Towards a General Framework for Formal Reasoning about Java Bytecode Transformation

Razika LOUNAS[1], Mohamed MEZGHICHE[1], and Jean-Louis LANET[2]

[1] University of Boumerdes, ALGERIA
{lounas_razika, mohamed.mezghiche}@umbb.dz
[2] University of Limoges, FRANCE
jean-louis.lanet@xlim.fr

Abstract. Program transformation has gained a wide interest since it is used for several purposes: altering semantics of a program, adding features to a program or performing optimizations. In this paper we focus on program transformations at the bytecode level. Because these transformations may introduce errors, our goal is to provide a formal way to verify the update and establish its correctness. The formal framework presented includes a definition of a formal semantics of updates which is the base of a static verification and a scheme based on Hoare triples and weakest precondition calculus to reason about behavioral aspects in bytecode transformation.
Key words:Java bytecode transformation, formal semantics, formal reasoning.

1 Introduction

Program transformation is a technique used for several proposes: altering semantics of a program, performing optimizations or adding features. Several tools were developed in this domain, for example, Java Syntactic Extender (JSE) [11] and BCEL [12] for Java. However, in some cases, the source code is not available (or not distributed). Transforming a program at bytecode level is an interesting alternative since several languages like Java, Java Card or $C\sharp$ are based on virtual machines executing bytecode. Besides, in transformations at bytecode level we don't need recompiling (which may take time) as in the case of transformations at source code level. On the other hand, bytecode level transformation is more complex than source-level manipulation to the users because they have to know bytecode language very well and because of the many low-level details one needs to deal with, in contrast with source code level.

Bytecode transformation is used in several applications. In [4], the authors developed an algorithm to ensure portable thread migration in Java. This algorithm is based on bytecode transformation. Bytecode is transformed in order to enable programs to save and restore their execution state after migration through the network. Another purpose for bytecode transformation is presented in [5] where a framework based on bytecode transformation is developed in order to enable Java applications to perform CPU management

Transforming a program may occur at runtime. The update is then said to be dynamic (Dynamic Software Update: DSU). In [6] [7], the authors presented a system to perform dynamic software update: while the Java Card virtual machine is executing the program, the bytecode is updated. In [3], a tool is developed in order to perform runtime bytecode update for Smalltalk.

This large interest of bytecode transformation and its use in many applications lead to the question of its correctness. In fact, a transformation may introduce an error which may alter the bytecode in a different way from that is expected by the programmer. In addition, some applications where the update occurs are critical, such as in Java Card. In these applications where security issues are involved the update must pass certification procedure for example Common Criteria [18]. For a certain certification level one has to provide a formal proof of the security mechanism implemented. A formal way to reason about transformations and verify their validity is then necessary.

In this work, we present a first step for a general framework for reasoning about bytecode transformation. We focus on Java bytecode and the system presented in [7] called embedDSU: a system to update dynamically Java Card applications. But this is not restrictive: the framework developed may be applied to other systems and for this it is general. The framework is divided in two parts: we propose an approach for a static analysis by defining a formal semantics for update to ensure the absence of type errors and then in the second part we propose an approach to reason about behavioral aspects using Hoare triples and predicate transformations.

This paper is organized as follow: in section 2 we give an overview of embedDSU. Section 3 introduces a verification approach through a static analysis of the bytecode. In section 4 we present the part of the framework which talks about reasoning on the behavioral aspects of updates. We present related work in section 5 and conclude in section 6.

2 Overview of EmbedDSU

EmbedDSU [6] [7], is a software-based DSU technique for Java based smart cards which relies on the Java virtual machine. It is based on the modification of an embedded virtual machine. EmbedDSU is divided in two parts: off-card and on-card:

- In off-card, in order to apply the update only to the parts of the application that are really affected by the update, a module called DIFF generator determines the syntactic changes between versions of classes. The changes are expressed using a Domain Specific Language (DSL). Then, the DIFF file result is transfered to the card and used to perform the update.
- The on-card part is divided into two layers: 1) Application Layer: The binary DIFF file is uploaded into the card. After a signature check with the wrapper, the binary DIFF is interpreted and the resulting instructions are transferred to the Patcher in order to perform the update. The Patcher has the role of initializing update data structures. These data structures are

read by the updater module to determine what to update and how to update, by the safeUpdatePoint detector module to determine when to apply the update and by the rollbacker to determine how to return to the previous version in case of update failure. All these issues pass through the introspection of the virtual machine. 2)System Layer: The modified virtual machine supports the followings features: (1) Introspection module which provides search functions to go through VM data structures like the references tables, the threads table, the class table, the static object table, the heap and stack frames for retrieving information necessary to other modules; (2) updater module which can modify and update object instances, method bodies, class metadata, references, affected registers in the stack thread and affected VM data structures; (3) SafeUpdatePoint detector module which permits to detect safe point in which we can apply the update by preserving coherence of the system.

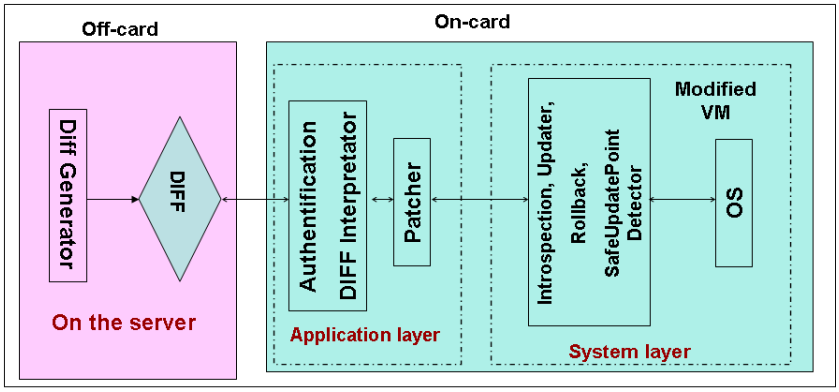

Fig. 1. Architecture of EmbedDSU

EmbedDSU updates three principal parts:

- The bytecode: The process updates first the bytecode of the updated class and the meta data associated with it: constant pool, fields table, methods table...
- The heap: The process updates the instances of the updated class in the heap, obtains new references for modified objects and updates instances using these references.
- The frames: The process updates in each frame in the thread stack the references of updated objects to point to new instances.

This paper addresses the first part: bytecode update at the method level. The types of updates that may occur are: adding, modifying or suppressing bytecode

instructions, changing the signatures of a method or modifying local variables. These updates are contained in the DIFF file (also called patch) which indicates exactly which is the update and where it occurs in the bytecode. For example, when adding an instruction, the patch informs the system which instruction to add and where to add it (information about the program pointer)

3 Updated Bytecode analysis for static verification

We present an approach for transformation validation based on static semantics of bytecode (figure 2) in order to avoid type errors in transformed programs. From a first versionBC_V1 and a second version BC_V2 (Version one transformed), we have a DIFF file. This DIFF file is applied to the first version. We obtain a version $BC\_V1.2$ (annotated BC_V1 on the figure). The goal of the validation is to establish that $BC\_V1.2$ and BC_V2 are semantically equivalent by comparing $V\_Sem\ S1$ and $V\_Sem\ S2$ representing the semantics of $BC\_V1.2$ and BC_V2 respectively.

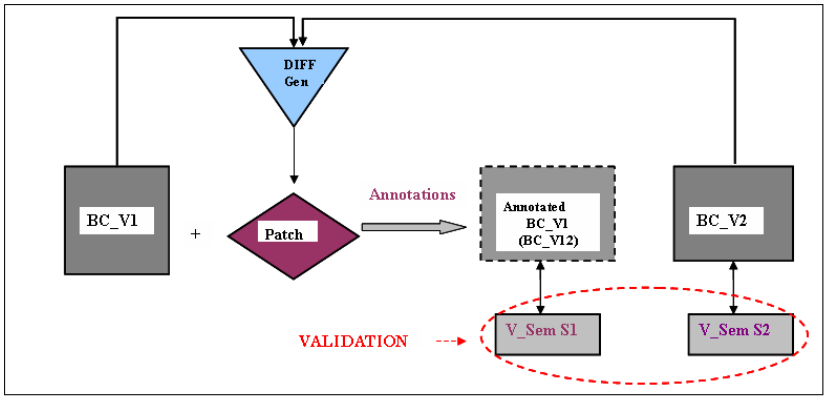

Fig. 2. Approach for static verification

The application of the DIFF to the first version is modeled syntactically as annotations (figure 3). We insert annotations to indicate instructions addition and suppression. For example, `Del \%2` : deletes the instruction at program counter (pc) 2 and `add \%6 inc`, adds the instruction inc at pc 6.

3.1 The language

For the definition of the static semantics, we adopt the formalism used by Freund and Mitchell [1]. The authors define a type system for a small subset of Java

bytecode. We extend this subset with instructions to indicate updates called update instructions (Upd_instr)for instruction addition, deletion and modification. In this definition, x is a local variable; L is an instruction address; A is a class name; f is a field name; l is a method name and pc the program counter.

```
Instruction::= | pop | if L| store x| load x| new A| goto L|inc
               |add |invokevirtual A l t|getfield  A f t|putfield A f t
Upd_Instr::= Add_Inst Instruction at pc
             |Dlt_Inst Instruction at pc |Mod_Inst Instruction at pc
```

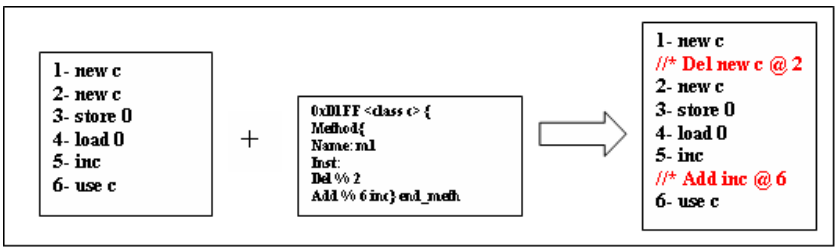

Fig. 3. Bytecode annotation with update instructions

3.2 Formal semantics

We propose a static semantics to express the effect of update instructions on a configuration of the bytecode. In the four rules shown in Fig 4 , F is a mapping from a program point to a mapping from a frame variable to a type. S is a mapping from a program point to an ordered sequence of types, i denotes a program point or an address of code. The map F_i gives a type of local variables at program point i. The string S_i gives the types of entries in the operand stack at program point i. These F and S are useful to our semantics since they contain typing information about valid local variables and entries in the operand stack respectively. SD represent the stack depth and M (mapping) is a function that associate a number to each line. Dom is the set of addresses used by the method. A configuration at line i is represented by $< (F, S, SD, M), i >$.

For illustration, the add of the instruction `new A` at line $i + 1$ allows us to obtain a new configuration if the stack depth is incremented, local variables are not affected and in the stack, the type A is inserted. $M2$ is the result of operations on $M1$. These operations which represent manipulations on bytecode are: range and shift. The operation range extracts from a mapping $M1$ a part $M2$ included between line n and line m. The operation shift shifts a part from

$$\frac{Add\_inst\ \ load\ x(i+1)\quad SD_{i+1}=SD_i+1\quad S_{i+1}=F_i[x].S_i\ \ F_{i+1}=F_i\ PC\_MAX++\quad M2=Add\_inst(M1,load\ x,i+1)\quad i+1\in DOM(BC)\ \ x\in VAR(BC)}{<(F_t,S_t,SD_t,M1),i>\rightarrow<(F_{t+1},S_{t+1},SD_{t+1},M2),i+2>}$$

$$\frac{Add\_inst\ \ new\ A(i+1)\quad SD_{i+1}=SD_i+1\quad S_{i+1}=A.S_i\ \ F_{i+1}=F_i\quad M2=Add\_inst(M1,new\ A,i+1)\quad PC\_MAX++\ \ i+1\in DOM(BC)}{<(F_t,S_t,SD_t,M1),i>\rightarrow<(F_{t+1},S_{t+1},SD_{t+1},M2),i+2>}$$

$$\frac{Add\_inst\ add(i+1)\quad SD_{i+1}=SD_i-1\quad S_i=int.int.S_0\Rightarrow S_{i+1}=int.S_0\quad M2=Add\_inst(M1,add,i+1)\quad i+1\in DOM(BC)\ \ F_{i+1}=F_i}{<(F_t,S_t,SD_t,M1),i>\rightarrow<(F_{t+1},S_{t+1},SD_{t+1},M2),i+2>}$$

$$\frac{Add\_inst\ getfield(A,f,t)(i+1)\quad SD_{i+1}=SD_i\quad S_i=A.S_0\Rightarrow S_{i+1}=t.S_0\quad M2=Add\_inst(M1,getfield(A,f,t),i+1)\quad PC\_MAX+3\ \ F_{i+1}=F_i}{<(F_t,S_t,SD_t,M1),i>\rightarrow<(F_{t+1},S_{t+1},SD_{t+1},M2),i+3>}$$

Fig. 4. Rules for update instructions

a mapping between n and m for p positions. Both operations are of type:
$mapping * int * int* \rightarrow mapping$

In order to take into account jumps in bytecode transformation, we define two other operations: look_for_jumps which returns from a mapping a list of jumps instructions represented by their line number and the operation update_jumps to update jump instruction:
$Look\_for\_jumps : mapping \rightarrow intlist$
$Update\_jumps : mapping * intlist * int \rightarrow mapping$

Due to a matter of space, we don't give in this paper rules for ordinary bytecode instructions, rules for instruction suppression and the remaining rules for instruction addition.

4 An approach for reasoning about transformations

In this section, we present an approach to reason about behavioral aspects of transformations on bytecode. This approach is based on bytecode specification in term of preconditions and postconditions and on predicate transformation to generate verification conditions. We give first some definitions before presenting the scheme of the approach.

4.1 Definitions

Definition 1. Hoare triplet *A Hoare triple is the basic object in Hoare logic [10]. It has the form of* $\{P\}\ S\ \{Q\}$ *where* P *and* Q *are logical formulas and* S *a program. The interpretation of* $\{P\}\ S\ \{Q\}$ *regarding partial correctness is: If* S *is executed in a state in which* P *holds, then it terminates in a state in which* Q *holds unless it aborts or runs forever. The interpretation in total correctness is: if* S *is executed in a state in which* P *holds, then it terminates in a state in which* Q *holds.*

Reasoning in Hoare logic is based on inference rules. Here is an example of a general rule:

$$\frac{P \Rightarrow P1 \quad \{P1\}\ S\ \{Q1\} \quad Q1 \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Definition 2. Weakest precondition (WP) calculus *The Weakest Precondition calculus [9] is a predicate transformer that takes a code S and a postcondition Q and returns a precondition. We write WP(S, Q): "the weakest precondition of S regarding Q". WP(S,Q) is a precondition for S that ensures Q as a postcondition. It is weakest in the sense that if we take any P such that* $\{P\}\ S\ \{Q\}$ *then* $P \Rightarrow WP(S,Q)$*. It satisfies* $\{WP(S,Q)\}\ S\ \{Q\}$*.*

Definition 3. Strongest postcondition (SP) calculus *The Strongest Postcondition calculus [9] is a predicate transformer that takes a precondition P and a code S and returns a postcondition. We write SP(P, S) as "the strongest postcondition of S regarding P". SP(P, S) is a postcondition for S that is ensured by precondition P. It is strongest in the sense that if we take any Q such that* $\{P\}\ S\ \{Q\}$ *then* $SP(P,S) \Rightarrow Q$*. It satisfies* $\{P\}\ S\ \{SP(P, S)\}$*.*

4.2 Approach Description

We propose an approach based on the definition of the concept of ***triple transformation***. It represents the idea that an update of an existing method M1 with precondition and postcondition P1 and Q1 results of a new method M2 with a new specification P2 and Q2. The triple {P1} M1 {Q1} is transformed via the update to a new triple {P2} M2 {Q2}. The approach defines these concepts: *initial triplet*, *target triplet* and *calculated triplet*:

Definition 4. Initial triplet *An initial triple* $\{P1\}\ M1\ \{Q1\}$ *represents a method M1, its precondition P1 and its postcondition Q1 at the initial state, that means before an update. This triple represents a method and its specification in the running code.*

Definition 5. Target triplet *A target triple* $\{P2\}\ M2\ \{Q2\}$ *represent a new version M2 of the initial version M1 and its specifications P2 and Q2. It is the goal of the update as it is written by the programmer. The methods M1 and M2 are written in bytecode. Pre/post-conditions (P1, Q1, P2 and Q2) are written using existing specification languages and tools by the programmer.*

Definition 6. Calculated Triplet *A calculated triple is a triple obtained starting from an initial triple with the application of the transformations contained in a patch (list of update instruction). It is the result of the transformation of an initial triple.It is calculated using the Transform_triple algorithm.*

As shown in figure 5, the approach is based on three steps:

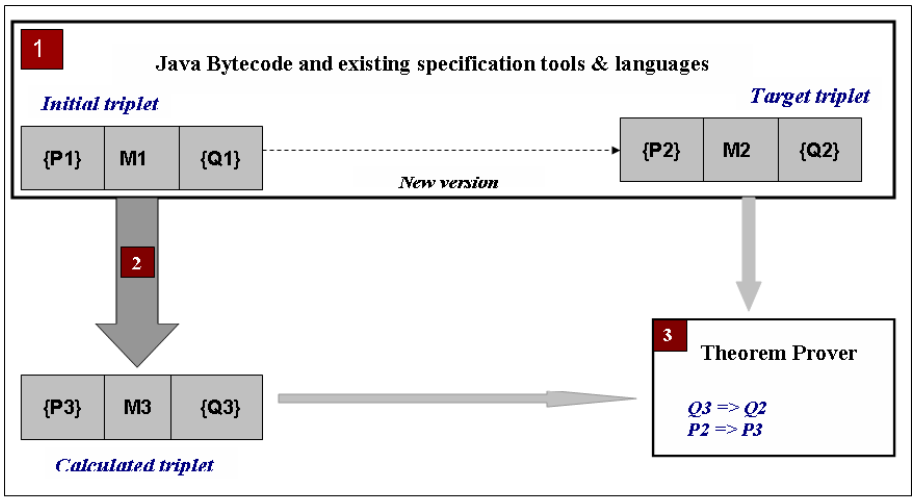

Fig. 5. The proposed Approach for reasoning

- **Step (1):** ***Programming and specification*** The initial code M1 is written in bytecode and the specification (pre/post-condition) is written using existing specification language and tools. The new version of M1 called M2 is written in bytecode. The desired specification of the update is expressed by the programmer using existing tools too and is expressed in term of pre/post-conditions of the new code M2.
- **Step (2):** ***Triple transformation*** Given an initial triple and a list of update instructions contained in a patch, this calculus transforms the initial triple step by step. Each step correspond to the application of an update instruction. We take the case of instruction insertion. The application of an update instruction returns an intermediate triple that will be taken as an argument of the calculus for the next update instruction. This is represented as a recursive algorithm called *Transform_triple*. It is based on the predicate transformation calculus: weakest precondition (wp) and strongest postcondition (sp).

```
Transform_triple (p1, q1, m1, patch1) = match patch1 with
 |[]->return (p1,q1)
 | Add_instr (X,i)::patch2-> let n=last_line(m1) in
          let m2=m1(+)(X,i) in let wp1 = WP (m1[i,n], q1) in
           if wp1 != WP(m2[i+1,n], q1)
              then Raise Exception
           else
           let p2=WP(m2 [ 1,i ], wp1) in
           let sp1 = SP (m1 [1,i-1], p1) in
            if sp1!=SP (m2[1,i-1], p1)
```

```
  then Raise Exception
else
 let q2 = SP (m2[I,n], sp1) in
  Transform_triple(p2,q2,m2,patch2)
```

The algorithm Transform_triple represents the application of a patch *patch1* to a method *m1* with specification *p1* and *q1*.The patch contains update instructions about inserting instructions *(X)*at an indicated line i (ADD_instr (X,i). As a result of the insertion of X, (represented by *(+)*), the code *m1* is transformed to m2. Then Transform_triple calculates a new precondition for *m2* using the *wp* calculus starting from the last line of *m2* and calculates a new postcondition using the *sp* calculus. The result is an intermediate triple {P2} m2 {Q2} that will be taken as an argument in the recursive call with the remaining patch *patch2*. The algorithm stops when the patch is empty and raises exceptions when errors occur in the calculus.

– **Step (3): *Implication proof*** The calculated triple needs to be matched to the target triple to establish the correctness of the transformation. The property that needs to be shown here is implication. We show that the calculated postcondition implies the target postcondition and that the target precondition implies the calculated postcondition: $Q3 \Rightarrow Q2 \bigwedge P2 \Rightarrow P3$.

5 Related work

Several studies have been conducted in order to use static semantics to prevent type errors in bytecode. Our work extends the formalism presented in [1]. This work defined semantics and a type system to study object initialization in bytecode. The original idea was developed in [14] to study bytecode subroutines. In [13], the authors extended the work [1] to bytecode subroutines, virtual method invocation and exceptions. On the behavioral side, using predicate transformation to reason about bytecode properties has been studied in [2]. The authors presented a verification condition generator for bytecode formalized in the Coq proof assistant and based on weakest precondition calculus. Another work using wp to generate verification conditions from an annotated bytecode is presented in [16] [17]. The use of strongest postcondition calculus is not as popular as the wp calculus. A study is presented in [15] as a basis for formal reverse engineering for an imperative language. Our work is close to [1] in the sense of the use of static semantics to analyze bytecode. The specificity of our work is the definition of semantics for updates. We use predicate transformation to reason about bytecode properties using existing tools for specification and proofs. Our framework uses both weakest precondition and strongest postcondition to reason about transformations.

6 Conclusion and future work

In this paper we propose a general framework for a formalization, verification and reasoning about Java bytecode transformation. We gave first an approach for verification by analyzing the modified bytecode to ensure absence of type errors. We gave then an approach for reasoning about bytecode transformation by using predicate transformations. The aim of the two methods combined is to provide a complete framework that provides the two aspects: static and behavioral. The second method focuses on behavioral aspects and aims to the definition to a rich assertion language to capture dynamic update features and effects on execution structures such as frames and objects in the heap (in a Java Card virtual machine for example). These structures are not available in the static aspect of the framework.

This work is on-going. Our aim immediately is to complete the implementation by extending the language to other instructions in bytecode and to the other possible transformations for methods (adding arguments for example). On the other side, we aim to complete the work concerning behavioral aspects by defining algorithms to take into account deleting instructions in predicate transformation and to choose a configuration of existing tools for specification and reasoning. The verification presented is implemented using the functional language Ocaml. We aim to use mathematical reasoning to prove its correctness. In the longer term, we wish to use a proof assistant to reason about bytecode transformation.

References

1. S-N.Freund J-C.Mitchell. A Type System for Object Initialization In the JavaTM Bytecode Language. In OOPSLA '98.
2. B. Grgoire and J-L. Sacchini. *Combining a verification condition generator for a bytecode language with static analysis*. In LNCS 4912, 2008.
3. M. Denker, S. Ducasse and E. Tanter. *Runtime bytecode transformation for Smalltalk*. In Computer Languages, Systems and Structures 32 (2006).
4. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. *Bytecode Transformation for Portable Thread Migration in Java*. Department of Information Science, Faculty of Science, University of Tokyo,2000.
5. W. Binder and J. Hulaas. *Java Bytecode Transformations for Efficient, Portable CPU Accounting*. Electronic Notes in Theoretical Computer Science 141 (2005)
6. A-C. Noubissi, *Mise á jour dynamique et scurisée de composants systéme dans une carte á puce*, PhD Thesis 2011, University of Limoges, France.
7. A.C. Noubissi, J. Cartigny and J-L Lanet. *Hot Updates for Java Based Smart Cards*. In HotSwUp'11 Third Workshop on Hot Topics in Software Upgrades, Hannover, Germany, April 2011
8. A-C.. Noubissi, J. Cartigny and J.-L. Lanet. *Incremental Dynamic Update For Java-based Smart Cards*, ICONS 2010, April 2010, Les Menuires, France
9. Edsger W. Dijkstra. *Notes on structured programming*. In O .- J. Dahl, E.W. Dijkstra and C.A.R. Hoare, editors, Structured Programming. Academic Press, 1972.

10. C.A.R. Hoare. *An axiomatic basis for computer programming*. Communication of the ACM, 12: 567-580, 583, 1969.
11. J. Bachrach and K. Playford . *The Java syntactic extender (JSE)*. Proceedings of OOPSLA 01,ACMSIGPLAN notices, vol. 36 (11), 2001.
12. M. Dahm. *Byte code engineering*. In: Proceedings of JIT 99, Dsseldorf, Deutschland, 1999
13. S-N. Freund and J-C. Mitchell. A Type System for the Java Bytecode Language and Verifier. Journal of Automated Reasoning 30: 271321, 2003
14. R. Stata and M. Abadi. *A type system for Java bytecode subroutines*. In ACM Transactions on Programming Languages and Systems 21(1) (1999).
15. G.C. Gannod and B.H.C. Cheng.*Strongest postcondition semantics as the formal basis for reverse engineering*.In the journal of automated software engineering, vol 3. 1/2 Kluwer Academic Publishers, 1996.
16. L. Burdy and M. Pavlova. *Java bytecode specification and verification*. In SAC 2006: 1835-1839
17. L. Burdy, M. Huisman and M. Pavlova. *Preliminary Design of BML: A Behavioral Interface Specification Language for Java Bytecode*. In FASE 2007: 215-229
18. Common Criteria . http://www.commmoncriteria.org/.

APPENDIX: More rules for static semantics

A. For instructions addition

$$\frac{\begin{array}{l}Add\_inst\ \ goto\ L(i+1)\\ SD_{i+1}=SD_i\\ S_{i+1}=S_i\ \ F_{i+1}=F_i\ PC\_MAX++\\ M2=Add\_inst(M1,goto\ L,i+1)\\ i+1,L\in DOM(BC)\end{array}}{<(F_i,S_i,SD_i,M1),i>\rightarrow\ <(F_{i+1},S_{i+1},SD_{i+1},M2),i+2>}$$

$$\frac{\begin{array}{l}Add\_inst\ \ pop\ (i+1)\\ SD_{i+1}=SD_i-1\\ S_i=t.S_0\rightarrow S_{i+1}=S_0\quad F_{i+1}=F_i\\ M2=Add\_inst(M1,pop,i+1)\\ PC\_MAX++\quad\ i+1\in DOM(BC)\end{array}}{<(F_i,S_i,SD_i,M1),i>\rightarrow<(F_{i+1},S_{i+1},SD_{i+1},M2),i+2>}$$

$$\frac{\begin{array}{l}Add\_inst\ \ store\ x(i+1)\\ SD_{i+1}=SD_i-1\ \ PC\_MAX++\\ S_i=t.S_0\ \ F_{i+1}=F_i[x\leftarrow t]\ S_{i+1}=S_0\\ M2=Add\_inst(M1,store\ x,i+1)\\ i+1\in DOM(BC)\ x\in VAR(BC)\end{array}}{<(F_i,S_i,SD_i,M1),i>\rightarrow\ <(F_{i+1},S_{i+1},SD_{i+1},M2),i+2>}$$

$$\frac{\begin{array}{l}Add\_inst\ putfield(A,f,t)(i+1)\\ SD_{i+1}=SD_i-2\\ S_i=t.A.S_0\Rightarrow S_{i+1}=S_0\\ M2=Add\_inst(M1,putfield(A,f,t),i+1)\\ PC\_MAX+3\quad\ F_{i+1}=F_i\ \ i+1\in DOM(BC)\end{array}}{<(F_i,S_i,SD_i,M1),i>\rightarrow<(F_{i+1},S_{i+1},SD_{i+1},M2),i+3>}$$

$$\frac{\begin{array}{l}Add\_inst\ \ invokevirtuel(A,l,t)(i+1)\\ SD_{i+1}=SD_i-(card^*(dom^*(t))+1)\\ S_{i+1}=tn_1.tn_2\ldots tn_n.S_0\rightarrow S_{i+1}=S_0\\ M2=Add\_inst(M1,invokevirtuel(A,l,t),i+1)\\ i+1\in DOM(BC)\ \ F_{i+1}=F_i\ \ PC\_MAX+3\end{array}}{<(F_i,S_i,SD_i,M1),i>\rightarrow\ <(F_{i+1},S_{i+1},SD_{i+1},M2),i+2>}$$

***Notations**

- **dom**: represents the domain of the invoked function (types of its arguments)
- **card:** represents the number of elements in the domain.

B. For instructions suppression

$$\frac{\begin{array}{l} Dlt\_inst \;\; pop \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD^{**}(a, M2[i+1]) \\ S_i = t.S_0 \rightarrow [M2]S_{i+1} = Effects\_STK^{**}(M2[i+1], t.S_0) \\ (M2)^* F_{i+1} = Effects\_F^{**}(M2[i+1], F_i) \\ M2 = Dlt\_inst(M1, pop, i+1) \\ i+1 \in DOM(BC) \;\; PC\_MAX -- \end{array}}{<(F_i,S_i,SD_i,M1),i> \rightarrow <(F_{i+1},S_{i+1},SD_{i+1},M2),i+Instr\_length(M2[1+i])>}$$

$$\frac{\begin{array}{l} Dlt\_inst \;\; new \; A \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ S_i = t.S_0 \rightarrow [M2]S_{i+1} = Effects\_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ M2 = Dlt\_inst(M1, new \; A, i+1) \\ i+1 \in DOM(BC) \;\; PC\_MAX -- \end{array}}{<(F_i,S_i,SD_i,M1),i> \rightarrow <(F_{i+1},S_{i+1},SD_{i+1},M2),i+Instr\_length(M2[1+i])>}$$

$$\frac{\begin{array}{l} Dlt\_inst \;\; load \; x \;\; (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ S_i = S_0 \rightarrow (M1)S_{i+1} = t.S_0, \;\; (M1)F_{i+1}(x) = t \rightarrow \\ (M2)S_{i+1} Effects\_STK(M2[i+1], S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ M2 = Dlt\_inst(M1, load \; x, i+1) \\ i+1 \in DOM(BC) \;\; PC\_MAX -- \end{array}}{<(F_i,S_i,SD_i,M1),i> \rightarrow <(F_{i+1},S_{i+1},SD_{i+1},M2),i+Instr\_length(M2[1+i])>}$$

****Notations:**

- Effect_STK (a,b): represents the effect of the instruction a on the stack a.
- Effect_F(a,b): represents the effect on the instruction a on F.
- Effect_SD(a,b): represents the effect of then instruction b on the stack depth a.
- (M2)F: represents F according to the mapping M2.