any syntax he likes. Another possible application can be an automatic versioning system. As time goes by there will be many different versions of the virtual machine in circulation and we encounter the problem of applications requiring a certain version of the VM before they can run. This problem can now be easily solved because the application itself can upgrade the virtual machine to the version it requires. From these examples and the case presented above it is obvious that reflection of the interpreter's primitives is really worthwhile researching, even though it has some serious implications on the design of the virtual machine. So should the VM implementation consist of many little modules, where each module corresponds to a single primitive, so that changes to the primitives will only have a limited impact. There is also the problem of poor performance. Most metacircular interpreters have a tendency to be slow. However we will explain in the next section how this performance degradation can be solved using JIT-compiling.

This is what we understand under the term Reflective Virtual Machine. A virtual machine that reflects as much as possible of meta level datastructures, resulting in a small mini-kernel and a metacircular interpreter, that is then reflected into the meta level. Also we have shown that many different applications domains can benefit from the flexibility that the RVM offers. Examples for this can be found in the domains of mobility, concurrency, scheduling, distribution, meta-programming and AOP, versioning tools and interpreter design.

4. RVM design

We have described what a reflective virtual machine looks like and what benefits it offers over other less reflective interpreters. Now we will give some guidelines about how such a reflective virtual can be implemented. For the implementation of the reflective virtual machine we started with a small stack machine called pico[2]. This is a small imperative programming language, simplicity being one of its primary design goals. It already offers a completely reified abstract grammar and can be very naturally converted to a complete reflective virtual machine. Evaluation in this interpreter is based on continuations, which we define as an indivisible part of an execution. For example a '+' primitive consists of three continuations: one continuation for the evaluation of the first argument, another continuation for evaluation of the second argument and a third continuation that actually calculates the result of the binary operator. Because the evaluation of the first two continuations might result in a large computation, involving many more continuations, we store all continuations on a continuation stack. This stack contains the 'future' of the current evaluation. Our Reflective Virtual Machine will be entirely defined in terms of these continuations and will therefore only consist of a small mini-kernel that each time picks the top continuation from the continuation stack and executes it. So if we succeed in reflecting these continuations in the object level we will have succeeded in a large part of the goals we set out in the definition of the virtual machine: reflection of the virtual machine's primitives.

We believe that there are three possible ways to reflect continuations into the object level. The first is to create an abstract grammar component that represents a continuation. This would allow the programmer to create new primitives by rearranging existing continuations. However this does not offer us the flexibility we had in mind and the performance would be sluggish. A second technique would be the metacircular evaluator where all primitives are written in the object language and are evaluated by the metacircular engine. This of course would allow us easy access to all primitives but the overall performance would be horrible. So we chose for the third option where all primitives are written at the object level, but are then run them through a JIT-compiler so that the execution can be carried out in reasonable time.

So by now we get to the point where the RVM looks like a mini-kernel written in the metalanguage, a bunch of primitives defined in the object level and a JIT-compiler. This allows the reflection of all the data-structures we wanted and allows a good performance, but leaves us with the problem of bootstrapping. This can be solved by supplying the virtual machine with a set of primitives written in the metalanguage. Once the virtual machine has booted we can compile all object-level primitives and replace the first set of meta level booting primitives.

Now that the underlying structure of the virtual machine is defined, we can take a look at how the continuation stack can be reflected. In theory this is not so difficult. We make sure that we use one of the language's datatypes (like a table or a list) as the internal representation for the stack and make sure everything that can ever be put on the stack is reflected. However in practice we must be cautious: since both the interpreter and the programmer can access the stack at the same time we must look out for concurrency problems. That is why we opt for a functional virtual machine with as few destructive operations as possible.

Apart from the stack also the abstract grammar, the environment and the memory model have to be reflected. This should not be a big problem since this is already implemented in many languages today and this issue is well understood.

5. Conclusion

We have shown how current day reflective architectures don't offer sufficient support for several application domains such as mobility, concurrency and scheduling. We have proven this claim by taking the malicious host problem as a case. The solution we proposed for this problem is based on the encryption of the agent itself, by redefining the interpreters semantics at runtime. Of course this requires very strong reflectional properties of our interpreter and that is why we introduced the reflective virtual machine.

A RVM is set out to be a platform that offers sufficient functionality to support mobility and is defined as a virtual machine that reifies it's entire computational state, including abstract grammar, stacks, memory and primitives. Further have we shown how such a RVM can be built and what special issues should be dealt with to avoid concurrency problems and keep a reasonable performance.

References

[1] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y., Y Kimura, OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java, Springer Verlag heidelberg, May 2000

[2] T. D'Hondt. http://pico.vub.ac.be/

[3] Gong, Java Security: Present and Near Future, 1997

[4] W. Van Belle, K. Verelst, T. D'Hondt, Location Transparent Routing in Mobile Agent Systems Merging Name Lookups with Routing December 1999

[5] B. Folliot, I. Piumarta, F. Riccardi, Virtual Virtual Machines, September 1997.

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

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay Back to the Future The Story of Squeak, A Practical Smalltalk Written in Itself

[8] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison Wesley, 1989, ISBN 0-201-13688-0

[9] T. Sander, C. F. Tschudin, Protecting Mobile Agents Against Malicious Hosts, November 11, 1997

[10] J. Feigenbaum and P. Lee. Trust Management, and proof carrying code in secure mobile-code applications (A position paper). march 1997