A formal verification of dynamic updating in a Java-based embedded system

Razika Lounas*

LIMOSE Laboratory,
Faculty of Sciences,
University of M'hamed Bougara of Boumerdes,
Avenue de l'indépendance, 35000,
Boumerdes, Algeria
and
Xlim Laboratory,
University of Limoges,
123 Avenue Albert Thomas, 87700,
Limoges, France
Email: razika.lounas@univ-boumerdes.dz
*Corresponding author

Mohamed Mezghiche

LIMOSE Laboratory,
Faculty of Sciences,
University of M'hamed Bougara of Boumerdes,
Avenue de l'indépendance, 35000,
Bumerdes, Algeria
Email: mohamed.mezghiche@univ-boumerdes.dz

Jean-Louis Lanet

INRIA LHS-PEC,
263 Avenue Général Leclerc,
35000, Rennes, France
Email: jean-louis.lanet@inria.fr

Abstract: Dynamic software updating (DSU) consists in updating running programs on the fly without any downtime. This feature is interesting in critical applications that must run continuously. Because updates may lead to safety errors and security breaches, the question of their correctness is raised. Formal methods are a rigorous means to ensure the correctness required by applications using DSU. In this paper, we present a formal verification of correctness of DSU in a Java-based embedded system. Our approach is based on three major contributions. First, a formal interpretation of the semantic of update operations to ensure type safety of the update. Secondly, we rely on a functional representation of bytecode, the predicate transformation calculus and a functional model of the update mechanism to ensure the behavioural correctness of the updated programs. It is based on the use of Hoare predicate transformation to derive a specification of an updated bytecode. Thirdly, we use the functional representation to model the safe update point detection mechanism. This mechanism guarantees that none of the updated methods are active. This property is called activeness safety. We propose a functional specification that allows to derive proof obligations that guarantee the safety of the mechanism.

Keywords: dynamic software updating; DSU; formal verification; weakest precondition calculus; dynamic update safety; critical systems.

Reference to this paper should be made as follows: Lounas, R., Mezghiche, M. and Lanet, J-L. (2017) 'A formal verification of dynamic updating in a Java-based embedded system', *Int. J. Critical Computer-Based Systems*, Vol. 7, No. 4, pp.303–340.

Biographical notes: Razika Lounas is a final year PhD student. She has the grade of Teacher-Researcher at the Computer Science Department at the University of Boumerdes, Algeria since 2009. She is also a member of the LIMOSE research laboratory at the same university and a member of the Xlim Laboratory at the Limoges University, France. She received her Magister Diploma form the Boumerdes University in 2009. Before that, she received her Diploma of Computer Engineering at the Tizi Ouzou University, Algeria. Her main research interests are dynamic software updating, formal methods and Java card applications.

Mohamed Mezghiche is a Professor in the Computer Science Department at the University of Boumerdes Algeria. He is also a Team Leader and the Director of the LIMOSE Laboratory at the same university. He received his PhD in Theoretical Computer Science from the University Paris 6 (France). His research interests are formal methods, program certification, theorem proving, logic and functional programming.

Jean-Louis Lanet is the Director of the High Security Labs of the Inria-RBA. Previously, he was a Full Professor in the Computer Science Department at the University of Limoges (2007–2014). He was also the Team Leader of Smart Secure Device (SSD) research group. Prior to that, he was a Senior Researcher at the Gemplus Research Labs (1996–2007). During this period he spent two years at INRIA (Sophia-Antipolis) (2003–2005) as an Engineer at the Direction des Relations Industrielles (DirDRI) and as a Senior Research Associate in the Everest team. He started his career as a researcher at Elecma, Electronic division of the Snecma, now a part of the Safran group (1984–1995) and his field of research is on jet engine control. His research interests include security of small systems like smart cards and software engineering.

This paper is a revised and expanded version of a paper entitled 'An approach for formal verification of updated Java bytecode programs' presented at 9th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECos), Bucharest, Romania, 10–11 September 2015.

1 Introduction

During their life cycle, programs need to be updated in order to change their semantics, perform optimisations or add features. Dynamic software updating (DSU) consists in updating running programs on the fly without any downtime. Systems implementing this feature require high availability. Indeed, in some applications such as banking, air traffic control and health support software, interrupting the application to perform a classical shut down, update and restart leads to considerable losses.

Dynamic software updating raises three major scientific problems: code and data update, update timing and update correctness. The first and the second issues are tackled by DSU systems by defining techniques such as functions indirection for code update (Chen et al., 2007; Duggan, 2005), state transfer functions for data update (Hayden et al., 2011; Gupta and Jalote, 1993) and introspection approaches to determine safe points to perform dynamic updates (Subramanian et al., 2009; Noubissi et al., 2011). These techniques are used in DSU systems in several application areas such as embedded systems (Lv et al., 2012; Noubissi et al., 2011), real time systems (Wahler et al., 2009; Seifzadeh et al., 2009) and operating systems (Arnold and Frans, 2009).

Given the increasing need for DSU and its use in critical systems, the question of its correctness is raised. In fact, a dynamic update may introduce errors which may alter the execution, leading the system to an unexpected state. Besides, in some cases, the update is critical (e.g., in smart card application) in such a way that an attacker can take advantage of an incorrect update. In such applications, that have to be managed from security and safety point of view, the update must pass some certification procedures for example Common Criteria (2015). For a certain certification level, one has to provide a formal proof of the security mechanism implemented. A formal way to specify updates and verify their correctness is then necessary.

In this certification scheme, seven evaluation assurance levels (EALs) are defined. These levels are a measure of assurance quality, where EAL 7 is the strongest. Assessment at the two highest levels, EAL 6 and 7, requires formal methods and gives not only the assurance that the security functions are implemented, but also that these functions are correct with respect to the security policies defined in the security target of the product. In the particular case of DSU, new issues are raised. Indeed, the certification process is a static view of a system and therefore requires specific treatment for certifying dynamic systems. Recently, the French ANSSI (*Agence Nationale de la Sécurité des Systèmes d'Information*) has proposed a dedicated process (ANSSI, 2015) allows to certify a product that can be dynamically changed, certifying only the update code and the loader. It defines the concepts and the methodology applicable to the evaluation of a product embedding a code loading mechanism and the usage of this loader as part of the assurance continuity process.

DSU correctness does not rely on a unique definition. It is instead based on the consideration of several correctness criteria, which can be divided into two categories. The first category regroups common properties that are shared by all updates such as type safety (Hjálmtýsson and Gray, 1998; Neamtu et al., 2006; Makris, 2009; Zhang et al., 2012) and consistency (Baumann et al., 2005; Hjálmtýsson and Gray, 1998). The second category refers to specific properties related to the semantics of updated programs (Hayden et al., 2012; Anderson and Rathke, 2009; Charlton et al., 2011).

In this work, we study the case of a DSU system for Java Card applications. The system presented in Noubissi et al. (2011) called EmbedDSU is a system developed to implement DSU functionalities for Java card applications. It is based on two parts: off-card and on-card. In the off-card part, a module called DIFF generator computes the syntactic changes between the old and the new version of the application and generates a DIFF file (called also a patch). This patch is then sent to the card to perform the update by other modules implemented by extending the Java card virtual machine. These modules represent the on-card part of the system.

The main objective of this paper is to deploy formal methods to guarantee correctness properties of EmbedDSU. We will focus on three properties:

- *Type safety:* this property is meant to ensure that data types for the program's constants, variables and methods comply to the contract defined by the class they belong to. It represents the corner stone of Java-based applications safety. In this work, we guarantee that updated programs do not introduce type errors.
- *Behavioural correctness:* once type safety is established, we consider the second property related to the behaviour of the updated program. We present an approach to establish that the obtained program, once the update is performed, implements the intended specifications which are expressed by the programmer before calculating changes off-card.
- *Activeness safety:* the third property is related to the computation of a safe update point. Searching a safe update point to perform DSU is a critical concern. Indeed, a hazardous application of DSU leads to errors in the application. We present a mechanism that brings the system into a quiescent state to perform the update. The main condition of the quiescent state is the absence of updated methods in the list of active methods of the application. This property is called activeness safety criteria. We propose a formal specification of the safe update point (SUP) detection mechanism and we derive proof obligations that guarantee the property.

The contributions of this paper are:

1. A formal semantics for update operations at the instruction level (adding, deleting and modifying instructions)
2. An extension of the formal semantics to consider more updates operations related to methods and fields (adding, deleting and modifying methods and fields)
3. Establishing the soundness of the semantics: this contributes, with contributions in items 1 and 2, to establish *type safety property*
4. An approach for formal verification of the specification of updated programs. This contribution leads to establish *the behavioural correctness*
5. Specification of the search SUP module and proposition of an approach based on functional specifications to establish the *activeness safety property*.

The contributions 2, 3, formalisation details of 4 and the contribution 5 represent the novelty of this paper with regard to our conference paper (Lounas et al., 2015).

This paper is organised as follows: Section 2 gives an overview of EmbedDSU. Section 3 introduces the language and the formal semantics of updates, as well as the guaranteed type safety property. In Section 4, we present an approach to verify the semantics of updated programs. We present our functional modelling of Java bytecode and propose a predicate calculus for update operations to ensure the verification of behavioural correctness property. In Section 5, we present formal specification of the SUP search mechanism and the verification of the activeness safety property. We discuss related work in Section 6 and conclude in Section 7.

2 Overview of EmbedDSU

EmbedDSU, introduced in Noubissi et al. (2011), is a software-based DSU technique for Java-based smart cards which relies on the Java card virtual machine. The virtual machine interprets Java card programs once they are compiled to bytecode and loaded to the card. The system EmbedDSU is based on the modification of the virtual machine (VM).

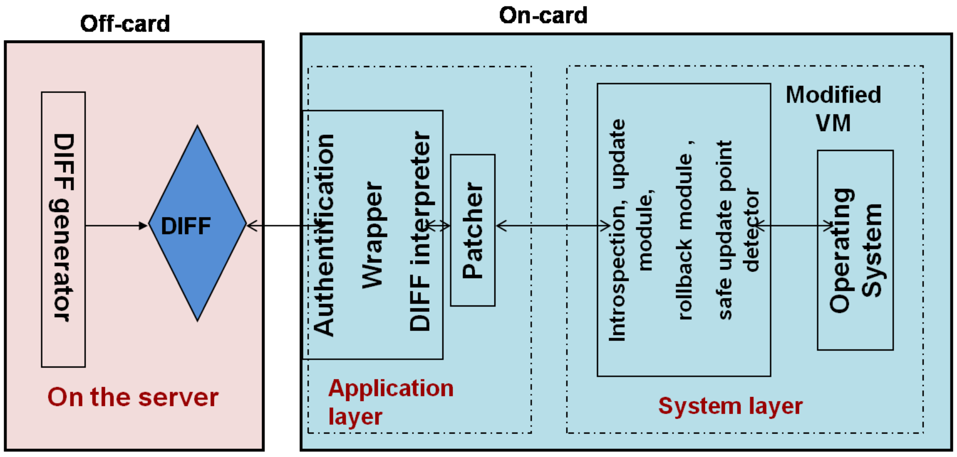
2.1 The system architecture description

The system EmbedDSU is composed of two parts: the off-card part and the on-card part. Its architecture is illustrated by Figure 1:

- In off-card, a module called *DIFF generator* determines the syntactic changes between versions of classes in order to apply the update only to the parts of the application that are really affected by the update. The changes are expressed using a domain specific language (DSL). Then, the DIFF file result is transferred 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* initialises data structures for update. These data structures are read by the *update module* module to determine what to update and howto update, by the *safeUpdatePoint* detector module to determine when to apply the update and by the rollback module to determine how to return to the previous version in case of update failure. These points require the introspection of the virtual machine.
  2. *System layer:* the modified virtual machine supports the followings features:
     a. *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
     b. *update module* which modifies object instances, method bodies, class metadata, references, affected registers in the stack thread and affected VM data structures
     c. *SafeUpdatePoint* detector module allows detecting SUP in which we can apply the update by preserving coherence of the system.

The system EmbedDSU is suitable for smart cards especially in term of resource limitations. It was established that sending a DIFF file is less resource consuming than sending the whole new version to the card and perform updates and that the resources implied by the update modules are acceptable in term of memory occupation (Noubissi, 2011).

Figure 1 Architecture of EmbedDSU (see online version for colours)

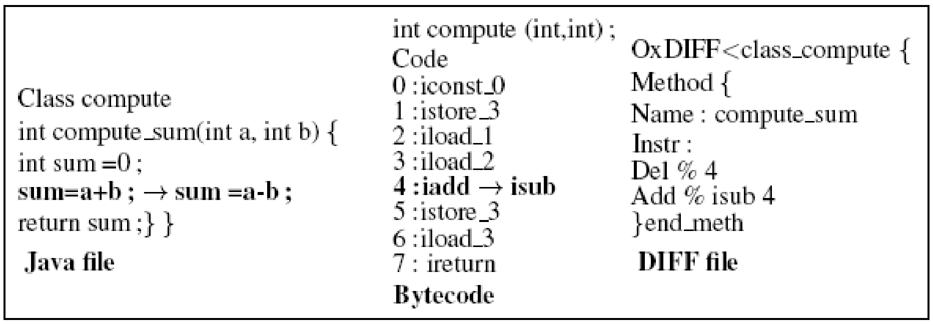


2.2 *The update process*

The system EmbedDSU updates three principal parts:

- *the bytecode:* the process updates first the bytecode of the updated class and the meta data associated with it, e.g., 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.

Figure 2 Example of a patch (DIFF file)



```
Class compute
int compute_sum(int a, int b) {
int sum =0 ;
sum=a+b ; → sum =a-b ;
return sum ;} }
Java file
```

```
int compute (int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
4 :iadd → isub
5 :istore_3
6 :iload_3
7 : ireturn
Bytecode
```

```
OxDIFF<class_compute {
Method {
Name : compute_sum
Instr :
Del % 4
Add % isub 4
}end_meth
DIFF file
```

Update process starts by updating method bodies and class meta data of the class to be updated and related classes. The DIFF file includes information on entries of each modified method, parameters, local variables and bytecode instructions. Then, to update the code of a class, the update module proceeds by copy while modifying class meta data like constant pool, field table, method table and constant table. For each method that is not deleted, the process copies while modifying method header and bytecode instructions so that the old version is transferred to a new space while modifying it to obtain the corresponding new version of the class. After updating the class, update process continues with the update of all other constant pools of related classes to modify references to old methods or fields in order to point to the new field entry table, or to point to the new method.

This paper addresses first the bytecode update at the method level. In bytecode language, each instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands. The types of updates that may occur are: adding, modifying or suppressing bytecode instructions, methods, local variables and fields. These updates are contained in the DIFF file which indicates the update and where it occurs. For example, Figure 2 shows an updated Java program (at the left side) and the corresponding modification at bytecode level, the patch (at the right side) indicates that the instruction *iadd* in the method *compute_sum* is deleted and the instruction *isub* is added at the same place provided by the program counter. We studied the update of instances in the heap and updating references in the stack in a previous work (Lounas et al., 2014). The SUP detection mechanism is detailed in Section 5. The next section presents the bytecode language, its semantics and a formal semantics of update operations performed by EmbedDSU.

3 Language and semantics for type safety

In this section, we present our formalised language. It provides a representative subset of bytecode instructions related to stack manipulation, object creation, arithmetic, fields, methods invocation and jump instructions. We also give a formal semantics for update operations. In the present paper, we extend the formal semantics given in our conference paper (Lounas et al., 2015) which deals only with update operations at instruction level (adding, deleting and modifying instructions). Here, we add the semantical rules for update operations related to methods and fields (adding, deleting and modifying methods and fields). This will help us to characterise well-formed updates.

3.1 The language

In Freund and Mitchell (1999), the authors present a formalisation of the semantics of a small Java bytecode based on a type system. Our proposition is mainly built by adding update instructions (*Upd_Instr*) to manage addition, deletion and modification of instructions within a method code. We introduce the following notations in our language definition: *x*, *a*, *L*, *A*, *f*, *l*, *t* and *pc* to denote respectively: a local variable, a constant, an instruction address, a class name, a field name, a method name, field type and method signature and the program counter.

$$\begin{aligned}
Instruction :=&\mid pop \mid if\ L \mid store\ x \mid load\ x \mid new\ A \\
&\mid binop \mid neg \mid const\ a \mid invokevirtual\ A\ l\ t \mid goto\ L \\
&\mid getfield\ A\ f\ t \mid putfield\ A\ f\ t \mid return
\end{aligned}$$

$$\begin{aligned}
Upd\_Instr ::= &\ Add\_Inst\ Instruction\ pc \\
&\mid Dlt\_Inst\ Instruction\ pc \\
&\mid Mod\_Inst\ Instruction\ instruction\ pc
\end{aligned}$$

The instruction *pop* extracts the top of the stack and *const a* pushes a constant *a* on the top of the stack. The instruction *load x* pushes the value in the variable *x* on the top of the stack whereas the instruction *store x* pops the top of the stack and stores it in the variable *x*. The instruction *if L* jumps to *L* if the top of the stack is not zero else it performs the next instruction. *Goto L* jumps to *L*. The instruction *New A* allocates a new object of type *A* and pushes it on the top of the stack. The instructions manipulating fields are: *getfield A f t* and *putfield A f t*. *Getfield* reads the field *f*, which has the type *t* of the object of class *A* whose reference is on the top of the stack and pushes its value on the top of the stack and putfield modifies the field *f* with the value popped form the stack. The instruction *invokevirtual* invokes the method *l* of signature *t* and the class *A*. The instruction *binop* is used to gather arithmetic binary operations: *add*, *mult* and *sub*. The instruction *neg* negates the top of the stack and return is for method *return*. We notice that a modification on an instruction is interpreted as a deletion followed by an addition.

Update instructions are respectively: adding an instruction, deleting instruction and modifying an instruction. We indicate the place of the update operation with *pc*. We will introduce further in this section, the update operations related to inserting, deleting and modifying methods and fields.

3.2 Operational semantics for bytecode instructions

The operational semantics is defined by a transition relation over configurations. In our model, based on the standard framework for operational semantics [see Freund and Mitchell (1999) and Bannwart and Müller (2005)], a configuration represents a step execution and is denoted by a tuple <*M*, *s*, *h*, *f*, *pc*> where: *s* consists of an operand stack, *h* represents the heap containing created instances, *f* a local variables map associating values to local variables, *pc* is a program counter and *M* the method body. A transition <*M*, *s*, *h*, *f*, *pc*> → <*M*, *s*2, *h*2, *f*2, *pc*2> takes the state from the configuration <*M*, *s*, *h*, *f*, *pc*>to the configuration <*M*, *s*2, *h*2, *f*2, *pc*2>. The rules for the instructions of our language are given in Figure 3.

- The rule (*Rpop*) indicates that the instruction *pop* extracts the top of the stack to obtain another configuration. The rule (*Rnew*) indicates the creation of a new object of class A, thereby the modification of the current heap. A reference to the new object is pushed onto the stack.
- The rule (*Rldx*) puts the value of *x* on the top of the stack and the rule (*Rstx*) pops a value from the stack and assigns it to a variable, *f* is modified accordingly.
- The instruction *if L* has two rules (*Rif*1 and *Rif*2). It either jumps to the indicated line or performs the following instruction according to the value of the top of stack.
- The rule (*Rcst*) indicates that a constant is pushed on the stack and the rule (*Rneg*) indicates that the top of the stack is replaced by its opposite.
- The rule (*Rget*) indicates that the value of the field *f* is obtained and pushed on the stack, whereas the rule (*Rput*) updates the heap with the new value of the field of the object which is on the top of the stack. The new value is popped from the second element of the stack.

- The unconditional jump *Goto l* is expressed by the rule (*Rgto*). The rule (*Rop*) indicates that a binary arithmetic operation pops two values from the stack, performs the binary operation and pushes the result.
- The rule (*Rinv*) expresses invoke of the method *l* on an object reference. The reference and parameters are popped from the stack and are replaced by the return value *v* of the invoked method after its execution.

The operational semantics gives a clear presentation of bytecode instructions and is a step to introduce the semantics of insertion and suppression of instructions which is discussed in the next section.

3.3 *Formal semantics for update instructions*

Lounas et al. (2015) presented a static semantics that expresses the effects of the update instructions in a configuration of the bytecode. This semantics is designed to describe conditions and results of update instructions. The particularity of this semantics is the expression of typing information related to local variables and operand stack of a method. This information is tracked step by step and thus prevents type errors in the updated code.

Figure 3 Rules for operational semantics

$$\frac{M[pc]=pop}{<M,v.s,h,f,pc>\rightarrow <M,s,h,f,pc+1>}\ (Rpop) \qquad \frac{M[pc]=new\ A,h'=h[create(A,ref)]}{<M,s,h,f,pc>\rightarrow <M,ref.s,h',f,pc+1>}\ (Rnew)$$

$$\frac{M[pc]=load\ x}{<M,s,h,f,pc>\rightarrow <M,f[x].s,h,f,pc+1>}\ (Rldx) \qquad \frac{M[pc]=store\ x}{<M,v.s,h,f,pc>\rightarrow <M,s,h,f[x\leftarrow v],pc+1>}\ (Rstx)$$

$$\frac{M[pc]=if\ l,v\neq 0}{<M,v.s,h,f,pc>\rightarrow <M,s,h,f,l>}\ (Rif1) \qquad \frac{M[pc]=if\ l}{<M,0.s,h,f,pc>\rightarrow <M,s,h,f,pc+1>}\ (Rif2)$$

$$\frac{M[pc]=const\ a}{<M,s,h,f,pc>\rightarrow <M,a.s,h,f,pc+1>}\ (Rcst) \qquad \frac{M[pc]=neg}{<M,v.s,h,f,pc>\rightarrow <M,(-v).s,h,f,pc+1>}\ (Rneg)$$

$$\frac{M[pc]=getfield\ \ a\ f\ t,v=h[o.f]}{<M,o.s,h,f1,pc>\rightarrow <M,v.s,h,f1,pc+1>}\ (Rget) \qquad \frac{M[pc]=putfield\ A\ f\ t,h'=h[o.f\leftarrow v]}{<M,o.v.s,h,f1,pc>\rightarrow <M,s,h',f1,pc+1>}\ (Rput)$$

$$\frac{M[pc]=goto\ l}{<M,s,h,f,pc>\rightarrow <M,s,h,f,l>}\ (Rgto) \qquad \frac{M[pc]=binop,op\in\{+,-,*\}}{<M,v1.v2.s,h,f,pc>\rightarrow <M,(v1\ op\ v2).s,h,f,pc+1>}\ (Rop)$$

$$\frac{M[pc]=invokevirtual\ A\ l\ t\ ,<l,\varepsilon,h,f_l,0>\rightarrow<l,v,h1,f_l',pc_l>}{<M,a_1...a_n.s,h,f,pc>\rightarrow <M,v.s,h1,f1,pc+1>}\ (Rinv)$$

3.3.1 *Concepts and notations*

To express our semantics for update operations, we need to introduce some concepts and notations:

- *Typing information:* in this semantics, we introduce two elements in order to track typing information: *F* and *S*. *F* is a mapping from a program point (representing an instruction address) 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 stacks 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. The empty sequence of types is denoted by (ε). The symbol $\top$ is used to represent a default initial value in typing variables.

- *Configuration information:* we consider configuration at line *i* as a tuple <(*F*, *S*, *SD*, *M*), *i*> where *F* and *S* represent typing information, *SD* represents the stack depth, *M* is a mapping that associates a number to each line of the code. It is obtained by a function noted *Map* on a bytecode *BC*.

3.3.2 Semantics rules

The goal of expressing semantics of update operation with typing information is to establish that the update leads to well typed programs. We are, now, able to define the judgement that expresses that a bytecode *BC* is well typed by *F* and *S* is:

$$\frac{\begin{array}{l} F_1 = F_\top,\ SD1 = 0 \\ S_1 = \varepsilon,\ M1 = Map(BC) \\ \forall_i \in DOM(BC),\ F, S, i \vdash BC \end{array}}{F, S \vdash BC}$$

Figure 4 Some rules for update operations for instructions

$$\frac{\begin{array}{l} Add\_inst\ \ new\ A(i+1) \\ SD_{i+1} = SD_i + 1 \\ S_{i+1} = A.S_i \\ F_{i+1} = F_i \\ M2 = Add\_inst(M1, new\ A, i+1) \\ PC\_MAX++ \\ i+1 \in DOM(BC) \end{array}}{<F_i, S_i, SD_i, M1, i> \to <F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1>}\ (Rup1)$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ new\ A\ \ (i+1) \\ SD_i = a \to \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, new\ A, i+1) \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC) \\ PC\_MAX-- \end{array}}{<F_i, S_i, SD_i, M1, i> \to <F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1>}\ (Rup2)$$

$$\frac{\begin{array}{l} Add\_inst\ add(i+1) \\ SD_{i+1} = SD_i - 1 \\ S_i = int.int.S_0 \Rightarrow S_{i+1} = int.S_0 \\ F_{i+1} = F_i \\ M2 = Add\_inst(M1, add, i+1) \\ PC\_MAX++ \\ i+1 \in DOM(BC) \end{array}}{<F_i, S_i, SD_i, M1, i> \to <F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1>}\ (Rup3)$$

$$\frac{\begin{array}{l} Dlt\_inst\ \ (add\ \ (i+1)) \\ M2 = Dlt\_inst(M1, add, i+1) \\ SD_i = a \to \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ S_i = int.int.S_0 \to \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC) \\ PC\_MAX-- \end{array}}{<F_i, S_i, SD_i, M1, i> \to <F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1>}\ (Rup4)$$

The first two lines of the judgement represent the initial configuration: all variables are mapped to the value $\top$, stack depth is zero, the sequence of types is initially empty and *M*1 is the initial mapping of the bytecode. In the last line, *DOM*(*BC*) is the set of addresses used by the method. The expression *F*, *S*, *i* $\vdash$ *BC* expresses that *BC* is well typed until the step *i* in the evaluation and the entire line expresses that the program *BC* is well typed at each step *i* in the evaluation. This premiss is derived from the semantics of update operations. The conclusion of the judgement expresses that *BC* is well typed.

Figure 4 shows four rules of update operation semantics. Transitions through configurations represent the evolution of typing information step by step (notations F_i and S_i) and ensure that the program is well typed at each step. In these rules, the information *PC_MAX* is used to express the maximum offset in a method bytecode. For illustration, the insertion of the instruction *new A* at line $i + 1$, represented by the rule (*Rup*1), allows us to obtain a new configuration if the stack depth is incremented, local variables are not affected and the type *A* is inserted at the top of the stack. For the insertion of an instruction representing an arithmetic binary operation *Binop*, we show the rule (*Rup*3) of the instruction add which is a special case of *Binop*: this operation pops two elements (integers) from the stack and then pushes the result. *mult* and *sub* have analogous explanations by writing the right operation. In these rules, the mapping *M*2 is the result of operations on *M*1. The operations which represent manipulations on bytecode are: *range* and *shift*. The operation *range* extracts from a mapping *M*1 a part *M*2 included between line *n* and line *m*. The second operation shifts a part from a mapping between *n* and *m* for *p* positions which is determined by the number of added instructions.

We define the operations *look_for_jumps* and *update_jumps* to take into account jumps in bytecode transformation: *look_for_jumps* returns from a mapping a list of jumps instructions represented by their line number and the operation update_jumps updates jump instructions:

$$Look\_for\_jumps : mapping \rightarrow int\ list$$

$$Update\_jumps : mapping * \mathrm{int}\ list * \mathrm{int} \rightarrow mapping$$

Table 1 Example for typing information track

i	*Instruction*	F_i	S_i
0	const 0	(1, *int*), (2, *int*), (3, *int*)	*int.ε*
1	store 3	(1, *int*), (2, *int*), (3, *int*)	*ε*
2	load 1	(1, *int*), (2, *int*), (3, *int*)	*int. ε*
3	load 2	(1, *int*), (2, *int*), (3, *int*)	*int.int. ε*
upd	Del_inst add (4)	(1, *int*), (2, *int*), (3, *int*)	*int.int ε*
upd	Add_inst sub (4)	(1, *int*), (2, *int*), (3, *int*)	*int.int ε*
4	sub	(1, *int*), (2, *int*), (3, *int*)	*int. ε*
5	store 3	(1, *int*), (2, *int*), (3, *int*)	*ε*
6	load 3	(1, *int*), (2, *int*), (3, *int*)	*int. ε*
7	return	(1, *int*), (2, *int*), (3, *int*)	*ε*

These operations update jumps within the bytecode if necessary. When we add for instance an instruction at *pc*, the instructions after this position are shifted and their numbers change. It is then necessary to update *goto* and *if* instructions accordingly. These modifications keep the structure of the bytecode coherent. In the rules for instructions suppression (*Rup*2 and *Rup*4), the notations *Effect_STK*, *Effect_F* and *Effects_SD* are used to express the effects of an instruction on the stack and the local variables and stack depth. They are used to readjust these elements to the instruction at (i + 1) in the new bytecode after the suppression. The notation (*M*2)*F* (respectively, (*M*2)*S*) is used to express *F* (respectively, *S*) in the mapping *M*2. We notice that in this formalisation, a modification is considered as suppression followed by an insertion. The remaining rules for inserting and suppressing instruction are given in the appendix. To illustrate how typing information evolves according to the semantics of (update) instructions, we reconsider the same program shown in Section 2 to illustrate DIFF files. The function computing the sum of two integers is rewritten in our bytecode sub language. The update consists in deleting *add* instruction and inserting *sub*. The evolution of typing information is shown in Table 1. The first column represents the number of the instruction. Update instructions (colored rows), are indicated by *upd*. The third column represents the evolution of typing information related to local variables. Local variables are represented by integers (1, 2 and 3). A type is associated to every variable. A variable and its type are represented as a couple (*var*, *type*). The evolution of the types in the operand stacks are shown on the fourth column. At the level of update instructions, the typing information guarantees that local variables and operand stack conforms the requirements of such instructions defined by the semantics.

3.4 Formal semantics for methods and fields

In this subsection, we propose a new extension of the formal semantics to handle methods and fields in a class. In this formalisation, a class C is defined by:

- its name
- a method table (noted Mt): which is a structure containing an entry for every method m in the class
- a field table (noted Ft): which is a structure containing entry for every field *f* in a the class
- its meta-data Meta including a constant table, status and an offset Table. 3.

3.4.1 Methods

This subsection presents the semantics to handle modifications related to methods. A method is defined with: its name m, its code BC, its signature sig and a structure to represent method information (*Im*) including: the maximum of local variables and the max_stack. Three cases are considered:

- *Add_method:* this operation leads to the introduction of a new method in a class C.
- *Dlt_method:* this operation leads to the suppression of a method from a class C.
- *Mod_method:* this operation leads to an override of an existing method in order to implement a new version in a class C.

The formalisation requires the definition of extension to the functions and mappings about type information. We introduce the sets *M*, *Fs*, *Ss* representing respectively:

- the set of method names
- the set of mappings *F* from a program point to a mapping from variables to types
- the set of mappings *S* from a program point to an ordered sequence of types. The string S_i gives the types of entries in the operand stacks at program point *i*.

Mappings *F* and *S* keep the same definition introduced in update operation semantics for instructions. We define two functions F^c and S^c:

- the function F^c: $M \rightarrow Fs$ is used to associate each method m in the class C to a mapping *F* representing its variables typing information
- the function S^c: $M \rightarrow Ss$ is used to associate each method m in the class C to a mapping *S* representing its operand stack typing information.

These functions and information are used to formalise the semantics of three cases as shown on Figure 5. Every case is represented by a rule that captures the condition that ensures type safety of update operations.

The rule (*Rm*1) expresses the introduction of a new method. In this rule, the first line represents the operation. The second line represents a check that the introduced method already exists in the method table. This check ensures that the operation does not inadvertently replace an existing method with the same name. The function *look_for_entry* returns a boolean true if a method with the same name and signature exists, false otherwise. In this case, an entry for the introduced method is created in the method table with the function *create_entry*. The creation of the entry leads to create and initialise typing information related to the inserted method (lines 4, 5 and 6). The function *upd_meta* is used to update meta data of the class with information related to the method (offsets table).

The rule (*Rm*2) formalises the operation of deleting a method. The check at the second line ensures that an entry for the methods exists in the method table. This operation leads to the suppression of this entry. Typing information about the deleted method is deleted as well. Offsets are adjusted with *upd_meta*.

Figure 5 Rules for introducing, deleting and modifying methods

$$\frac{\begin{array}{l} Add\_method(m, lm, BC, sig) \\ look\_for\_entry(Mt, m, sig) = false \\ create\_entry(Mt, m, sig) \\ S_s \leftarrow S_s \cup S^c(m) \\ S^c(m) \leftarrow \varepsilon \\ F_s \leftarrow F_s \cup F^c(m) \\ init(F^c(m), Im.loc) \\ upd\_meta(Meta, m) \end{array}}{F^c, S^c \vdash C} \; (Rm1) \qquad \frac{\begin{array}{l} Mod\_method(m, lm, BC, sig) \\ look\_for\_entry(Mt, m, sig) = true \\ Del\_entry(Mt, m, sig) \\ S_s \leftarrow S_s[S^c(m) \leftarrow \varepsilon] \\ var\_upd(m, Im.loc) \\ init(F^c(m), Im.loc) \\ upd\_meta(Meta, m) \\ S^c(m), F^c(m) \vdash BC \end{array}}{F^c, S^c \vdash C} \; (Rm3)$$

$$\frac{\begin{array}{l} Del\_method(m, lm, BC, sig) \\ look\_for\_entry(Mt, m, sig) = true \\ Del\_entry(Mt, m, sig) \\ S_s \leftarrow S_s \setminus S^c(m) \\ F_s \leftarrow F_s \setminus F^c(m) \\ upd\_meta(Meta, m) \end{array}}{F^c, S^c \vdash C} \; (Rm2)$$

The rule (*Rm*3) expresses the modification of an existing method. The check at line 2 of the rule ensures that this method has already an entry in the method table (we notice that an update in a signature is considered as deleting a method and introducing a new method). The typing information $S^c(m)$ is re-initialised in order to perform the update from the method beginning. The typing information $F^c(m)$ is re-initialised after performing potential updates in locals (*var_upd*(*m*, *Im.loc*)). Typing information must validate the correctness of all the instructions in the new bytecode BC of the method (the last premiss). This condition is based on the check performed at instruction level and defined in the previous subsection. The function (*var_upd*(*m*, *Im.loc*)) is used to manage updates in local variables of a method. The case of adding, deleting or modifying a local variable is a particular part of method modification. An introduced variable is defined with its name and type. Rules for variables are given in the appendix.

We notice that in the rules for methods (and fields in the next subsection), the conclusions are not expressed as evolution of configurations. They express directly the fact of obtaining a well typed program. This is due to the fact that configurations are used to express evaluation within the code of a method (at instruction level) and that operations for adding, deleting or modifying methods (or fields) are expressed at the level of a class. The second reason is that the impact of such operations on instructions is detected and held at update operations related to instructions.

3.4.2 Fields

We present in this section an extension of the semantics to handle modifications related to fields. In this formalisation, a field is represented by its name *f* and its type *tf*. An entry in the field table is created for every field in the class. The typing information is supplied by a mapping to keep track of typing information for fields. This is represented by the mapping *Fl* which associates a type for each field at every point in the program. Figure 6 represents the rules to check adding, deleting and modifying a field in a class.

Figure 6 Rules for introducing, deleting and modifying fields

$$\frac{\begin{array}{l} Add\_field(f, tf, val) \\ look\_for\_entry(Ft, f, tf) = false \\ create\_entry(Ft, f, tf) \\ Fl \leftarrow Fl \cup \{(f, tf)\} \end{array}}{F^c, S^c, Fl \vdash C}\ (Rf1) \qquad \frac{\begin{array}{l} Del\_field(f, tf) \\ look\_for\_entry(Ft, f, tf) = true \\ Del\_entry(Ft, f, tf) \\ Fl \leftarrow Fl \setminus \{(f, tf)\} \end{array}}{F^c, S^c, Fl \vdash C}\ (Rf2)$$

$$\frac{\begin{array}{l} Mod\_field(f, tf, val) \\ look\_for\_entry(Ft, f, tf) = true \\ Set\_val(Ft, f, tf, val) \end{array}}{F^c, S^c, Fl \vdash C}\ (Rf3)$$

The rule (*Rf*1) expresses the introduction of a new field. In this rule, similarly to the case of methods rule, the first line represents the operation. The second line represents a check if the introduced field already exists in the field table. The function *look_for_entry* returns a boolean true if a field with the same name and type exists, false otherwise. In this case, an entry for the introduced field is created in the field table with the function *create_entry*. The information about the type is recorded in *Fl*.

The rule (*Rf*2) formalises the operation of deleting a field. The check at the second line ensures that an entry for the field exists in the field table. This operation leads to the suppression of this entry and the typing information related to it.

The rule (*Rf*3) expresses the modification of an existing field. The check at line 2 of the rule ensures that this method has already an entry in the method table (the same name and the same type). The modification concerns only the modification of the value of a field with *Set_val*. A modification of the type is considered as a suppression of the old field followed by the insertion of the new field. Thus, in this case, there is no operation on *Fl*.

3.5 Soundness

In this subsection, we outline the soundness of the update. The global soundness theorem states that a well-typed program and a well formed update operations (ensured by the semantics) leads to a well typed updated program. We present first a lemma to express that the program is well typed initially, then a single step update soundness to express that an update operation from a well typed program leads to another well typed program.

*Definition 1: (Initial soundness).*We consider a bytecode *BC*, its initial mapping *M*, its typing information *F* and *S* and a DIFF file Δ containing update instructions. At the initial configuration $<F_\top, \varepsilon, 0, M, 0>$, where variable types are initialised at their default value, typing stack information is empty, stack depth is zero and the number of the evaluation step is zero, the program is well typed, we write: $F, S, 0 \vdash BC$.

Lemma 1: (Single-step update soundness). Given a bytecode *BC*, typing information *F* and *S*, a mapping *M* and a *DIFF* file Δ containing update instructions, we have:

$$\begin{aligned}
&\qquad \forall SD, SD' \in N, i, i' \in DOM(BC), j \in N.\\
&F, S, i \vdash BC\\
&\wedge < F_i, S_i, SD, M, i > \overset{\Delta(j)}{\rightarrow} < F_{i'}, S_{i'}, SD', M', i' >\\
&\wedge j \leq length(\Delta)\\
&\Rightarrow F, S, i' \vdash BC
\end{aligned}$$

This lemma expresses that the evaluation of an update instruction from a step *i* such that the bytecode is well typed until *i* leads to a well typed bytecode until the next evaluated step. The transition $\overset{\Delta(j)}{\rightarrow}$ represents the application of the update instruction at line *j* which has to be a valid line in the DIFF file Δ. The expression *length*(Δ) represents the number of update instructions in the DIFF file Δ represented as a list of update instructions. *M*′ represents the resulting mapping.

Proof sketch. The proof of this lemma is based on the different cases obtained by examining all the update instructions represented by their semantics. We sketch the proof by studying the cases of inserting *new* instruction and *add*:

Case 1 Δ(*j*) represents the update instruction *add_inst new A* (*i* + 1). We assume that the hypotheses are satisfied. This update instruction leads, by the semantics, to a configuration where: $F_{i'} = F_i$, $S_{i'} = A.S_i$, $SD' = SD + 1$, $i' = i + 1$ and the mapping *M*′ is obtained by inserting in *M* the new instruction by applying *shift*, *range* and *Update_jumps* operations.

Case 2 Δ(*j*) represents the update instruction *add_inst add i* + 1. We assume that the hypotheses are satisfied. This update instruction leads, by the semantics, to a configuration where: $F_{i'} = F_i$, $S_{i'} = int.S_0$ such as $S_i = int.int.S_0$, $SD' = SD - 1$,

$i' = i + 1$ and the mapping M' is obtained by inserting the new instruction in M by applying *shift*, *range* and *Update_jumps* operations. □

The soundness of a single step in the update leads to express multiple step and method update soundness.

Lemma 2: (Multiple-step update soundness). Given a bytecode BC, typing information F and S, a mapping M and a DIFF file Δ containing update instructions, we have:

$$\begin{aligned}
&\forall SD, SD' \in N, i, i' \in DOM(BC), j \in N. \\
&F, S, i \vdash BC \\
&\wedge < F_i, S_i, SD, M, i > \overset{\Delta(j*)}{\rightarrow} < F_{i'}, S_{i'}, SD', M', i' > \\
&\wedge j \leq length(\Delta) \\
&\Rightarrow F, S, i' \vdash BC
\end{aligned}$$

This lemma expresses that a sequence of sound single update steps starting from a well typed bytecode step leads to a well typed bytecode at the resulting step. The symbol $\overset{\Delta(j*)}{\rightarrow}$ represents the application of update instructions contained in Δ ranging from the first update instruction to the j^{th} update instruction. This can be represented by a sequence of transitions starting from the configuration corresponding to i and finishing at the configuration corresponding to $i' : C_i \overset{\Delta(1)}{\rightarrow} C_{i+1} \overset{\Delta(2)}{\rightarrow} C_{i+2} \ldots \overset{\Delta(j)}{\rightarrow} C_{i'}.M'$ represents the resulting mapping.

Proof sketch. The proof of this lemma is by induction on the length of $\Delta(j*)$. We assume that the hypotheses are satisfied:

Basis case	$length(\Delta(j*)) = 0$. The configuration remains unchanged and thus, satisfied by hypothesis.
Induction case	we suppose that the property is satisfied for $length(\Delta(j*)) = n$. Thus, for $length(\Delta(j*)) = n + 1$, the demonstration follows the same pattern as lemma 1, on the basis of examining the cases of the update instruction number $(n + 1)$. □

This lemma introduces the update at the level of an entire method.

Theorem 1: (Method update soundness). Given a method m, its bytecode BC, an initial mapping M, typing information F and S and a DIFF file Δ_m containing update instructions related to the method. The update of a method m according to Δ_m starting from an initial configuration C_0 such as $C_0 = <F_\top, \varepsilon, 0, M, 0>$ leads to a well typed program.

$$\begin{aligned}
&\forall SD' \in N\ i' \in DOM(BC). \\
&C_0 \overset{\Delta_m}{\rightarrow} < F_{i'}, S_{i'}, SD', M', i' > \\
&\wedge \exists j, j \geq i' \wedge j \in DOM(BC) \wedge BC(j) = return \\
&\Rightarrow F, S \vdash BC
\end{aligned}$$

In this theorem, the soundness of a method update with update instructions contained if a DIFF file Δ_m starting from an initial configuration leads to a well typed method code. The step *i′* represents the configuration obtained after the application of all the DIFF file Δm. The transition $\overset{\Delta_m}{\rightarrow}$ represents the application of update instructions contained in Δ_m which can be represented by a sequence of transitions $C_0 \overset{\Delta_m(0)}{\rightarrow} C_1 \overset{\Delta_m(1)}{\rightarrow} C_2 \ldots \overset{\Delta_m(j)}{\rightarrow} C_{i'}$ where $\Delta_m(j)$ represents the last update instruction in the DIFF file. The rest of the evolution until the end of the method (reaching the *return* instruction) is evaluated with the standard bytecode instruction of the language (Freund and Mitchell, 2003). The demonstration of this theorem is by induction of the length of the DIFF file related to the method. Thus, the proof follows the same scheme as for lemma 2.

We express now the general soundness theorem. This theorem guarantees the soundness of all types of updates operations contained in the DIFF file (instructions, methods and fields).

Theorem 2: (General soundness). Given a class *C*, its set of methods *Meth_c*, typing information sets for methods F_s and S_s, a DIFF file Δ and typing information F^c, S^c and *Fl* for the class, the update of *C* with regard to Δ leads to a well typed program if and only if:

$$\begin{aligned}
&\forall m \in Meth\_c, \forall i \in 1 \ldots length(\Delta),\\
&\Delta(i) = Add\_method(m, \mathrm{Im}, BC, sig) \Rightarrow (S_m = \varepsilon) \wedge S_m, init(F_m), 0 \vdash BC \Rightarrow\\
&F^c, S^c, Fl \vdash C \wedge\\
&\Delta(i) = Del\_method(m, \mathrm{Im}, BC, sig) \Rightarrow (F_s \leftarrow F_s \setminus \{F_m\} \wedge S_s \leftarrow S_s \setminus \{S_m\}) \Rightarrow\\
&F^c, S^c, Fl \vdash C \wedge\\
&\Delta(i) = Mod\_method(m, \mathrm{Im}, BC, sig) \Rightarrow F_m, S_m \vdash BC \Rightarrow F^c, S^c, Fl \vdash C \wedge\\
&\Delta(i) = Add\_field(f, tf, val) \Rightarrow F^c, S^c, Fl \setminus \{(f, tf)\} \vdash C \wedge\\
&\Delta(i) = Del\_field(f, tf) \Rightarrow F^c, S^c, Fl \setminus \{(f, tf)\} \vdash C \wedge\\
&\Delta(i) = Mod\_field(f, tf, val) \Rightarrow F^c, S^c, Fl \vdash C
\end{aligned}$$

In this theorem, all types of update performed in a class are considered. The updates are contained in the DIFF file. The variable *i* is used to indicate the current update instruction.

The number of update instructions in the DIFF file are represented by *length*(Δ). When dealing with method modification, we consider that the updates related to the method are specified in a Δm which is a part of the general DIFF file Δ. In this theorem, F_m and S_m express typing information for a method *m*. It is initialised to default value and empty when inserting a new method. When modifying a method, its bytecode has to be well typed with regard to F_m and S_m (results from Theorem 1). The update operations for fields result straightforwardly from semantics rules. General soundness theorem ensures that the updated program will not attempt to perform any illegal operations and that all operations are sound with respect to the system type safety. The entire demonstration of these lemmas and theorems is planned to appear in a technical report. It implies that our model is correct by showing that no erroneous updated programs are accepted.

3.6 Application

Given the fact that type safety represents the corner stone of the Java card platform safety and security, developing a verification tool to ensure this property for updated programs is a crucial issue. In the standard life cycle of a smart card application, a program must pass through bytecode verification before it is uploaded on-card. Bytecode verification is also performed on-card by type checking.

The EmbedDSU system introduces a step in the life cycle of a smart card application. The virtual machine is extended to perform update operations and obtain a new version of the program. The process leads to a bytecode program which did not pass through the standard verification. Our semantics represent the first specification which captures how Java card bytecode must be checked for update. Based on this specification, our work extends the code update module with on-card verification. We developed a prototype for the verifier using our semantics. The formalisation is embedded in a system with high order logic (Coq proof assistant) in order to guarantee type safety. This property ensured, we have to tackle the behavioural correctness property. Indeed, once an updated program is ensured to be type safe, one has to guarantee that the intended specification is preserved by the application of the update operations contained in the DIFF file. This property is detailed in the next section.

4 Approach for formal verification of behavioural correctness

The mechanism of EmbedDSU implies the modification of the bytecode of a running application on-card after the conventional verification during its life cycle process. In this process, bytecode passes verification process based especially on type verification. The applications of update operations on-card are performed with insertion and suppression of instructions according to the DIFF file. Consequently, after the update process on-card, a new bytecode that was not submitted to the conventional verification process is obtained. Our work allows to:

- Ensure the validity of update operations of the DIFF file according to the formal specification of the Java Card virtual machine specification and that the updates lead to a type safe program. This property was discussed in the precedent section.
- Guarantee that the application of the update leads to a bytecode with the specification that conforms to the intended specification (provided by the programmer).
- Guarantee that the update is applied at a safe point. This will be discussed in Section 5.

In the second point, we aim to establish that given an initial program $P1$, its new version $P2$ and a DIFF file Δ containing the specification of the transformation derived from the differences between $P1$ and $P2$, the application of the DIFF file on-card on $P1$ (noted *App_PATCH*) leads to $P2'$. The two programs $P2$ and $P2'$ are verified to be semantically equivalent. This equivalence ensures that the system indeed implemented the desired transformation. This problem can be expressed equationally by:

$$\forall P1, P2, P2', \Delta = DIFF(P1, P2') = App\_PATCH(P1, \Delta) \Rightarrow P2 \equiv P2'$$

This raises two major issues:

1 the modelling of the application of the DIFF file on an existing program

2 the expression of the equivalence which guarantees the correctness of the update (noted ≡ in the equation).

Figure 7 Approach for verification of behavioural correctness (see online version for colours)

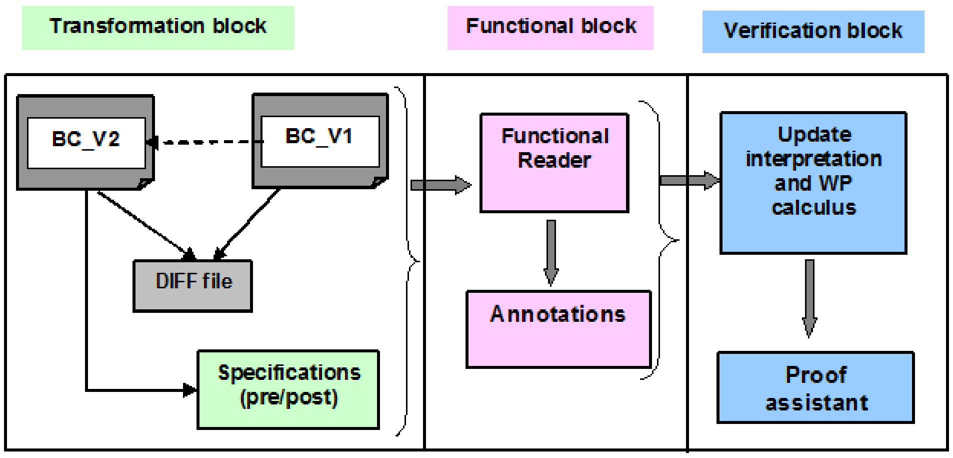
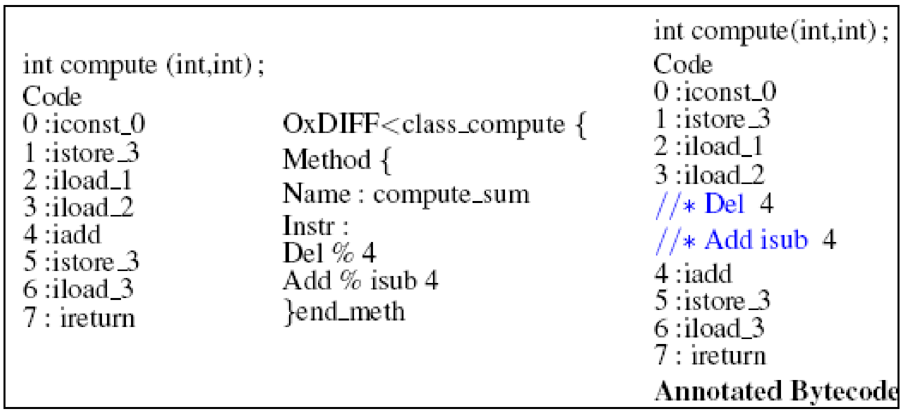


Figure 7 gives an overview of an off-card approach for formal verification of behavioural correctness of updated programs. The approach is split in three parts:

- *The transformation block:* at this stage, we obtain from a first version of a bytecode program BC_V1 and a second version BC_V2 (version one transformed), a DIFF file. The versions BC_V1 and BC_V2 are supposed to be correct. The DIFF file will be applied to the on-card first version. We obtain a new version on-card. The goal of our approach is to establish that the on-card new version and BC_V2 are semantically equivalent. At this level, the specifications of both BC_V1 and BC_V2 are provided by the programmer using existing specification languages.
- *The functional block:* a functional model is defined to represent and manipulate the Java Card bytecode. We implement an automatic translator called *functional_reader* which takes a program written in bytecode and produces a functional representation of it. The application of the DIFF file is represented at this level as annotations of the functional representation with expressions indicating the place of the update operation and its nature (addition of instructions, deletion …)
- *The verification block:* the goal is to verify that the bytecode obtained by transformation is equivalent to the one written by the programmer, i.e., it satisfies the same specification.

The specification of the obtained bytecode in its functional representation with annotations is performed by a weakest precondition calculus that we define specially to deal with update operations. Then, a verification condition generator gives statements to be verified to establish that the obtained specification matches the specification given by the programmer at the transformation block. A proof assistant is used to discharge verification conditions.

Figure 8 Bytecode annotation with update instructions (see online version for colours)



```
int compute (int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
4 :iadd
5 :istore_3
6 :iload_3
7 : ireturn
```

```
OxDIFF<class_compute {
Method {
Name : compute_sum
Instr :
Del % 4
Add % isub 4
}end_meth
```

```
int compute(int,int) ;
Code
0 :iconst_0
1 :istore_3
2 :iload_1
3 :iload_2
//* Del  4
//* Add isub  4
4 :iadd
5 :istore_3
6 :iload_3
7 : ireturn
Annotated Bytecode
```

4.1 Annotation and functional representation of bytecode

The DIFF file containing the update instructions is calculated at bytecode level and then is sent to perform the update on-card. In order to ensure that we send the right one, we model its application on an initial version of bytecode $P1$ as annotations. The operation of annotating a bytecode with expressions indicating where an update instruction occurs and what is the operation involved can be defined recursively as an annotation function which transforms a program to an annotated program.

$$Annot([], P) \equiv P$$

$$Annot(Upd_i :: \Delta, P) \equiv let P' = Add\_Annot\_Line(Upd_i, P)\ in\ Annot(\Delta, P')$$

The annotation of a program with an empty DIFF file ([]) is the program itself otherwise, the function iterates over the update operations starting from the update instruction at the head of the DIFF file (Upd_i) in the DIFF file and adds a corresponding annotated line ($Add\_Annot\_Line(Upd_i, P)$) to the program. The rest of update instruction (Δ) is then considered. The symbol :: is a list constructor linking the head of the list to its rest. Figure 8 shows an annotated program obtained by the application of a DIFF file on an initial bytecode. The annotations are represented as special comments. For example, *Del 4*: deletes the instruction at program counter (*pc*) 4 and *add isub 4*, adds the instruction *isub* at *pc* 4.

In this framework, we use a functional representation based on the Coq specification language to represent manipulated data (integers, objects and variables), instructions of the sub language, update instructions, programs and annotated programs. The formalisation of these concepts is used to perform the calculus of the specification of updated programs using a dedicated calculus. This calculus and formalisation details are presented in the following subsection.

4.2 WP-based verification

The approach for verification is based on the fact that the update of a program (of its semantics) implies the transformation of its specification. In Hoare (1969) logic, a program $P1$ and its specification is represented by a triple $\{pre1\}P1\{post1\}$ where $pre1$ ($post1$) is the precondition (postcondition) of the program $P1$. A new version of this triple written off-card by the programmer is $\{pre2\}P2\{post2\}$ (called the target triple). The DIFF file is performed with $P1$ and $P2$ and then sent to the card to perform update operations, meaning, obtaining a new bytecode and a new specification. The goal is to establish that the target triple and the obtained triple after performing update operations match.

4.2.1 Interpretation of the update

In order to formally define the update interpreter, we need to define some notions. In this interpretation, a state is modelled by a three-tuple: <*Heap*, *Frame*, *Stack_Frame*> which represents the machine state where *Heap* represents the contents of the heap, *Frame* represents the execution state of the current method and, *Stack_Frame* is a list of frames corresponding to the call stack. A frame contains the following elements: the stack of operands *OperandStack* and the values of the local variables *LocalVar* at the program point *PC* of the method *Method*(<*H*, *Method*, *PC*, *OperandStack*, *LocalVar*>). The definition of the update interpretation is based on the notion of step.

Definition 2: (*Step*) The semantics of an instruction (update instruction) is specified as a function step: *Bytecode_Prog* ∗ *State* ∗ *Specification* –> *State* ∗ *StepName* ∗ *Specification* that, given a bytecode *BC*, a state *S* and a specification *SP*, computes the next state *S*′, the name of the next step and a new specification.

Definition 3: (*Java bytecode update interpreter*) We define an update interpreter (*Upd_int*) which iterates over steps. It takes as parameters an annotated program in its functional representation, an initial state and an initial specification and relies on predicate calculus and update interpretation function to produce a new state and a new specification. The interpreter is defined as $Upd\_int(BC, S) = (S', Sp')$, with $S = initial(BC, Sp)$ representing the function for defining an initial state for the execution of the bytecode *BC* with the initial specification *Sp*. The bytecode *BC* is given with its parameters and an initial heap. The result of the interpreter is a state *S*′ and a new specification *Sp*′.

Definition 4: (Verified updated bytecode)

- Let $P1$ and $P2$ be the first and the new version of a program and P a patch
- let $P2' = annot(P1, P)$ be the program obtained by annotation of $P1$ with P
- let $f(P2')$ the functional representations of $P2'$
- let $\text{spec}(P1) = (pre1, post1)$ the specification of $P1$ and $spec(P2) = (pre2, post2)$ the specification of $P2$

The program $P2'$ is a successfully verified update of $P1$ if and only if: *verification*(*spec*($P2$), *spec*(P2′)) succeeds where *spec*($P2'$) is obtained by a predicate transformation on $f(P2')$ starting from $post2$.

Table 2 Defining rules for weakest precondition calculus for update operations

$\mathbf{wp}(\mathbf{Add\_instr(pop,\ i)}) = (\mathit{shift\_exp}^2(@E_i))$
$\mathbf{wp}(\ \mathbf{Add\_instr(store\ x,\ i)}) = \mathit{shift\_exp}^2(@E_i)(S(0)/x)$
$\mathbf{wp(Add\_nstr(if\ L,\ i))} = ((S(0)=0) \Rightarrow \mathit{shift\_exp}^2(EL)) \wedge (S(0) \neq 0 \Rightarrow \mathit{shift\_exp}^2(@E_i))$
$\mathbf{wp}(\mathbf{Add\_instr(load\ x,\ i)}) = \mathit{unshift\_exp}(\mathit{shift\_exp}(@E_i))(x/S(0))$
$\mathbf{wp}(\mathbf{Add\_instr(const\ a,i)}) = \mathit{unshift\_exp}(\mathit{shift\_exp}(@E_i))(a/S(0))$
$\mathbf{wp}(\mathbf{Add\_instr(new\ A,i)}) = \mathit{unshift\_exp}(\mathit{shift\_exp}(@E_i[\mathit{create}(H,A)/S(0),\ A::H=H]))$
$\mathbf{wp}(\mathbf{Add\_instr(add,\ i)}) = (\mathit{shift\_exp}^2(@E_i))[(s(1)+S(0))/S(1)]$
$\mathbf{wp(Add\_instr(neg,\ i)} = (\mathit{unshift\_exp}(@E_i))[-S(0)/S(0)]$
$\mathbf{wp}(\mathbf{Add\_nstr(getfield\ a\ f\ t,\ i)}) = \mathit{shift\_exp}(@E_i[(\mathit{val}(S(0),(a,f)))/S(0)]) \wedge S(0) \neq \mathit{null}$
$\mathbf{wp}(\mathbf{Add\_instr(putfield\ a\ f\ t,\ i)}) = (\mathit{shift\_exp}^3(@E_i))[H((S(0),(a,f)) := S(1))/H]$ $\wedge S(0) \neq \mathit{null}$
$\mathbf{wp(goto\ l1)} = \mathit{shift\_exp}(E_{l1})$

4.2.2 Weakest precondition calculus

In this section, we define bytecode update logic in terms of a weakest precondition calculus. The proposed weakest precondition (WP) considers that each (update) instruction has a precondition. An instruction with its precondition is called an instruction specification and is noted as: E_i: I_i where I_i is the instruction and the expression E_i its specification. This notation expresses that the precondition E_i holds when the program pointer is at the program counter *i*. Table 2 shows the calculus of the WP rules for the update operations (inserting instructions).

Functions and notations used. The functions *shift_exp* and *unshift_exp* are used to express the effect of pushing (popping) elements to (from) the stack *S* and the effect of shifting an expression regarding to the stack elements due to the insertion of instructions. They are defined as follows:

$$\mathit{shift\_exp}(\mathit{Exp}) = \mathit{Exp}[s(i+1)/s(i)\ \mathit{for\ all}\ i \in N]$$
$$\mathit{unshift\_exp} = \mathit{shift\_exp}^{-1}$$

The elements of the stack are represented by positive integers, the top of stack is 0. The symbol @ is used to express the old specification associated to a position *i*: when an instruction is added at position *i*, the program and the specification are shifted from *i* position and then a new instruction is inserted. Its precondition is calculated with the specification of the instruction that was at position *i* before the update.

In the rules, for the instructions *store x*, *load x* and *pop*, a precondition is obtained, as in Hoare's (1969) assignment by substituting the right-hand side by the left-hand side in the postcondition. The precondition of an instruction *store x* under a postcondition E_{i+1}

(the precondition of the following instruction) is given by: *shift_exp*(E_{i+1})(S(0) / x) meaning that if the expression E holds after the execution of *store x* then it also holds for the top of the stack before storing it in x. The function *shift_exp* is used to express that before the execution of the instruction, the top of the stack corresponding to the instruction at $i + 1$ was at index 1.

Inserting an instruction, e.g., *store x* at line i means that the precondition of the old instruction at i becomes the postcondition of the inserted instruction and thus the calculated precondition starts from this old postcondition ($@E_i$). The function *shift_exp* is used twice (*shift_exp*^{2}) to express also the impact due to the insertion of the instruction on the specifications of the following instructions.

The instructions *new*, *putfield* and *getfield* are heap manipulating instructions. The function *create* used in the instruction *new A* returns a new object of type A in the heapH. This obtained heap (A :: H) replaces the old heap. The function *val* used in the definition of getfield to get the value of the field f of the class a from the address (top of the stack). This value is then pushed on the stack. In *putfield*, the value of the field designated by the top of the stack is updated with the value at the second element of the stack. The insertion of this instruction which pops two values implies three applications of *shift_exp*.

In order to establish semantical equivalence of a code written by the programmer and a program obtained by applying a DIFF file, we check the equivalence of the weakest precondition of an annotated program obtained by WP calculus and a precondition written by the programmer before DIFF file is performed.

4.2.3 Formalisation details

The formalisation of the proposed approach relies on the formalisation on data types and the formalisation of the WP calculus:

- The definition of manipulated data, objects, instructions and update instructions is mainly based on the concepts of lists for both bytecode programs and annotation function. Figure 9 shows a fragment of the Coq formalisation, mainly, the principle data structures. The formalisation starts by defining the data manipulated by the program to formalise the instructions (*instruction*) and update instructions (*update_instruction*). The definition of an instruction is given by the name of a construct (representing the name of the instruction) followed by its arguments. For example, for the instruction new, the construct *new* takes a *class* as argument. The instruction *putfield* is represented by the construct *putfield* followed by three arguments: the class (*class*), the names of the field (*string*) and its type. A bytecode line (*bc_line*) is a record composed of a number and an instruction. An annotated line is based on update instructions. A line in an annotated code is either a standard instruction line (*std*) or an annotation line (*annoted*). An annotated code is a list of annotated lines.
- The formalisation of the WP calculus is mainly based on the notion of stacks. Two categories of stacks are defined: operand stack (*op_stack*) and logical stack (*logical_stack*) (Figure 9). The first stack is used to contain the data manipulated by the instructions of a program. The type *logical_stack* contains logical expressions. It is used to represent the specifications of the instructions of the program (the E_i expressions). The expressions of the logical stack relate to the content of the operand stack and the program instructions. A modification trough an update instruction has

an impact on the logical stack. This impact is defined using the *shift* and *unshift* operations which performs recursively on logical stacks on the base of WP rules. The correspondence between the operand stack, the list of instructions and the stack of the specifications is established through the notion of instruction identifiant (*Exp_id*).

Figure 9 Extract from the formalisation (see online version for colours)

```
Inductive instruction : Set :=
|pop: instruction
|store: var → instruction
|load: var →instruction
|const: value →instruction
|add: instruction
|neg: instruction
|ifL: nat→instruction
|goto: nat→instruction
|new: class→instruction
|getfield: class→string→data_type→instruction
|putfield: class →string →data_type →instruction
|invokevirtual: class → string→data_type →instruction
|retur:instruction.

Inductive update_instruction :Set :=
|Add_instr: instruction→ nat→update_instruction
|Del_instr: instruction → nat →update_instruction.

Record bc_line : Set :=
{num: nat;
inst: instruction}.
Record annot_line : Set:=
{numA: nat ;
annot: update_instruction}.
Inductive annot_code_line :Set :=
|std: bc_line →annot_code_line
|annoted: annot_line→annot_code_line.
Definition annot_code := (list annot_code_line).

Inductive op_stack :Set :=
| empty_op: op_stack
| cons_op: value→ op_stack →op_stack.
Inductive Expr: Type :=
|Exp_id: nat → logic_Exp → Expr.
Inductive logical_stack: Set :=
|empty_ls: logical_stack
|cons_ls:Expr →logical_stack →logical_stack.
```

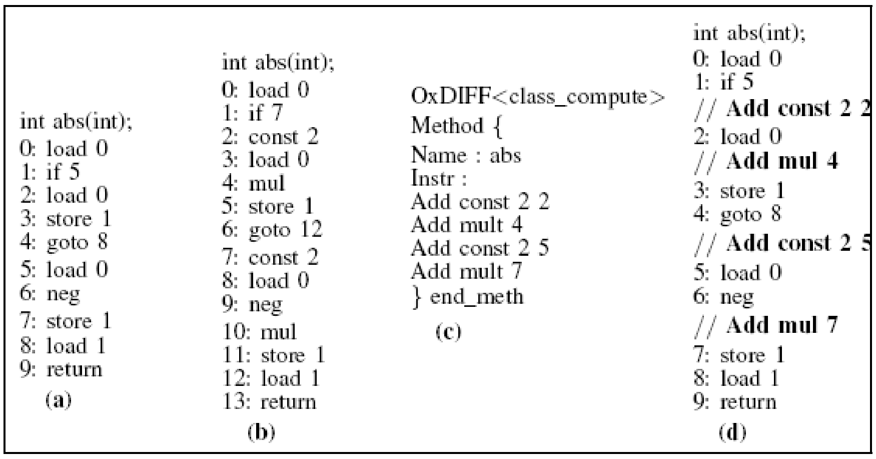
4.2.4 Example of WP calculus

In order to illustrate how the logic works, we take the example of the function *abs* that returns the absolute value of an integer taken as an argument. This function is then transformed in order to get the double of the result in the initial calculus: for an integer given as an argument, the new function returns the abstract value multiplied by two (modified abs). The specifications of the two functions are respectively:

$$\{p = P\}\ abs\ \{P \geq 0 \rightarrow result = P) \wedge (P < 0 \rightarrow result = -P)\}$$

$$\{p = P\}\ \mathrm{mod}\mathit{ified\ abs}\ \{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\}$$

Figure 10 Example of an annotated bytecode (a) the first version (b) the new version (c) the DIFF file (d) the annotated program



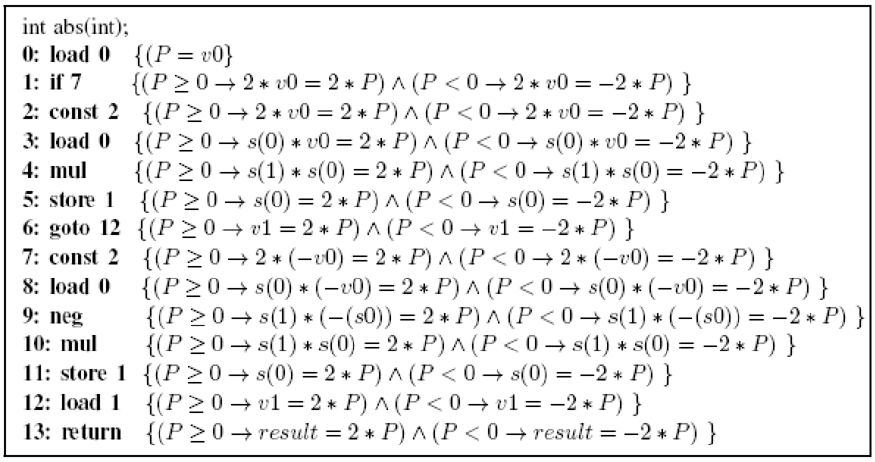
```
int abs(int);
0: load 0
1: if 5
2: load 0
3: store 1
4: goto 8
5: load 0
6: neg
7: store 1
8: load 1
9: return
```
(a)

```
int abs(int);
0: load 0
1: if 7
2: const 2
3: load 0
4: mul
5: store 1
6: goto 12
7: const 2
8: load 0
9: neg
10: mul
11: store 1
12: load 1
13: return
```
(b)

```
OxDIFF<class_compute>
Method {
Name : abs
Instr :
Add const 2 2
Add mult 4
Add const 2 5
Add mult 7
} end_meth
```
(c)

```
int abs(int);
0: load 0
1: if 5
// Add const 2 2
2: load 0
// Add mul 4
3: store 1
4: goto 8
// Add const 2 5
5: load 0
6: neg
// Add mul 7
7: store 1
8: load 1
9: return
```
(d)

Figure 11 WP calculus on the modified function



int abs(int);

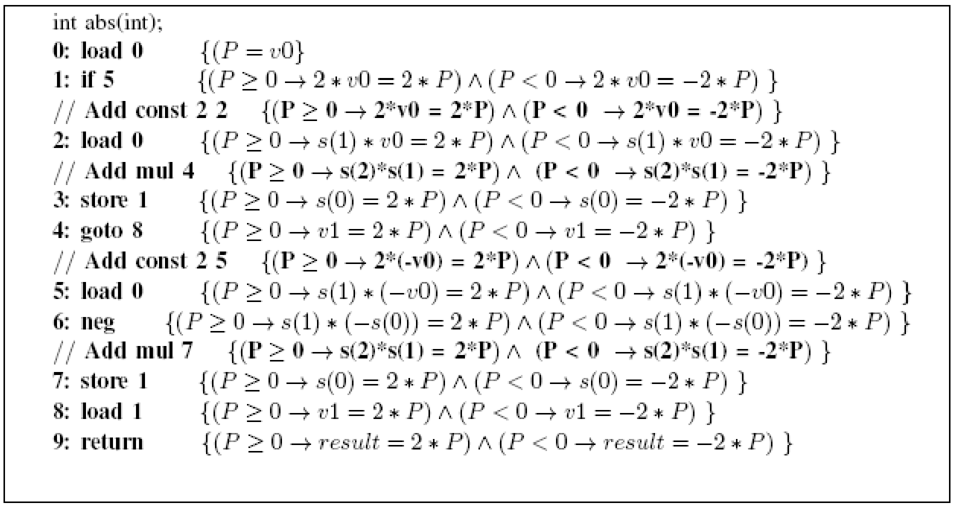
0: load 0	$\{(P = v0\}$
1: if 7	$\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
2: const 2	$\{(P \geq 0 \rightarrow 2 * v0 = 2 * P) \wedge (P < 0 \rightarrow 2 * v0 = -2 * P)\ \}$
3: load 0	$\{(P \geq 0 \rightarrow s(0) * v0 = 2 * P) \wedge (P < 0 \rightarrow s(0) * v0 = -2 * P)\ \}$
4: mul	$\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
5: store 1	$\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
6: goto 12	$\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
7: const 2	$\{(P \geq 0 \rightarrow 2 * (-v0) = 2 * P) \wedge (P < 0 \rightarrow 2 * (-v0) = -2 * P)\ \}$
8: load 0	$\{(P \geq 0 \rightarrow s(0) * (-v0) = 2 * P) \wedge (P < 0 \rightarrow s(0) * (-v0) = -2 * P)\ \}$
9: neg	$\{(P \geq 0 \rightarrow s(1) * (-(s0)) = 2 * P) \wedge (P < 0 \rightarrow s(1) * (-(s0)) = -2 * P)\ \}$
10: mul	$\{(P \geq 0 \rightarrow s(1) * s(0) = 2 * P) \wedge (P < 0 \rightarrow s(1) * s(0) = -2 * P)\ \}$
11: store 1	$\{(P \geq 0 \rightarrow s(0) = 2 * P) \wedge (P < 0 \rightarrow s(0) = -2 * P)\ \}$
12: load 1	$\{(P \geq 0 \rightarrow v1 = 2 * P) \wedge (P < 0 \rightarrow v1 = -2 * P)\ \}$
13: return	$\{(P \geq 0 \rightarrow result = 2 * P) \wedge (P < 0 \rightarrow result = -2 * P)\ \}$

In the specification, *P* denotes the logical value at the entry and *result* is the result of the function. Figure 10 shows the bytecode of the first version (a) and the second version (b) of the described function. The part (c) of the figure shows the DIFF file generated from the two versions. The last part of the figure (d) shows the bytecode of the function *abs* annotated with update instructions. We notice that in this bytecode, local variables are represented by integers: in *load 1* for example, the number 1 means the local variable 1. The same notation is applied to other local variables.

In Figure 11, the WP calculus is performed on the bytecode (without annotation) starting from the post-condition of the new version. The WP calculus is applied on the annotated bytecode as shown on Figure 12. The specifications for the update instructions are in bold. This example shows that we obtain the same precondition $\{P = v0\}$ which means that at the beginning of the calculus the logical value P is in the first local variable of the function. This result expresses the equivalence of the two bytecodes according to the proposed definition of verified updated program. This ensures the behavioural correctness of a type safe updated program according to a DIFF file.

The application of the DIFF file in order to obtain a type safe updated program with the intended specification has to consider the execution state of the program in order to preserve the safety of the system. This concern is discussed in the next section.

Figure 12 WP calculus on an annotated bytecode



```
int abs(int);
0: load 0          {(P = v0}
1: if 5            {(P ≥ 0 → 2 * v0 = 2 * P) ∧ (P < 0 → 2 * v0 = −2 * P) }
// Add const 2 2     {(P ≥ 0 → 2*v0 = 2*P) ∧ (P < 0  → 2*v0 = -2*P) }
2: load 0          {(P ≥ 0 → s(1) * v0 = 2 * P) ∧ (P < 0 → s(1) * v0 = −2 * P) }
// Add mul 4       {(P ≥ 0 → s(2)*s(1) = 2*P) ∧  (P < 0  → s(2)*s(1) = -2*P) }
3: store 1         {(P ≥ 0 → s(0) = 2 * P) ∧ (P < 0 → s(0) = −2 * P) }
4: goto 8          {(P ≥ 0 → v1 = 2 * P) ∧ (P < 0 → v1 = −2 * P) }
// Add const 2 5     {(P ≥ 0 → 2*(-v0) = 2*P) ∧ (P < 0  → 2*(-v0) = -2*P) }
5: load 0          {(P ≥ 0 → s(1) * (−v0) = 2 * P) ∧ (P < 0 → s(1) * (−v0) = −2 * P) }
6: neg         {(P ≥ 0 → s(1) * (−s(0)) = 2 * P) ∧ (P < 0 → s(1) * (−s(0)) = −2 * P) }
// Add mul 7       {(P ≥ 0 → s(2)*s(1) = 2*P) ∧  (P < 0  → s(2)*s(1) = -2*P) }
7: store 1         {(P ≥ 0 → s(0) = 2 * P) ∧ (P < 0 → s(0) = −2 * P) }
8: load 1          {(P ≥ 0 → v1 = 2 * P) ∧ (P < 0 → v1 = −2 * P) }
9: return          {(P ≥ 0 → result = 2 * P) ∧ (P < 0 → result = −2 * P) }
```

5 Ensuring activeness safety

The application of the updates raises the question of active methods. Indeed, a naive modification may lead to system inconsistency: updating a method while it is active leads to the use of different versions of the same method and thus, generating an incoherent behaviour of the system. This situation is avoided by ensuring activeness safety property. This property ensures that an update may be performed only if the functions modified by the update are not active. It implies that the modified functions are not on the stack of a running program. This is ensured by analysing the applications to define SUP. We propose, in this section, a formalisation and a verification of a technique used to compute a SUP in EmbedDSU and other systems. We verify that it guarantees the correctness of the system by ensuring activeness safety.

5.1 SUP detection

This subsection presents a description of the concepts used in the technique of safe update point detection and its mechanism.

5.1.1 Methods and virtual machine modes

Methods are considered from activeness point of view. Four cases are noticed:

- *Active/not active methods:* a method is active if it is running. This means that the method owns a frame in the execution stack because it has been invoked. Otherwise, a method is not active.
- *Restricted/not restricted methods:* a method is restricted if it is active in the VM and concerned by the update. A method is not restricted if it is not concerned by the update whatever it is active or not.

The search of the appropriate point to perform DSU is performed by a safe update detection algorithm. This algorithm ensures that no restricted method is executing during the update. The update process defines three modes for the virtual machine:

- *The standard mode:* during this mode, the virtual machine works normally until detecting an update.
- *The pending mode:* at update detection, the virtual machine switches to seek a safe update point mode. A SUP corresponds to a state where restricted methods are not executing.
- *The update mode:* after detecting a SUP, the update process is performed at the levels described in Section 2.

The main idea is to force methods involved into an update to finish. This mechanism provides a highest priority to such methods that will be popped from the stack.

5.1.2 Search SUP mode functions

In order to detect a SUP, a function *searchSUP* introspects the frames for each thread of the running application. If the update is possible (no restricted methods in the frames), then the update mode is set to true and the search SUP mode is set to false.

If restricted methods are present in the stack, we obtain the number of frames associated to restricted methods by VM introspection. It is stored in a variable *counter*. The search SUP mode uses the following functions to lead the systems to a safe update point:

- *createFrame:* this function is used at every method invocation to create an associated frame. The frame is placed at the head of the frame list of the thread. This function is adapted to handle DSU mechanism during research SUP mode: at every created frame, a check is performed. If the thread has no frames related to a restricted methods in its execution stack then it is blocked. This policy avoids having other frames related to a method to be modified in the stack thread. The goal is to let restricted methods finish their execution and lead them to inactive state.
- *releaseFrame:* this function is used at the end of the execution of a method to suppress its frame. In the search SUP mode, this function is adapted in order to check if restricted methods exist in the thread stack. If not, the thread is suspended. The counter *counter*, associated to the executing thread, is decremented at every release

of a frame associated to a restricted method. A safe update point is reached if this number reaches zero for every thread.

- *switchThread:* if the VM is in the normal mode, this function selects the next thread to be the active one. If the mode is pending and if all the remaining threads are blocked then it switches the update mode. No restricted method is present in any thread, we have reached a SUP.

At the starting point, the process counts all frames related to a modified method present in the stack thread. If the value is not equals to zero, then the update is pending, the virtual machine can continue to execute other applications. However, the value is decremented each time these methods finish their execution. When the value equals zero, then the SUP is obtained and the virtual machine can switch to the update mode.

5.2 Ensuring activeness safety criteria

We present in this subsection a formal verification of activeness safety. The formalisation is based on a functional modelling provided by the Why3 platform (Bobot et al., 2016). We use the Why3 platform to write specifications. This platform provides the language Why3 which is a functional language with logics. After specification, the platform offers an interface to different provers in which proof obligations can be discharged. The choice of this platform is motivated by the high level of expressiveness of the language for both computational and logic level and the large choice of provers supported by the platform including the Coq proof assistant.

5.2.1 Specification

This subsection presents the proposition of a functional formalisation of the DSU mechanism to reach a SUP. Precisely, our specification relies on:

- a formalisation of the notion of time as a functional stream, virtual machine structures and modelling of behaviours as modes across time;
- a formalisation of predicates and functions related to the update mechanism;
- a formalisation of theorems to state activeness safety property.

In Why3, specifications are based on the notions of theory. A theory is a list of declarations. Declarations introduce new types, functions and predicates, state axioms, lemmas and goals. Figure 13 shows the specification of types used in the formalisation.

We represent the notion of time as a functional stream. Streams represent a potentially infinite sequence of data of the same kind. The evaluation of a part of a stream is done on demand, whenever it is needed by a current computation. The stream type is an abstract data type; one does not need to know how it is implemented. In this formalisation, a stream of instants represented by integers (*type instant = I int*) is considered.

Figure 13 Extract of types required by the specification (see online version for colours)

```
type Stream
type time = T Stream instant
type id_meth = Id_meth
type id_cls = Id_cls
type id_thr= Id_thr
type flag = On int | Off int
type value = Val_Int int
type thr_status= Running |Waiting |Blocked |DSU_blocked
type event = Upd_req |Upd_end |End_frame |New_frame
type vm_mode = Std |Sup |Upd
type behavior = Behav Stream instant vm_mode
type meth = {m_id: id_meth; owner: id_cls;loc: variable list; code: bc; sign:types list ;upd_flag: flag }
type clas = {c_id:id_cls; table_m: meth list; table_f: field list; var: variable list }
type thread = { t_id:id_thr; status: thr_status, t_fr: frame list }
type frame = {i_fr: id_fr; m_fr: id_meth; op_stk: value list; pc:int; loc: list value; t_fr: id_thr }
```

The type *flag* is used to indicate a restricted method. It is set to one if a method concerned by the update is active. A method is specified by its identifier, signature, the list of its local variables and owner (the class of the method). Another information in method called *upd_flag* is defined. This information is used to indicate whether a method is concerned or not by an update. The *upd_flag* corresponds initially to zero. If the method is concerned by an update (according to a DIFF file) then this flag is changed.

A thread is represented by its identifier, its list of frames, its owner and its status. A thread passes trough different states. In standard mode, a thread is scheduled then executed (*Running*). If it meets a sleep instruction, it goes to the sleeping state (*Waiting*). The thread gets blocked if it wants to acquire a lock but it cannot because another thread owns it, the active thread has to go to the blocked state (*Blocked*) and wait until another thread releases the lock. In our modelisation, another state is necessary. The thread gets *DSU_blocked* if the virtual machine enters the search SUP mode to force running restricted methods.

The virtual modes of the virtual machine are represented by the type *vm_mode*, enumerating respectively the standard mode, the search SUP mode (pending) and the update mode. The evolution of the modes during time is represented as VM behaviour.

Figure 14 Some predicates used by the specification (see online version for colours)

```
predicate is_updated (m:meth) = m.upd_flag= On 0
predicate is_restricted (m:meth)(t:thread)= is_updated(m)
    ∧ exists f:frame. inlist(f,t.t_fr) ∧ m.m_id=f.meth
predicate is_running (t: thread) = t.thr_status = Running
predicate is_dsu_blk (t:thread) = t.thr_status = DSU_blocked
```

Figure 14 presents some illustration of predicates related to the types. For instance, the predicate *is_restricted* is used to state that a method is restricted if it is concerned by the update (*is_updated* predicate) and if it has frames in execution. The predicates are used in both modelling the functions representing the mechanism and writing theorems that ensures activeness safety property.

5.2.2 Verification

The process of verification is based on the Why3 platform. This framework generates from a Why3 language file a set of proofs that guarantee the correctness of the established theory. Figure 15 represents some theorems form our specification, the three lemmas are related to the three modes of the virtual machine. The first lemma ensures that updated methods are not running during the update. The second lemma related to the standard mode, guarantees that all the threads that have been blocked during the search SUP mode to reach a SUP are unlocked after the update release. The third lemma is crucial to establish the activeness safety property. It states that during the execution, a virtual machine passes through different modes (represented by a stream associating modes to instants). It states that during the search mode, the number of restricted method is decreasing across time (from an instant t1 to t2). The predicate *before* is used to express that t1 is before t2 in the stream.

Figure 15 represents two goals. The first goal, establishes that the virtual machine reaches the SUP. Indeed, the number of restricted method reaches zero. This goal is principally based on the *sup_mode*1 lemma. The second goal ensures that a started update terminates and the virtual machine returns to the standard mode. These two goals guarantee the system safety.

The Why3 platform provides interfaces with different proof systems (provers and proof assistants). Due to the high expressiveness required by the formalisation, we worked on the Coq proof assistant.

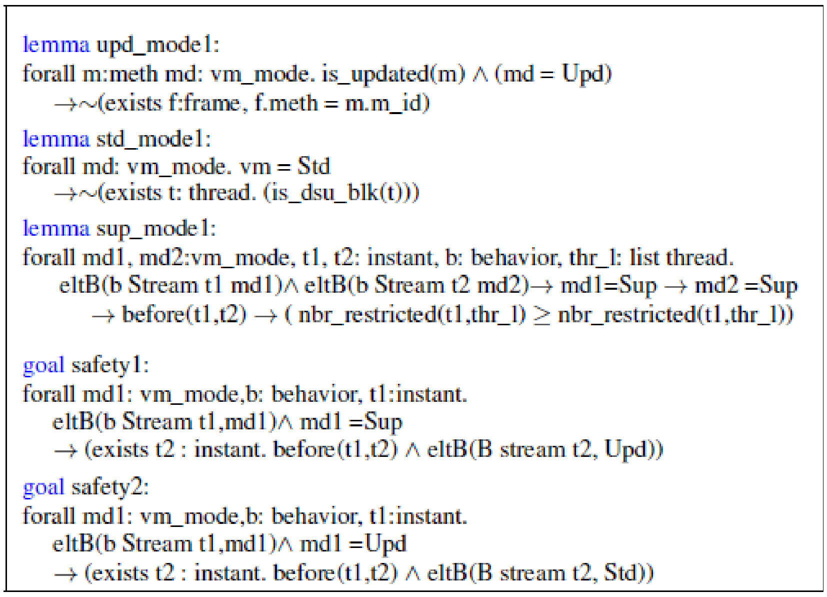
6 Related work

Several studies have been conducted in order to use formal methods to ensure DSU correctness. The concept of DSU correctness is not unique and relates to the different scientific problems (code update, data update and update timing). Correctness criteria are divided in literature into two categories. The first category relies on common properties that are shared by all updates such as type safety (Hjálmtýsson and Gray, 1998; Neamtu et al., 2006; Makris, 2009; Zhang et al., 2012) and activeness safety (Hayden, 2012). The second category refers to specific properties related to the semantics of updated programs (Hayden et al., 2012; Anderson and Rathke, 2009; Charlton et al., 2011). They are also called behavioural properties. Techniques and systems related to DSU are surveyed in Seifzadeh et al. (2013) and Miedes and Munoz-Escoi (2012).

Several formalisms are used to establish DSU correctness criteria. Zhang et al. (2012) used an algebraic formalism to ensure correctness of DSU systems based on the mechanism of POLUS (Chen et al., 2007). The programs is formalised in terms of sort and operations and the update mechanism as a rewriting system. This work focuses on two types of correctness: common property correctness and correctness based on properties defined by the user. The process of verification is based on three parts: choose an initial configuration, formalise properties and then verify. Type safety is specified as a predicate. System rewriting is then used to establish the property. Other works used different formalisms to ensure type safety such as functional formalism and type system in respectively (Anderson and Rathke, 2009; Duggan, 2005).

Behavioural properties are studied in Charlton et al. (2011): the authors provided a framework such that the desired properties are expressed within the updated code using Hoare logic (HL) style Hoare (1969) by writing preconditions, post-conditions and assertions within the code. The system computes proof obligations which are discharged by theorem proving. In Anderson (2013), the behavioural properties are expressed in a type system extended with effects. The idea behind this work is that the correctness of an update depends on a state characterised by the code and the shared resources. The type system ensures that the modified system will behave as expected by keeping track of the effect of each update operation. The formalism includes a notion of world constraints to keep the difference between the effect of an update operation and the expected specification of an update. The language considered is a typed lambda-calculus with recursion and threads.

Figure 15 Theorems from our formalisation (see online version for colours)



```
lemma upd_mode1:
forall m:meth md: vm_mode. is_updated(m) ∧ (md = Upd)
    →~(exists f:frame, f.meth = m.m_id)
lemma std_mode1:
forall md: vm_mode. vm = Std
    →~(exists t: thread. (is_dsu_blk(t)))
lemma sup_mode1:
forall md1, md2:vm_mode, t1, t2: instant, b: behavior, thr_l: list thread.
     eltB(b Stream t1 md1)∧ eltB(b Stream t2 md2)→ md1=Sup → md2 =Sup
         → before(t1,t2) → ( nbr_restricted(t1,thr_l) ≥ nbr_restricted(t1,thr_l))

goal safety1:
forall md1: vm_mode,b: behavior, t1:instant.
    eltB(b Stream t1,md1)∧ md1 =Sup
    → (exists t2 : instant. before(t1,t2) ∧ eltB(B stream t2, Upd))
goal safety2:
forall md1: vm_mode,b: behavior, t1:instant.
    eltB(b Stream t1,md1)∧ md1 =Upd
    → (exists t2 : instant. before(t1,t2) ∧ eltB(B stream t2, Std))
```

In our work, the contribution is at both system level and single program level. First, we presented formal semantics for update operations and established type safety. Using semantics to prevent type errors in bytecode, the contribution extends the formalism presented in Freund and Mitchell (1999). This work defined semantics and a type system to study object initialisation in bytecode. The original idea was developed in Stata and Abadi (1999) to study bytecode subroutines. Freund and Mitchell (2003) extended the work (Freund and Mitchell, 1999) to bytecode subroutines, virtual method invocation and exceptions. Qian et al. (2000) proposed formal semantics for Java class loading. The specificity of the formalisation is the definition of semantics related to operations about class loading, loaders delegating, class renaming and data structures related to class loading mechanism.

Our work is close to Freund and Mitchell (1999) in the use of static semantics to analyse bytecode. The main difference between our formalisation and existing ones is the fact that ours is based on the definition of update operations. It ensures that each update operation operates on data with appropriate types. Performing an update operation, for instance inserting an instruction, requires the readjustment of the code to take into account the inserted instruction. The formal semantics define data structures and operations to check typing information and code readjustment. The other main difference is the formalisation of an extension of the Java card virtual machine (JCVM) to implement DSU. Our work is the first attempt to formalise such an extension on the JCVM. Indeed, the proposed formal semantics guarantees that the defined operations of the system preserve type safety with regard to virtual machine specifications.

The second established property is behavioural correctness of the updated code. Using predicate transformation to reason about bytecode properties has been studied in Grégoire et al. (2008). The authors presented a verification condition generator for bytecode formalised in the Coq proof assistant and based on weakest precondition calculus. Another work using weakest precondition to generate verification conditions from an annotated bytecode is presented in Burdy and Pavlova (2006) and Burdy et al. (2007). Our bytecode logic for weakest precondition calculus is inspired by Bannwart and Müller (2005). It allows reasoning at single program level. We presented a Hoare-style logic combined with instruction specification in term of precondition. The main difference between the proposed WP calculus and existing predicate transformations is the fact that we presented WP rules dedicated to update operations to ensure update correctness with regard to specifications written by the programmer on the basis of DIFF file information.

Correctness criteria related to update timing are addressed in literature through two major approaches: the first category improves the programming language to offer the possibility to insert update points within the code. Stoyle et al.'s (2007) updates are performed at points satisfying *confreeness* configurations: the code is labelled with update expressions at points where the update is possible in addition with the types that must not be updated. These points are inferred by a static analysis called updateability analysis. The major drawback of this category is that prediction techniques relies generally on the modification of the semantics of the programming language to offer the possibility to insert update points within the code.

The second category (Hayden, 2012; Noubissi et al., 2011) relies on mechanisms to introspect the state of the application and drive it to a quiescent mode and then performs the update. Establishing activeness safety property is formally established using a bisimulation technique for C-like programs in Hayden (2012) and functional model for component- based programs in Buisson et al. (2016). Other properties related to update timing do exist in literature. We studied other properties in Lounas et al. (2017). In the present paper, the formalisation aims to verify the mechanism of searching safe update point for DSU for Java Card with regard to activeness safety by specifying both the mechanism and properties using a functional specification and based on the concept of streams.

7 Conclusions

DSU consists in updating running programs on-the-fly without any downtime. This feature is important in critical systems that must run continuously. The use of DSU in critical systems leads to the use of formal methods that offer the high level of guarantee required by such applications. In this paper, we proposed an approach for formal specification and verification of DSU in Java Card applications.

We verified three properties: first, we established that update operations are well typed by defining a formal semantics with regard to Java Card specification and type safety. This semantics is used to establish well-formed updates. Secondly, we established behavioural correctness of updated bytcodes. We proposed an approach that relies on a dedicated weakest precondition calculus to establish that the updated program matches the programmer's specifications. Our third contribution consists in the specification of the update mechanism in order to ensure the safety of the update in term of timing. We specified the activeness safety property that guarantees that the update occurs at a safe point.

Our work is based on functional representation. This is motivated by the fact that this kind of specification offers high expressiveness and eases the integration of our work with existing formal methods to construct proofs. We integrated our work in theWhy3 framework but other formal frameworks can be considered.

Our study started with considering the system EmbedDSU but this is not restrictive. The proposed framework can be generalised to specification and verification of updated programs written in languages that are compiled to bytecode. Besides, several research works rely on the principle of searching a safe update point to perform DSU. We believe that the presented contribution could be adapted.

As a future work, we plan to extend our work to include exceptions in the formalisation of update operations and to extend the functional formalisation to verify other parts of the DSU system such as instances update in the heap.

References

Anderson, A. and Rathke, J. (2009) 'Migrating protocols in multi-threaded message-passing systems', *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades (HotSWUp)*.

Anderson, G. (2013) *Behavioural Properties and Dynamic Software Update for Concurrent Programmes*, Phd Thesis, University of Southampton.

Arnold, J. and Kaashoek, M.F. (2009) 'Ksplice: automatic rebootless kernel updates', *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pp.187–198.

Bannwart, F and Müller, P, (2005) 'A program logic for bytecode', in *Electron. Notes Theor. Comput. Sci.*, Vol. 141, pp.255–273, Elsevier Science Publishers B.V.

Baumann, A., Kerr, J., Da Silva, D., Krieger, O. and Wisniewski, R.W. (2009) 'Module hot-swapping for dynamic update and reconfiguration in K42', in *6th Linux.Conf.*, Au.

Bobot, F., Filliâtre, J-C., Marché, C., Melquiond, G. and Paskevich, A. (2016) The *Why3 Platform Manual*, University Paris-Sud, CNRS, Inria.

Buisson, J., Dagnat, F., Leroux, E. and Martinez, S. (2016) 'Safe reconfiguration of Coqcots and Pycots components', *Journal of Systems and Software*, Vol. 122, pp.430–444.

Burdy, L. and Pavlova, M. (2006) 'Java bytecode specification and verification', in *SAC 2006*, pp.1835–1839.

Burdy, L., Huisman, M. and Pavlova, M. (2007) 'Preliminary design of BML: a behavioral interface specification language for Java bytecode', in *FASE 2007*, pp.215–229.

Charlton, N., Horsfall, B. and Reus, B. (2011) 'Formal reasoning about runtime code update', in *ICDE Workshops*, pp.134–138.

Chen, H., Yu, J., Chen, R., Zang, B. and Yew, P-C. (2007) 'POLUS: a powerful live updating system', *Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society, pp.271–281.

Chen, J. and Huang, L. (2009) 'Dynamic service update based on OSGi', *WRI World Congress on Software Engineering, (WCSE)*, pp.493–497.

Common Criteria (2015)[online] http:///www.commoncriteriaportal.org/ (accessed 28 September 2017).

Duggan, D. (2005) *Type-based Hot Swapping of Running Modules*,pp.181–220, Acta Inf. Springer-Verlag, New York, Inc.

Freund, S.N and Mitchell, J.C 'A type system for object initialization in the Java bytecode language', in *ACM Trans. Program. Lang. Syst.*, pp.1196–1250.

Freund, S.N and Mitchell, J.C. (2003) 'A type system for the Java bytecode language and verifier', *In J. Autom. Reasoning*, Vol. 30, No. 3, pp.271–321.

Grégoire, B, Sacchini, J.L and Sivan, R. (2008) 'Combining a verification condition generator for a bytecode language with static analyses', in *Proceedings of the 3rd conference on Trustworthy global computing*, Springer-Verlag, pp.23–40.

Gupta, D. and Jalote, P. (1993) 'On line software version change using state transfer between processes', *Software – Practice and Experience*, Vol. 23, No. 9, pp.949–964.

Hayden, C.M., Smith, E.K, Hicks, M., and Foster, J.S. (2011) 'State transfer for clear and efficient runtime upgrades', in *Proceedings of the 3rd International Workshop on Hot Topics in Software Upgrades, HotSWUp 2011*, IEEE Computer Society.

Hayden, C.M. (2012) *Clear, Correct and Efficient Dynamic Software Updates*, PhD Thesis, University of Maryland, USA.

Hayden, C.M., Magill, S., Hicks, M., Foster, N. and Foster, J.S. (2012) 'Specifying and verifying the correctness of dynamic software updates', *Proceedings of the 4th International Conference on International Conference on Verified Software: Theories, Tools, Experiments (VSTTE).*

Hjálmtýsson, G. and Gray, R. (1998) 'Dynamic C++ classes: a lightweight mechanism to update code in a running program', *Proceedings of the Annual Conference on USENIX Annual Technical Conference.*

Hoare, C.A.R, (1969) 'An axiomatic basis for computer programming', in *Commun. ACM*, Vol. 12, pp.576–580.

Lounas, R., Jafri, N., Legay, A., Mezghiche, M. and Lanet, J-L. (2016) 'A formal verification of safe update point detection in dynamic software updating', in Cuppens, F., Cuppens, N., Lanet, J.L. and Legay, A. (Eds.): *Risks and Security of Internet and Systems*, Vol. 10158, *CRiSIS,* Lecture Notes in Computer Science, Springer, Cham.

Lounas, R., Mezghiche, M. and Lanet, J-L. (2015) 'An approach for formal verification of updated java bytecode programs', in Hedia, B.B. and Vladicescu, F.P. (Eds.): *Proceedings of the 9th Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS*, Bucharest, Romania, 10–11 September 2015.

Lounas, R., Mezghiche, M. and Lanet, J-L. (2014) 'Mise á jour dynamique des applications JavaCard: Une approche pour une mise á jour sûre du tas', *Conférence en IngénieriE du Logiciel* (*CIEL*), CNAM, Paris, France.

Lv, W., Zuo, X. and Wang, L. (2012) 'Dynamic software updating for onboard software', *Second International Conference on Intelligent System Design and Engineering Application (ISDEA)*, pp.251–253.

Makris, K. and Ryu, K.D. (2007) 'Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels', *SIGOPS Oper. Syst. Rev.*, pp.327–340.

Makris, K. (2009) *Whole Program Dynamic Software Updating*, PhD Thesis, Arizona State University, USA.

Miedes, E. and Munoz-Escoi, F.D. (2012) *A Survey About Dynamic Software Updating*, Technical report ITI-SIDI-2012/003, University of Valencia, Spain.

Neamtu, I., Hicks, M., Stoyle, G. and Oriol, M. (2006) 'Practical dynamic software updating for C', *Conference on Programming Language Design and Implementation, (PLDI)*, *Proceedings of the 27th ACM SIGPLAN*, pp.72–83.

Noubissi, A.C. (2011) *Mise á jour dynamique et sécurisée de composants systéme dans une carte á puce*, PhD Thesis, University of Limoges, France.

Noubissi, A.C., Iguchi-Cartigny, J. and Lanet, J.L. (2011) 'Hot updates for Java based smart cards', in *ICDE Workshops*, pp.168–173.

Qian, Z., Goldberg, A. and Coglio, A. (2000) 'A formal specification of Java class loading', *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp.325–336.

Secrétariat général de la défense et de la sécurité nationale (ANNSI) (2015) 'Security requirements for post-delivery code loading', *Agence Nationale de la Sécurité des Systèmes d' Information*, Paris.

Seifzadeh, H, Kazem, A.A.P, Kargahi, M and Movaghar, A. (2009) 'A method for dynamic software updating in real-time systems', *Eighth IEEE/ACIS International Conference on Computer and Information Science (ICIS),* pp.34–38.

Seifzadeh, H., Abolhassani, H. and Moshkenani, M.S. (2013) 'A survey of dynamic software updating', *Journal of Software: Evolution and Process*, Vol. 25, No. 5, pp.535–568.

Stata, R. and Abadi, M. (1999) 'A type system for Java bytecode subroutine', in *ACM Trans. Program. Lang. Syst.*, Vol. 21, pp.90–137.

Stoyle, G., Hicks, M., Bierman, G., Sewell, P. and Neamtiu, I. (2007) 'Mutatis mutandis: safe and predictable dynamic software updating', *ACM Trans. Program. Lang. Syst.*

Subramanian, S., Hicks, M. and McKinley, K.S. (2009) 'Dynamic software updates: a VM-centric approach', *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, pp.1–12.

Wahler, M., Richter, S. and Oriol, M. (2009) 'Dynamic software updates for real-time systems', *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades (HotSWUp)*, pp.2:1–2:6

Zhang, M., Ogata, K. and Futatsugi, K. (2012) 'An algebraic approach to formal analysis of dynamic software updating mechanisms', *AsiaPacific Software Engineering Conference (APSEC)*, pp.664–673.

Appendix

We give in this appendix the remaining rules for update operations related to instruction insertion and suppression. Remarks:

- In the rules for variables, the symbol $\perp$ represents the default value the added variables are initialised with. The *Change* operation is used to update the map accordingly and the set of local variables of a method (*Im.loc*) is updated in each rule according to the type of the operation.
- In the instruction *invokevirtual*, the function *dom* represents the domain of the invoked function (types of its arguments) and the function card represents the number of elements in the domain.

Table 3 Rules for Introducing, deleting and modifying variables

$\dfrac{\begin{array}{l} Add\_Loc(x, tx, m) \\ look\_for\_var(Im.loc, x, tx) = false \\ Im.loc \leftarrow Im.loc \cup \{(x, tx)\} \\ v(x) \leftarrow \perp \\ M2 = Change(M1) \end{array}}{F^c, S^c \vdash C}(Rv1)$	$\dfrac{\begin{array}{l} Del\_Loc(x, tx, m) \\ look\_for\_var(Im.loc, x, tx) = false \\ Im.loc \leftarrow Im.loc \setminus \{(x, tx)\} \\ M2 = Change(M1) \end{array}}{F^c, S^c \vdash C}(Rv2)$
$\dfrac{\begin{array}{l} Mof\_Loc(x, tx, val, m) \\ look\_for\_var(Im.loc, x, tx) = true \\ v(x) \leftarrow val \\ M2 = Change(M1) \end{array}}{F^c, S^c \vdash C}(Rv3)$	

Table 4 Rules for update operations (insertion of instructions)

$\dfrac{\begin{array}{l} Add\_inst\ goto\ L(i+1) \\ SD_{i+1} = SD_i \ \ PC\_MAX++ \\ S_{i+1} = S_i \quad F_{i+1} = F \\ M2 = Add\_inst(M1, got\ L, i+1) \\ i+1, L \in DOM(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Add\_inst\ store\ x(i+1) \\ SD_{i+1} = SD_i - 1 \quad PC\_MAX++ \\ S_i = t.S_0 \quad F_i[x \leftarrow t] \\ S_{i+1} = S_0 \\ M2 = Add\_inst(M1, store\ x, i+1) \\ i+1 \in DOM(BC)\ x \in VAR(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$
$\dfrac{\begin{array}{l} Add\_inst\ pop(i+1) \\ SD_{i+1} = SD_i - 1 \ \ F_{i+1} = F_i \\ S_i = t.S_0 \rightarrow S_{i+1} = S_0 \\ M2 = Add\_inst(M1, pop, i+1) \\ PC\_MAX++ \\ i+1 \in DOM(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Add\_inst\ putfield(A, f, t)(i+1) \\ SD_{i+1} = SD_i - 2 \ \ F_{i+1} = F_i \\ S_i = t.A.S_0 \Rightarrow S_{i+1} = S_0 \\ M2 = Add\_inst(M1, putfield(A, f, t), i+1) \\ PC\_MAX + 3 \quad i+1 \in DOM(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$

Table 4 Rules for update operations (insertion of instructions) (continued)

$\dfrac{\begin{array}{l} Add\_inst\ getfield(A, f, t)(i+1) \\ SD_{i+1} = SD_i \\ S_i = A.S_0 \Rightarrow S_{i+1} = t.S_0 \\ M2 = Add\_inst(M1, getfield(A, f, t), i+1) \\ PC\_MAX + 3 \quad F_{i+1} = F_i \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Add\_inst\ load\ x(i+1) \\ SD_{i+1} = SD_i + 1 \\ PC\_MAX{+}{+} \\ S_{i+1} = F_i[x].S_i \quad F_{i+1} = F_i \\ M2 = Add\_inst(M1, load\ x, i+1) \\ i+1 \in DOM(BC)\ x \in VAR(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$
$\dfrac{\begin{array}{l} Add\_inst\ invokevirtual(A, l, t)(i+1) \\ SD_{i+1} = SD_i - \left(card\left(dom(t)\right) + 1\right) \\ S_{i+1} = tn1.tn2 \geq \ldots tn_n.S_0 \to S_{i+1} = S_0 \\ M2 = Add\_inst(M1, invokevirtual(A, l, t), i+1) \\ i+1 \in DOM(BC) \quad F_{i+1} = F_i \\ PC\_MAX + 3 \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Add\_inst\ if\ \ L(i+1) \\ SD_{i+1} = SD_i \\ PC\_MAX{+}{+} \\ S_{i+1} = S_i \quad F_{i+1} = F_i \\ M2 = Add\_inst(M1, if\ L, i+1) \\ i+1 \in, L \in DOM(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$
$\dfrac{\begin{array}{l} Add\_inst\ const\ a(i+1) \\ SD_{i+1} = SD_i + 1 \\ PC\_MAX{+}{+} \\ S_{i+1} = int.S_i \quad F_{i+1} = F_i \\ M2 = Add\_inst(M1, const\ a, i+1) \\ i+1 \in DOM(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Add\_inst\ neg(i+1) \\ SD_{i+1} = SD_i \quad F_{i+1} = F_i \\ S_i = int.S_0 = S_{i+1} \\ M2 = Add\_inst(M1, neg, i+1) \\ PC\_MAX{+}{+} \\ i+1 \in DOM(BC) \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$

Table 5 Rules for update operations (suppression of instructions)

$\dfrac{\begin{array}{l} Dlt + inst\ goto\ L(i+1) \\ SD_i = a \to \\ SD_{i+1} = Effects\_SD\left(a, M2[i+1]\right) \\ M2 = Dlt\_inst(M1, gotoL, i+1) \\ (M2)S_{i+1} = Effects\_STK\left(M2[i+1], S_i\right) \\ (M2)F_{i+1} = Effects\_F\left(M2[i+1], F_i\right) \\ i+1, L \in DOM(BC)\ PC\_MAX{-}{-} \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Dlt + inst\left(store\ x\ (i+1)\right) \\ SD_i = a \to \\ SD_{i+1} = Effects\_SD\left(a, M2[i+1]\right) \\ M2 = Dlt\_inst(M1, store\ x, i+1) \\ S_i = t.S_0, F_i[x] = t \to \\ (M2)S_{i+1} = Effects\_STK\left(M2[i+1], t.S_0\right) \\ (M2)F_{i+1} = Effects\_F\left(M2[i+1], F_i\right) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \to \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$

Table 5 Rules for update operations (suppression of instructions) (continued)

$\dfrac{\begin{array}{l} Dlt + inst\ if\ L(i+1) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, ifL, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC\_MAX -- \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Dlt + inst\ (pop\ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, pop, i+1) \\ S_i = t.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], t.S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$
$\dfrac{\begin{array}{l} Dlt + inst\ (putfield(A, f, t)\quad (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, putfield(A, f, t), i+1) \\ S_i = A.t.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Dlt + inst\ (getfield(A, f, t)\ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, getfield(A, f, t), i+1) \\ S_i = A.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], A.S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$
$\dfrac{\begin{array}{l} Dlt + inst\ (neg\ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, neg, i+1) \\ S_i = int.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1, L \in DOM(BC)\ PC\_MAX -- \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Dlt + inst\ (load\ x\ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, load\ x, i+1) \\ (M1)S_{i+1} = t.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_0) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$
$\dfrac{\begin{array}{l} Dlt + inst\ (const\ a\ (i+1)) \\ SD_i = a \rightarrow \\ SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, const\ a, i+1) \\ S_i = S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX -- \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$	$\dfrac{\begin{array}{l} Dlt + inst\ (invokevirtuel(A, l, t)\ (i+1)) \\ SD_i = a \rightarrow SD_{i+1} = Effects\_SD(a, M2[i+1]) \\ M2 = Dlt\_inst(M1, invokevirtuel(A, l, t), i+1) \\ S_i = tn_1.tn_2 \ldots tn_n.S_0 \rightarrow \\ (M2)S_{i+1} = Effects\_STK(M2[i+1], S_i) \\ (M2)F_{i+1} = Effects\_F(M2[i+1], F_i) \\ i+1 \in DOM(BC)\ PC\_MAX - 3 \end{array}}{\langle F_i, S_i, SD_i, M1, i\rangle \rightarrow \langle F_{i+1}, S_{i+1}, SD_{i+1}, M2, i+1\rangle}$