How to formally specify the Java Bytecode semantics using the B method

Ludovic CASSET
Jean-Louis LANET
How to formally specify the Java Bytecode semantics using the B method

Ludovic Casset
Ludovic.Casset@gemplus.com

Jean Louis Lanet
Jean-Louis.Lanet@gemplus.com

Introduction

The new platforms (i.e., Java Card, MultOS and Smart Card for Windows) allow dynamic storage and the execution of downloaded executable content, which is based on a virtual machine for portability across multiple smart card microcontrollers and for security reasons. Due to the reduced amount of resources, a specific Java has been specified for the Java card industry, known as the JavaCard 2.1 standard. The Java card specification describes the smart card specific features of the virtual machine (i.e., applet firewall, shareable interfaces, Installer...).

All those mechanisms prevent hostile applets to break the security of the smart card. However, the smart card security is based on the assumptions that the JCRE (Java Card Runtime Environment) is correctly implemented. The correctness of the JCRE is important because it is the means to avoid an applet to reference illegally another applet objects. In fact, not only the applet firewall but also the complete JCRE and the virtual machine must be correctly implemented. In order to prove such a correctness we have to use formal methods to ensure that the implementation is a valid interpretation of the specification.

In the specification, it is not explicitly explain how and when the different controls are done (i.e., type checking, control flow...). The smart card industry proposes an architectural design where the checks are performed off-card. A defensive virtual machine where all the checks are performed at the run time has too poor performances. The developers have to extract the static and the dynamic semantics. The static constraints are performed with an off-the-shelf verifier and the on-card interpreter implements the dynamic semantics. If we want to formally implement the interpreter we have to expect that the verifier has been correctly implemented. We propose hereafter a model based on the refinement technique that avoid this potential incoherence.

After a brief presentation of related work, we present the bytecode subset used in our model. Then we define the state of the defensive virtual machine using the method B of Jean Raymond Abrial [Abr-96]. An example of instruction refinements is provided. Then we conclude with the extension of our work.
Related Work

There has been much work on a formal treatment of Java and specifically at the Java language level by [Nip-98], [Dro-97] and [Sym-97]. They define a formal semantics for a subset of Java in order to prove the soundness of their type system. A closer work to our approach has been done by [Qia-98]. The author consider a subset of the byte code and its work aims to prove the run time type correctness from their static typing. Using its specification he proposes a proof of a verifier that can be deducted from its virtual machine specification.

The Kimera project [Sir-97] proposes a verifier implementation that has been carefully designed and tested but not based on formal methods. An interesting work has been partially done by [Coh-96] in order to implement formally a defensive virtual machine. It is possible to prove that this model is equivalent to an aggressive interpreter plus a sound byte code verifier.

A new approach

Our approach is based on the Defensive Java Virtual Machine (DJVM) that we split to obtain in the one hand the bytecode verifier and in the other hand the interpreter. At the abstract level, we define the DJVM. By successive refinements, we extract the run-time checks in order to de-synchronize verification and execution process. Then, we obtain invariants representing the formal specification of the static checks. We implement those specifications with on on-the-shelf type inference algorithm.

The Freund and Mitchell subset

Freund and Mitchell introduce in [Fre-98] a bytecode subset. Instructions in this subset are chosen to represent, at the flow and data levels, bytecode instructions. We use a small variant of this subset. The difference comes from the specialization of instructions \texttt{Istore} and \texttt{Iload} which load or store local variables of type integer. In these 10 instructions, one can find instructions allowing integer manipulations and also instructions allowing object creation, initialization and action. Informal specification of these instructions is given below (Figure 1).

Operational and static semantics are described in [Fre-98]. Freund and Mitchell also prove this subset is sufficient to study object initialization, flow and data-flow controls.

- **Inc** adds one to the integer in top of stack.
- **Pop** removes the top element of the stack.
- **Istore x** removes the integer from the top of stack and puts it into local variable x.
- **Halt** terminates program execution.
- **Init σ** initializes the object of type σ on the top of stack.
- **Push0** pushes integer on stack.
- **If L** jumps to L or to next instruction according to the value of the integer L.
- **Iload x** loads value from local variable L and puts it on top of stack.
- **New σ** allocates a new uninitialized object of type σ on the top of stack.
- **Use σ** performs an operation on a initialized object of type σ.

**Figure 1** Informal specification of Freund and Mitchell instructions
Flow control and type correctness

Checking a program means insuring that all instructions are executed in a safe way. We first begin with executing controls on flow and types. We assume we work on a subset of Java types: integer, addri (unitialized object) and addr (initialized object). For such a work, we define a state and its properties. A state is defined by:

- the pc, the program counter which value is included in method domain.
- the type stack, bounded by top_stack, in which there is the type corresponding to the value in the stack.
- the type frame, bounded by max_frame and containing types of local variables.

We assume constraints insure the correct state. Then, we use transfert functions related to each instruction to lead to another correct state. The static semantics gives the constraint set, as the operational semantics gives the transfert functions. We define a complete lattice with the three types described previously. To implement an algorithm checking types, such as the one presented by Dwyer in [Dwy-95], we need such a lattice to organize types and to have relations between types. This algorithm is implemented in the off-card verifier.

The B model

We explain the model on a particular instruction, the instruction Inc. An informal specification of this instruction can be: Inc takes an integer on top of stack and add one to this integer. Clearly, the instruction, on flow level, increments the pc to go to the next instruction. For type verification, it checks that the type on top of stack is an integer. We don’t care about the value, as we say previously.

Our abstract model represents the DJVM: we make checks on pc domain and on types and then we execute the instruction. Figure 2

```
ins_inc = SELECT (methode (apc) = inc )
THEN
  IF (apc < size (methode) ∧ top_stack>0 ∧ types_stack(top_stack)=INTEGERS)
  THEN apc := apc + 1
  END
END;
```

**Figure 2** Instruction Inc in the DJVM machine.

Then, we refine until all checks appears in the invariant. The execution is done if the variable unchecked set by the invariant is false.

After two refinements, it is possible to express the checks with the following invariant (Figure 3).

```
∀ka. ( ( ka ∈ dom(methode)) ∧ methode(ka)= inc ∧ unchecked = FALSE
⇒ ka < size(methode) ∧ STtop_stack(ka) > 0 ∧ STtypes(ka)=INTEGERS ∧
STtop_stack(ka) = STtop_stack(ka+1) ∧ STtypes(ka+1)=INTEGERS) ∧
∀cka. ( cka ∈ dom(methode) ∧ methode(cka)=inc ∧
(STtop_stack(cka)=0 ∨ STtypes(cka) ≠ INTEGERS )
⇒ unchecked = TRUE)
```

**Figure 3** The invariant for Inc after two refinements.

In the operation related to instruction Inc, we do not perform test any more (Figure 4).
ins inc = \text{SELECT} (\text{methode} (apc) = \text{inc} \land \text{unchecked} = \text{FALSE} )

THEN apc := apc + 1

END;

Figure 4 The operation for instruction \textbf{Inc} in the last refinement.

The first refinement introduce the flow control. We make the type control in a second refinement. Once, we have done this work, we bring to the fore that we have split the Defensive machine. Then we introduce another abstract machine that initialized the variable unchecked by performing static checks on the bytecode. This machine is in fact our verifier. The last refinement of the defensive machine appears to be our interpreter.

Conclusion

We entirely prove the split of the defensive machine at the flow control level. We are about to finish the proof concerning the type correctness. The modelization is good and the only remaining problem is to achieve the proof, and the few proof obligations seem to be true. It is just a matter of time. We also begin to integrate the entire bytecode instruction sets. We keep on working on the implementation of the bytecode verifier and interpreter.

Références


http://www.cli.com/software/djvm

[Dro-97] S. Drossopoulou, S. Eisenbach, \textit{Java is Type Safe - Probably}.


[Nip-98] T. Nikkpow, D. Oheimb, \textit{Javalght is Type-Safe - Definitely}
25\textsuperscript{th} ACM symposium on Principle of Programming Languages, Jan-1998.
