A Formal Specification of the Java Bytecode Semantics using the B method

Ludovic Casset[1]

Phone: +33 (0)4.42.36.54.52
Ludovic.Casset@gemplus.com

Jean Louis Lanet [2]

Phone: +33.(0)4.42.36.64.22
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 ressources, a specific Java has been specified for the Java card industry, known as the Java Card 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 Applet Firewall which is an important part of the JCRE is crucial. 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 insure that the implantation 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...). A defensive virtual machine where all the checks are performed at runtime has too poor performances. Thus, the smart card industry proposes an architectural design where the checks are performed off-card. The developpers 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 B method

---

[1] ESIL, Ecole Supérieure d'Ingénieurs de Luminy, département informatique, Luminy case 925 - 13288 Marseille cedex 09.

[2] Gemplus Research Lab, Av du Pic de Bertagne, 13881 Gémenos cedex.

[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 bytecode and its work aims to prove the runtime 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-98] 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 formally implement a defensive virtual machine. It is possible to prove that this model is equivalent to an angressive interpreter plus a sound bytecode verifier.

A new approach

Our approach is based on the Defensive Java Virtual Machine (DJVM) split in order 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 runtime 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 an 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 choosen to represent, at the control flow and data levels, most of the bytecode instructions. We use a small variant of this subset. The difference comes from the specialization of instructions **Istore** and **Iload** which load or store local variables of type integer. In these instructions, one can find instructions allowing integer manipulations and also instructions allowing object creations, initializations and uses. Informal specification of these instructions is given below (Fig.1).

By describing operational and static semantics, Freund and Mitchell prove that this subset is sufficient to study object initialization, flow and data-flow controls.

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

Fig.1. *Informal specification of the instruction subset.*

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*, type of the element of the stack,
- the *type frame* containing types of local variables.

The expected properties of the program are:

- confinement: a program cannot access objects or part of the program out of its workspace,
- stack access: no overflow or underflow during stack manipulation,
- initialisation: an object must be initialized once and only once. The access of an uninitialized object is not allowed.
- type correctness: it is forbidden to convert an integer into a référence; and no arithmic is allowed on pointer.

Assuming such constraints guarantee the correct state. Then, we use transfert functions related to each instruction to change 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 them. This algorithm is implemented in the off-card verifier.

The B model of the defensive machine

We explain the model on a particular instruction, the instruction **Inc**. An informal specification of this instruction can be: ***Inc*** *add one to the integer in the top of the stack*

and let the rest of the stack unchanged. 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.

Our abstract model represents the DJVM: we perform checks on *pc* domain and on types and then we execute the instruction (Fig.2)

ins_iload = **SELECT** (*methode* (*apc*) = *iload*)
THEN
IF (*apc* < **size** (*methode*) ∧ *top_stack* < *max_stack*
∧ *parametre*(*apc*) ∈ **dom**(*types_frames*)
∧ *types_frames*(*parametre*(*apc*))= *INTEGERS*)
THEN
apc := *apc* + 1
‖ *top_stack* := *top_stack*+1
‖ *types_stacks*:=*types_stacks*⩤{*top_stack*+1↦*INTEGERS*}
END
END;

Fig.2. *Instruction* ***Iload*** *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 (Fig.3).

∀*kd*.((*kd* ∈ **dom**(*methode*)) ∧ *methode*(*kd*) =*iload* ∧ *unchecked* = **FALSE**
⇒*kd* < **size**(*methode*) ∧ *SSTtop_stack*(*kd*) < *max_stack*
∧ *SSTtop_stack*(*kd*)=*SSTtop_stack*(*kd*+1)-1
∧ *SSTtypes*(*kd*+1)(*SSTtop_stack*(*kd*+1))=*INTEGERS*
∧ *SSTtypes*(*kd*)⩤{*SSTtop_stack*(*kd*)+1 ↦ *INTEGERS*}=*SSTtypes*(*kd*+1)
∧ *parametre*(*kd*) ∈ **dom**(*SSTtypes_frames*(*kd*)) ∧ *parametre*(*kd*)≤*max_frame*
∧ *parametre*(*kd*)≥0 ∧ *SSTtypes_frames*(*kd*)(*parametre*(*kd*))=*INTEGERS*
∧ *SSTtypes_frames*(*kd*)=*SSTtypes_frames*(*kd*+1))

Fig.3. *The invariant for* ***Iload*** *after two refinements*.

Then we obtain an offensive interpreter for the instruction **Iload,** *i.e.*, we just verify that previously the program passed successfully the verifier (see below).

ins_iload = **SELECT**(*methode* (*apc*) = *iload* ∧ *unchecked* = **FALSE**)
THEN
apc := *apc* + 1
‖ *top_stack* := *top_stack*+1
‖ *types_stacks*:=*types_stacks*⩤{*top_stack*+1↦*INTEGERS*}
END;

Fig.4. *The operation for instruction* ***Iload*** *in the last refinement.*

With this approach, we bring to the fore that we split the original defensive machine. We introduce another abstract machine to initialize the variable unchecked by performing static checks on the bytecode. This machine is in fact the specification of our verifier. The last refinement of the defensive machine appears to be our offensive interpreter. We have 489 Proof Obligations (PO). the project is entirely proved.

The fixed point calculus for type correctness

Computing the right type for a given *pc* is rather difficult because several paths can lead to this *pc* in the tree of possible executions. So, as performing the verification, one must checks that the type obtained is the right one and no error will occur during execution.

The method we use is to compute a fixed point. It means that, considering all paths leading to a given pc, we search the type satisfying all of them. If the program is correct, such a type exists and is usable. Otherwise, checks raise an error.

To complete such a work, we introduce a lattice (Fig.5) over types used in the bytecode. In our study, we have three different usable types: INTEGERS, Addri and Addr. To obtain a complete lattice [Dwy-95], we add a top value TOP, a bottom value $\perp$, a partial-order $\subseteq$ and a binary operator *Meet* Π. We assume that TOP and $\perp$ are non usable type.

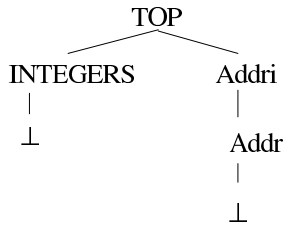

Fig.5. *The complete lattice.*

According to the partial-order over the lattice, we have the relations:

- $\perp \subseteq$ INTEGERS $\subseteq$ TOP,
- $\perp \subseteq$ Addr $\subseteq$ Addri $\subseteq$ TOP.

With such a lattice, we can solve the flow equations:

$$Types[r] = \perp$$
$$\forall\, n \neq r,\ Types[n] = \Pi \{ f_i\,(Types[i]) \mid i \in Preds(n) \}$$

where r is the root node of the tree, *Types* gives the type of the node n, f_i is the transfert function of the node i and *Preds* the set of all predecessors of node n. The transfert function associated to each node fits with the instruction of the given node. In our study, we have ten instructions and ten transfert functions.

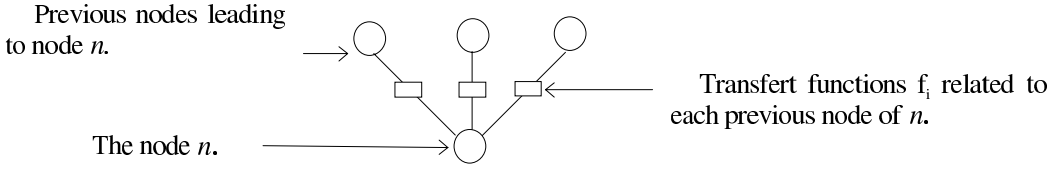

Fig.6. *The representation of the flow equation problem.*

We choose the algorithms presented by Dwyer [Dwy-95] because he proves that his algorithms converge on the greatest fixed point. The complexity of his algorithm is $O(h.N^2)$ where h is the height of the tree and N is the number of nodes.

In the B model we present the specification of the flow equation for the fixed point calculus. First, we introduce the *Meet* operator which, in fact represents the complete lattice over types, the partial-order and the binary operator (Fig.7).

$$
\begin{array}{l}
Meet \in JTYPES \times JTYPES \rightarrow JTYPES \wedge \\
\forall tt.(\ tt \in JTYPES \Rightarrow \forall tp.(tp \in JTYPES \Rightarrow Meet(tt,tp)=Meet(tp,tt))) \wedge \\
\forall tt.(\ tt \in JTYPES \Rightarrow Meet(tt,tt)=tt) \wedge \\
Meet(addr,addri)=addri \wedge Meet(addr,INTEGERS)=TOP \wedge Meet(addr,TOP)=TOP \wedge \\
Meet(addri,TOP)=TOP \wedge Meet(addri,INTEGERS)=TOP \wedge Meet(INTEGERS,TOP)=TOP
\end{array}
$$

Fig.7. *The Meet operator definition*

Then, we specify the set *Preds*. This set is made of all predecessors of a given *pc*. In our model, we add a new feature. For a given *pc*, we associate its predecessors and, for each predecessor we associate the supposed type it attributes to the different variables in the stack and in the frames through the transfert function (Fig.8).

$$
\begin{array}{l}
Preds \in 1..\mathbf{size}(methode) \rightarrow (1..\mathbf{size}(methode) \nrightarrow JTYPES) \\
Preds(ka+1)=Preds(ka+1) \lhd \{ka \mapsto INTEGERS\}
\end{array}
$$

Fig.8. *Definition and example of use of Preds.*

Finally, we translate the flow equation as follow (Figure 9). For each *pc*, we add this element to *Preds* as predecessors of *pc+1*. We associate to *pc* the type of *pc+1* using this path. Then, we compute a partial fixed point thanks to the set *Preds* by combining types through the complete lattice.

$$
\begin{array}{l}
\forall ii.(ii \in \mathbf{dom}(Preds(ka+1)) \wedge Preds(ka+1) \neq \varnothing \wedge \\
Preds(ka+1)(ii) \neq SSTtypes(ka+1)(SSTtop\_stack(ka+1)) \\
\Rightarrow SSTtypes(ka+1)=SSTtypes(ka) \lhd \{SSTtop\_stack(ka) \\
\mapsto Meet(SSTtypes(ka+1)(SSTtop\_stack(ka+1)),Preds(ka+1)(ii))\})
\end{array}
$$

Fig.9. *The flow equation in B*

At the end of the program, type of variables for every *pc* is computed. If no unsuable type remains, the program is correct for types point of view. Otherwise, the verifier raises an error.

Conclusions and Future Work

We entirely proved the defensive machine model at the flow and type control level. We are modelizing the two different parts of the defensive machine, the verifier and the interpreter. The work is already done for the flow control and we are integrating the type control for the instruction subset and in particular the calculus of the fixed point as presented. The integration of the fixed point calculus is proved at 90% and we are still working on it to improve the model.

In the meantime, we use the results of A. Requet [Req-98] on the JavaCard 2.1 bytecode specification. With his work, we bring to the fore the static and the dynamic semantics of each real instruction. Integrating all these studies, we complete our model to present a defensive machine, a bytecode verifier and an interpreter, matching the JavaCard 2.1 standard.

References

[Abr-96] J. R. Abrial, *The B Book. Assigning Programs to Meanings*, Cambridge University Press 1996.

[Coh-96] Cohen, *Defensive Java Virtual Machine Specification*
`http ://www.cli.com/software/djvm`

[Dro-97] S. Drossoppoulou, S. Eisenbach, *Java is Type Safe - Probably*.

[Dwy-95] M. Dwyer, *Data Flow Analysis for verifying correctness properties of concurrents programs*, Phd thesis, University of Massachusetts, Sept 95.

[Fre-98] S. N. Freund, J. C. Mitchell *A type System for Object Initialization in the Java Bytecode Language* In. Proc. Conf. On Object-Oriented Programming, Systems, Languages, and Applications, pages 310-328. ACM Press 1998. `http ://theory.standford.edu/~freunds`.

[Qia-98] Z. Qian, *Least Types for Memory Locations in Java Bytecode*, Kestrel Institute, Tech. Report, 1998.

[Nip-98] T. Nipkow, D. Oheimb, *Javalight is Type-Safe - Definitely*
25th ACM symposium on Principle of Programming Languages, Jan-1998.

[Req-98] A. Requet, *Spécification Formelle en B d'un Convertisseur de Bytecode pour Applets Javacard*, Rapport de DEA, Université de Nantes, Septembre 1998.

[Sir-98] E. G. Sirer, A. J. Gregory, B. N. Bershad, *Kimera: A Java System Architecture*.
`Http ://kimera.cs.washington.edu/`, 1998

[Sym-97] D.Syme, *Proving Java Type Soundness*, Technical report, University of Cambridge Computer Laboratory, 1997.