Java Bytecode Specification and Verification

Lilian Burdy
INRIA Sophia-Antipolis
Lilian.Burdy@sophia.inria.fr

Mariela Pavlova
INRIA Sophia-Antipolis
Mariela.Pavlova@sophia.inria.fr

Abstract

We propose a framework for establishing the correctness of untrusted Java bytecode components w.r.t. to complex functional and/or security policies. To this end, we define a bytecode specification language (BCSL) and a weakest precondition calculus for sequential Java bytecode. BCSL and the calculus are expressive enough for verifying non-trivial properties of programs, and cover most of sequential Java bytecode, including exceptions, subroutines, references, object creation and method calls.

Our approach does not require that bytecode components are provided with their source code. Nevertheless, we provide a means to compile JML annotations into BCSL annotations by defining a compiler from the Java Modeling Language (JML) to BCSL. Our compiler can be used in combination with most Java compilers to produce extended class files from JML-annotated Java source programs.

All components, including the verification condition generator and the compiler are implemented and integrated in the Java Applet Correctness Kit (JACK).

1. Introduction

Establishing trust in software components that originate from untrusted or unknown producers is an important issue in areas such as smart card applications, mobile phones, bank cards, ID cards and whatever scenario where untrusted code should be installed and executed.

In particular, the state of the art proposes different solutions. For example, the verification may be performed over the source code. In this case, the code receiver should make the compromise to trust the compiler, which is problematic. Bytecode verification techniques [12] are another solution, which does not require to trust the compiler. The bytecode verifier performs static analysis directly over the bytecode yet, it can only guarantee that the code is well typed and well structured. The Proof Carrying Code paradigm (PCC) and the certifying compiler [14, 15] are another alternative. In this architecture, untrusted code is accompanied by a proof for its safety w.r.t. to some safety property and the code receiver has just to generate the verification conditions and type check the proof against them. The proof is generated automatically by the certifying compiler for properties like well typedness or safe memory access. As the certifying compiler is designed to be completely automatic, it will not be able to deal with rich functional or security properties.

We propose a bytecode verification framework with the following features:

- a bytecode specification language and a compiler from source program annotations into bytecode annotations. Thus, bytecode can benefit from the source specification and does not need to be accompanied by its source code.
- verification condition generator over Java bytecode, which supports the bytecode annotation.

In a client-producer scenario, the first point brings to the producer means to supply the sufficient specification information which will allow the client to establish trust in the code, especially when the client policy is potentially complex and a fully automatic specification inference will fail. On the other hand, the second point enables the client to check the untrusted annotated code.

Our approach is tailored to Java bytecode. The Java technology is widely applied to mobile and embedded components because of its portability across platforms. For instance, its dialect JavaCard is largely used in

smart card applications and the J2ME Mobile Information Device Profile (MIDP) finds application in GSM mobile components.

The proposed scheme is composed of several components. We define a bytecode specification language, called BCSL, and supply a compiler from the high level Java specification language JML [10] to BCSL. BCSL supports a JML subset which is expressive enough to specify rich functional properties. The specification is inserted in the class file format in newly defined attributes and thus makes not only the code mobile but also its specification, allowing the Java bytecode benefit from the source annotation. These class file extensions do not affect the JVM performance. We define a bytecode logic in terms of weakest precondition calculus for the sequential Java bytecode language. The logic gives rules for almost all Java bytecode instructions and supports the Java specific features such as exceptions, references, method calls and subroutines. We have implementations of a verification condition generator based on the weakest precondition calculus and of the JML specification compiler. Both are integrated in the Java Applet Correctness Kit tool (JACK) [7].

The full specifications of the JML compiler, the weakest precondition predicate transformer definition and its proof of correctness can be found in [16].

The remainder of the paper is organized as follows: Section 2 reviews scenarios in which the architecture may be appropriate to use; Section 3 provides a brief overview of related work; Section 4 presents the bytecode specification language BCSL and the JML compiler; Section 5 discusses the main features of the weakest precondition calculus; Section 6 discusses the relationship between the verification conditions for JML annotated source and BCSL annotated bytecode; Section 7 concludes with future work.

2. Framework

The overall objective is to allow a client to trust a code produced by an untrusted code producer. Our approach is especially suitable in cases where the client policy involves non trivial functional or safety requirements and thus, an automatic specification inference can not be applied. To this end, we propose a PCC technique that exploits the JML compiler and the weakest predicate function presented in the article.

The framework is presented in Fig. 1; note that certificates and their checking are not yet implemented and thus are in oblique font.

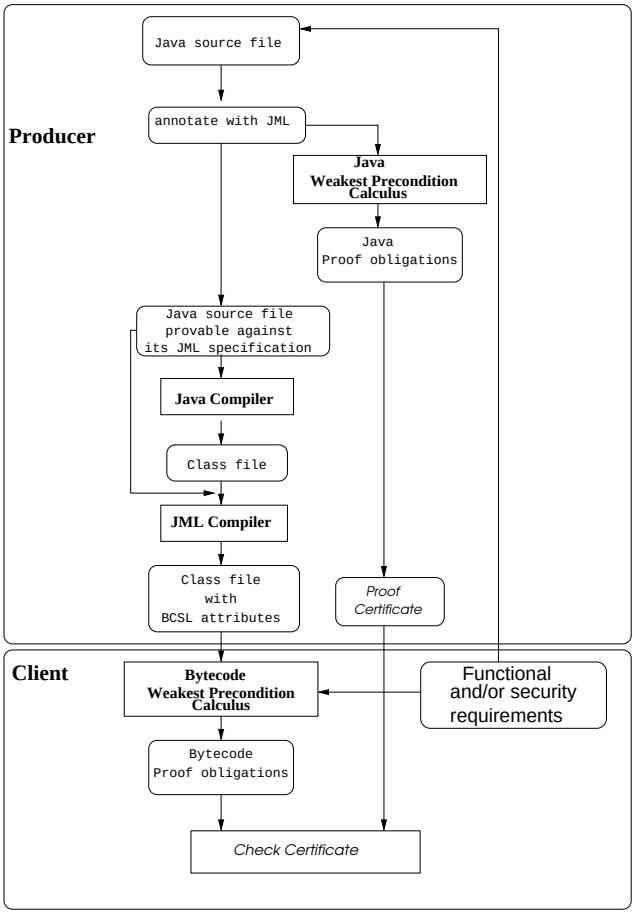

Figure 1. THE OVERALL ARCHITECTURE

In the first stage of the process the client provides the functional and (or) security requirements to the producer. The requirements can be in different form:

- A specified interface that describes the application to be developed. In that case, the client specifies in JML the features that have to be implemented by the code producer.
- An API with restricted access to some method. In this case, the client can protect its system by restricting the API usage. For example, suppose that the client API provides transaction management facilities - the API method `open` for opening and method `close` for closing transactions. In this case, a requirement can be for no nested transactions. This means that the methods `open` and `close` can be annotated to ensure that the method `close` should not be called if there is no transaction running and the method `open` should not be called if there is already a running transaction. In this scenario, we can apply results of previous work [17].

Typically, the development process involves annotating the source code with JML specification, generating verification conditions, using proof obligation generator over the source code and discharging proofs which represent the program safety certificate and finally, the producer sends the certificate to the client along with the bytecode. Yielding certificates over the source code is based on the observation that proof obligations on the source code and non-optimized bytecode respectively are syntactically the same modulo names and basic types. Every Java file of the untrusted code is normally compiled with a Java compiler to obtain a class file. Every class file is extended with user defined attributes that contain the BCSL specification, resulting from the compilation of the JML specification of the corresponding Java source file.

To implement this architecture we use JACK [7] as a verification condition generator both on the consumer and the producer side. JACK is a plugin for the eclipse[1] integrated development environment for Java. Originally, the tool was designed as verification condition generator for Java source programs against their JML specification. JACK can interface with several theorem provers (AtelierB, Simplify, Coq, PVS). We have extended the tool with a compiler from JML to BCSL and a bytecode verification condition generator. In the next sections, we introduce the BCSL language, the JML compiler and the bytecode weakest precondition calculus which underlines the bytecode verification condition generator.

1 http://www.eclipse.org

3. Related Work

We now review research works which treat similar problematic.

JVer [8] is a tool for verifying that downloaded Java bytecode programs do not abuse client computational resources. The bytecode programs are annotated with pre and postconditions written in a subset of JML specification language. The tool, however, doesnot support a compiler from high level specification annotations into bytecode annotations.

In [2], P.Muller and F.Bannwart define a Hoare logic over a bytecode language with objects and exceptions. A compiler from source proofs into bytecode proofs is also defined. To our knowledge, subroutines are not treated. Invariants are inferred by fixpoint calculation, differently from the approach presented here, where all specification clauses including loop invariants are compiled from the high level JML specification (see section 4.2).

The traditional PCC and the certifying compiler proposed by Necula (see [14, 15]) is an architecture for establishing trust in unknown code in which the code producer accompanies the code with a proof certificate. Differently from our approach, as the certifying compiler infers automatically a type specification such as loop invariants and generates automatically the proof certificate it is not applicable for complex security policies.

There are also other close areas to our work, but we do not describe them because of space limitations. Among them are bytecode verification for which Xavier Leroy in [12] gives detailed overview, the definition of bytecode logic e.g. the work of C. Quigley [18] where a Hoare style bytecode logic is defined and the research of Nick Benton (see [4]) which combines both functional correctness and well typedness for a stack based bytecode language. The curious reader may also take a look at Spec# [3] which is a static verification framework and where the method and class contracts are inserted in the intermediate code and the verification is done over the intermediate code.

4. Bytecode Specification Language (BCSL)

In this section, we introduce a bytecode specification language which we call BCSL (short for ByteCode Specification Language). BCSL is based on the design principles of JML (Java Modeling Language) [19, 10],

which is a behaviorial interface specification language following the design by contract approach [5].

Before going farther, we give a flavour of what JML specifications look like. Fig. 2 shows an example of a Java class and its JML annotation that models a list stored in an array field. From the example, we notice that JML annotations are written in comments and so they are not visible by the Java compiler. The specification of method `isElem` declares that when the method is called the field `list` must not be null in its precondition (introduced by `requires`) and that its return value will be true if and only if the internal array `list` contains the object referenced by the argument `obj` in its postcondition(`ensures`). The method loop is also specified by its invariant (`loop_invariant`) which states that whenever the loop entry is reached the elements inspected already by the loop are all different from `obj`.

```
public class ListArray {
  Object[] list;
  //@requires list != null;
  //@ensures \result == (\exists int i;0 <= i &&
             i < list.length && list[i] == o ) ;
  public boolean isElem(Object obj)
  {
    int i = 0;
    //@loop_modifies i;
    //@loop_invariant i <= list.length && i >=0
    //@  && (\forall int k;0 <= k && k < i ==>
    //@  list[k] != obj);
    for (i = 0; i < list.length; i++ ) {
      if ( list[i] == obj) {
        return true;
      }
    }
    return false;
  }
}
```

Figure 2. CLASS LISTARRAY WITH JML ANNOTATIONS

In the following, we give the grammar of BCSL and sketch the compiler from JML to BCSL.

4.1. Grammar

BCSL corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties.

Specification clauses in BCSL that are taken from JML and inherit their semantics directly from JML include:

- class specification, i.e. class invariants and history constraints
- method preconditions, normal and exceptional postconditions, method frame conditions (the locations that may be modified by the method). We also support behavioral subtyping by specification inheritance as described in [9].
- inter method specification, for instance loop invariants
- predicates from first order logic
- expressions from the programming language, like field access expressions, local variables, etc.
- specification operators. For instance `\old(`$\mathcal{E}$`)` which is used in method postconditions and designates the value of the expression $\mathcal{E}$ in the prestate of a method, `\result` which stands for the value the method returns if it is not void

BCSL has few particular extra features that JML lacks :

- loop frame condition, which declares the locations that can be modified during a loop iteration. We were inspired for this by the JML extensions in JACK [7]
- stack expressions - `c` which stands for the stack counter and `st(Arithmetic_Expr)` standing for a stack element at position `Arithmetic_Expr`. These expressions are needed in BCSL as the Java Virtual Machine (JVM) is stack based. They do not appear in the specification clauses. The reason for this is that proving the soundness of the bytecode weakest precondition predicate transformer function becomes considerably complicated. This is not a restriction in the scenario in which we use BCSL specification — compiling JML to BCSL specification, as JML doesnot contain stack expressions.

4.2. Compiling JML into bytecode specification language

We now turn to explaining how JML specifications are compiled into user defined attributes for Java class files. Recall that a class file defines a single class or interface and contains information about the class name,

interfaces implemented by the class, super class, methods and fields declared in the class and references. The Java Virtual Machine Specification (JVMS) [13] mandates that the class file contains data structure usually referred as the **constant_pool** table which is used to construct the runtime constant pool upon class or interface creation. The runtime constant pool serves for loading, linking and resolution of references used in the class. The JVMS allows to add to the class file user specific information([13], ch.4.7.1). This is done by defining user specific attributes (their structure is predefined by JVMS).

Thus the "JML compiler" [2] compiles the JML source specification into user defined attributes. The compilation process has three stages:

1. Compilation of the Java source file. This can be done by any Java compiler that supplies for every method in the generated class file the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [13], yet almost all standard non optimizing compilers can generate these data. The **Line_Number_Table** describes the link between the source line and the bytecode of a method. The **Local_Variable_Table** describes the local variables that appear in a method. Those attributes are important for the next phase of the JML compilation.

2. Compilation of the JML specification from the source file and the resulting class file. In this phase, Java and JML source identifiers are linked with their identifiers on bytecode level, namely with the corresponding indexes either from the constant pool or the array of local variables described in the **Local_Variable_Table** attribute. If, in the JML specification a field identifier appears for which no constant pool (cp) index exists, it is added in the constant pool and the identifier in question is compiled to the new cp index. It is also in this phase that the specification parts like the loop invariants and the assertions which should hold at a certain point in the source program must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. The compilation of an expression is a tag followed by the compilation of its subexpressions.

   Another important issue in this stage of the JML compilation is how the type differences on source and bytecode level are treated. By type differences we refer to the fact that the JVM (Java Virtual Machine) does not provide direct support for intergral types like byte, short, char, neither for boolean. Those types are rather encoded as integers in the bytecode. Concretely, this means that if a Java source variable has a boolean type it will be compiled to a variable with an integer type. For instance, in the example for the method `isElem` and its specification in Fig.2 the postcondition states the equality between the JML expression `\result` and a predicate. This is correct as the method `isElem` in the Java source is declared with return type boolean and thus, the expression `\result` has type boolean. Still, the bytecode resulting from the compilation of the method `isElem` returns a value of type integer. This means that the JML compiler has to "make more effort" than simply compiling the left and right side of the equality in the postcondition, otherwise its compilation will not make sense as it will not be well typed. Actually, if the JML specification contains program boolean expressions that the Java compiler will compile to bytecode expression with an integer type, the JML compiler will also compile them in integer expressions and will transform the specification condition in equivalent one[3].

   Finally, the compilation of the postcondition of method `isElem` is given in Fig. 3. From the postcondition compilation, one can see that the expression `\result` has integer type and the equality between the boolean expressions in the postcondition in Fig.2 is compiled into logical equivalence. The example also shows that local variables and fields are respectively linked to the index of the register table for the method and to the corresponding index of the constant pool table (#19 is the compilation of the field name `list` and `lv[1]` stands for the method parameter `obj`).

3. add the result of the JML compilation in the class file as user defined attributes. Method specifications, class invariants, loop invariants are newly defined attributes in the class file. For example, the specifications of all the loops in a method are compiled to a unique method attribute: whose syntax is given in Fig. 4. This attribute is an array of data structures each describing a sin-

2 Gary Leavens also calls his tool jmlc JML compiler, which transforms jml into runtime checks and thus generates input for the jmlrac tool

3 when generating proof obligations we add for every source boolean expression an assumption that it must be equal to 0 or 1. Actually, this must always hold as we assume that programs are well typed

$$\texttt{\result} = 1$$

$$\Longleftrightarrow$$

$$\exists var(0).\left(\begin{array}{l} 0 \leq var(0) \wedge \\ var(0) < len(\#19(\texttt{lv[0]})) \wedge \\ \#19(\texttt{lv[0]})[var(0)] = \texttt{lv[1]} \end{array}\right)$$

Figure 3. The compilation of the postcondition in Fig. 2

gle loop from the method source code. Also for each loop in the source code there must be a corresponding element in the array. More precisely, every element contains information about the instruction where the loop starts as specified in the **Line_Number_Table**, the locations that can be modified in a loop iteration, the invariant associated to this loop and the decreasing expression in case of total correctness,

```
JMLLoop_specification_attribute {
    ...
    {  u2 index;
       u2 modifies_count;
       formula modifies[modifies_count];
       formula invariant;
       expression decreases;
    } loop[loop_count];
}
```

- **index**: The index in the `LineNumberTable` where the beginning of the corresponding loop is described
- **modifies**[]: The array of locations that may be modified
- **invariant** : The predicate that is the loop invariant. It is a compilation of the JML formula in the low level specification language
- **decreases**: The expression which decreases at every loop iteration

Figure 4. Structure of the Loop Attribute

The JML compiler does not depend on any specific Java compiler, but it requires the presence of a debugging information, namely the presence of the **Line_Number_Table** attribute for the correct compilation of inter method specification, i.e. loops and assertions. We think that this is an acceptable restriction for the compiler. The most problematic part of the compilation is to identify which source loop corresponds to which bytecode loop in the control flow graph. To do this, we assume that the control flow graph is reducible (see [1]), i.e. there are no jumps from outside a loop inside it; graph reducibility allows to establish the same order between loops in the bytecode and source code level and to compile the invariants to the correct places in the bytecode.

5. Weakest Precondition Calculus For Java Bytecode

In this section, we define a bytecode logic in terms of a weakest precondition calculus. We assume that the bytecode program has passed the bytecode verification procedure, i.e. it is well typed and thus the calculus deals only with program functional properties.

The proposed weakest precondition *wp* supports all Java bytecode sequential instructions except for floating point arithmetic instructions and 64 bit data (`long` and `double` types), including exceptions, object creation, references and subroutines. The calculus is defined over the method control flow graph and supports BCSL annotation, i.e. bytecode method's specification like preconditions, normal and exceptional postconditions, class invariants, assertions at particular program point among which loop invariants.

In Fig. 5, we show the *wp* rules for some bytecode instructions. As the examples show the *wp* function takes three arguments: the instruction for which we calculate the precondition, the instruction's postcondition ψ and the exceptional postcondition function ψ^{exc} which for any exception `Exc` returns the corresponding exceptional postcondition $\psi^{exc}(\texttt{Exc})$. The function *wp* must satisfy the following property: if the instruction `ins` starts execution in a state where the predicate $wp(\texttt{ins}, \psi, \psi^{exc})$ holds then if it terminates normally then the poststate must satisfy the predicate ψ and if terminates on exception `Exc` then the poststate must satisfy $\psi^{exc}(\texttt{Exc})$. In the draft paper [16], we show that the *wp* function has this property (i.e. the calculus is correct). The proof is done by defining an operational semantics for the bytecode instructions and exploits several substitution lemmas.

Returning back to the example, the expression `c` and `st(c)` stand respectively for the the stack counter and the element on the top of the stack. This is because the JVM is stack based, i.e. the instructions take their arguments from the method execution stack and put the result on the stack. The *wp* rule for `Type_load i` increments the stack counter `c` and

loads on the stack top the contents of the local variable `lv[i]`.

In the rest of the section, we consider the following specific features: instance fields, method invocations, loops, exception handling and subroutines.

$$wp(\texttt{Type\_load}\ i,\ \psi,\ \psi^{exc}) = \psi[\texttt{c} \leftarrow \texttt{c}+1][\texttt{st(c+1)} \leftarrow \texttt{lv[i]}]$$

where i is a valid local variable index

$$wp(\texttt{putField Cl.f},\ \psi,\ \psi^{exc}) = \left(\begin{array}{l} \texttt{st(c-1)} \neq \texttt{null} \Rightarrow \\ \quad \psi\ \begin{array}{l}[\texttt{c} \leftarrow \texttt{c}-2] \\ [\texttt{Cl.f} \leftarrow \texttt{Cl.f} \oplus [\texttt{st(c-1)} \rightarrow \texttt{st(c)}]]\end{array} \\ \wedge \\ \texttt{st(c-1)} = \texttt{null} \Rightarrow \\ \quad \psi^{exc}(\texttt{NullPointerException})\ \begin{array}{l}[\texttt{c} \leftarrow 0] \\ [\texttt{st(0)} \leftarrow \texttt{st(c)}]\end{array} \end{array}\right)$$

Figure 5. RULES FOR SOME BYTECODE INSTRUCTIONS

5.1. Manipulating object fields

Instance fields are treated as functions, where the domain of a field `f` declared in the class `Cl` is the set of objects of class `Cl` and its subclasses. We are using function update when assigning a value to a field reference as, for instance in [6]. In Fig.5, we give the *wp* rule for the instruction `putfield Cl.f`, which updates the field `Cl.f`[4] of the object referenced by the reference stored in the stack below the stack top `st(c-1)` with the value on the stack top `st(c)`. Note that the rule takes in account the possible exceptional termination.

5.2. Method calls

Method calls are handled by using their specification. A method specification is a contract between callers and callees — the precondition of the called method must be established by the caller at the program point where the method is invoked and its postcondition is assumed to hold after the invocation. The rule for invocation of a non-void instance method is given in Fig. 6. In the precondition of the called method, the formal parameters and the object on which the method is called are substituted with the first *n+1* elements from the top of stack. Because the method returns a value, if it terminates normally, any occurrence of the JML keyword `\result` in $\psi^{post}(m)$ is substituted with the fresh variable $fresh\_var$. Because the return value in the normal case execution is put on the stack top, the $fresh\_var$ is substituted for the stack top in ψ. The resulting predicate is quantified over the expressions that may be modified by the called method. We also assume that if the invoked method terminates abnormally, by throwing an exception of type `Exc`, on returning the control to the invoker its exceptional postcondition $\psi^{exc}_{\texttt{m}}(\texttt{Exc})$ holds. The rule for static methods is rather the same except for the number of stack elements taken from the stack.

$$\begin{array}{l} wp(\texttt{invoke}\ \ \texttt{m},\ \psi,\ \psi^{exc}) = \\ \quad \psi^{pre}(m) \wedge \\ \quad \forall_{j=1..s} e_j. \left(\begin{array}{l} \psi^{post}(\texttt{m})\ \begin{array}{l}[\texttt{lv[i]} \leftarrow \texttt{st(c+i-nArg(m))}]_{\texttt{i=0}}^{\texttt{nArg(m)}} \\ [\backslash\texttt{result} \leftarrow \texttt{fresh\_var}]\end{array} \\ \quad \Rightarrow \\ \quad \psi\ \begin{array}{l}[\texttt{c} \leftarrow \texttt{c} - \texttt{nArg(m)}] \\ [\texttt{st(c)} \leftarrow \texttt{fresh\_var}]\end{array} \end{array}\right) \\ \wedge_{i=1}^{k} \\ \quad \forall_{j=1..s} e_i \left(\begin{array}{l} \psi^{exc}_{\texttt{m}}(\texttt{Exc}_\texttt{i}) \\ \quad \Rightarrow \\ \quad \phi_{\texttt{Exc}_\texttt{i}}\ \begin{array}{l}[\texttt{c} \leftarrow 0] \\ [\texttt{st(0)} \leftarrow \texttt{st(c)}]\end{array} \end{array}\right) \end{array}$$

$\psi^{pre}(\texttt{m})$ - the specified precondition of method `m`
$\psi^{post}(\texttt{m})$ - the specified postcondition of method `m`
$\psi^{exc}_{\texttt{m}}$ - the exceptional function for method `m`
`nArg( m )` - the number of arguments of `m`
$e_j, j = 1..s$ - the locations modified by method `m`
$\texttt{Exc}_\texttt{i}, \texttt{i} = 1..\texttt{k}$ - the exceptions that `m` may throw
$\phi_{\texttt{Exc}_\texttt{i}}, i = 1..k$ - the precondition of the exception handler protecting the instruction against $\texttt{Exc}_\texttt{i}$ if it exists, otherwise if the exception $\texttt{Exc}_\texttt{i}$ is not handled
$\phi = \psi^{exc}(\texttt{Exc}_\texttt{i})[\texttt{EXC} \leftarrow \texttt{st(c)}]$

Figure 6. *wp* RULE FOR A CALL TO AN INSTANCE NON VOID METHOD

4 `Cl.f` stands for the field `f` declared in class `Cl`

5.3. Loops

Finding the preconditions of loops on bytecode and source programs is different because of their different nature — the first one lacks while the second has structure. While on source level loops correspond to loop statements, on bytecode level we have to analyse the control flow graph in order to identidy them. The analysis consists in finding the backedges in the control flow graph using standard techniques as described in [1].

We assume that a method's bytecode is provided with sufficient specification and in particular loop invariants. Under this assumption, we build an abstract control flow graph where the backedges are replaced by the corresponding invariant. We apply the *wp* function over the abstract version of the control flow graph which generates verification conditions for the preservation and initialization of every invariant in the abstraction garph.

5.4. Exceptions and Subroutines

Exception handlers are treated by identifying the instruction at which the handler compilation starts. The JVM specification mandates that a Java compiler must supply for every method an **Exception_Table** attribute that contains data structures describing the compilation of every implicit (in presence of subroutines) or explicit exception handler: the instruction at which the compiled exception handler starts, the protected region (its start and end instruction indexes), and the exception type the exception handler protects from. Thus, for every instruction `ins` in method `m` which may terminate exceptionally on exception `Exc` the exceptional function ψ^{exc} (Fig.5, Fig.6) returns the *wp* predicate of the exceptional handler protecting `ins` from `Exc` if such a handler exists. Otherwise, ψ^{exc} returns the specified exceptional postcondition for exception `Exc` as specified in the specification of method `m` .

Subroutines are treated by abstract inlining[5]. First, the instructions of every subroutine are identified. To this end, we suppose that the bytecode has been certified by a bytecode verifier which guarantees that there are no recursive subroutines. We also assume that every subroutine terminates with a `ret` instruction[6]. Thus, by abstract inlining, we mean that whenever the *wp* function is applied to `jsr ind`, a postcondition ψ and an exceptional postcondition function ψ^{exc}, its precondition $wp^{\texttt{jsr ind}}$ is calculated as follows: first *wp* is applied to the bytecode instructions of the subroutine starting at instruction `ind`, the postcondition ψ and the exceptional postcondition function ψ^{exc}. This actually results in the weakest predicate $wp^{\texttt{jsr ind}}$ of the subroutine starting at index `ind` and which guarantees that after its execution ψ will hold in the normal case, otherwise if the subroutine terminates on exception `Exc` then $\psi^{exc}(\texttt{Exc})$ will hold.

5 NB: we do not transform the bytecode. It is rather the *wp* function that treats subroutines as if the subroutines were inlined

6 Normally, the compilation of subroutines ends with a `ret` instruction. Still, Java compilers can sometimes also generate a subroutine ending with a jump

6. Comparison between source and bytecodes proofs

The purpose of this section is to give a comparison between bytecode and source proof obligations. In particular, we illustrate this by the proof obligations of the example program in Fig.2.

We studied the relationship between the source code proof obligations generated by the standard feature of JACK and the bytecode proof obligations generated by our implementation over the corresponding bytecode produced by a non optimizing compiler over the examples given in [11]. The proof obligations were the same modulo program variables names and basic types.

We return now to our example from the previous sections and give in Fig.7 one of the proof obligations on source and bytecode level respectively concerning the postcondition correctness. The verification conditions on bytecode and source level have the same shape modulo names (see Section 4.2 for how names are compiled). Also in Section 4.2, we discussed the compilation of the JML postcondition from Fig. 2. Particularly, we saw that the compiler has to transform the source postcondition in an equivalent formula and we gave the compilation in Fig.3.

Despite those transformations, the source and bytecode goal respectively (which are actually the postcondition) on bytecode and source level are not only semantically equivalent but syntactically the same (except for the variable names). Still, in the bytecode proof obligation we have one more hypothesis than on source level. The extra hypothesis in the bytecode proof obligation is related to the fact that the result type is boolean but the JVM encodes boolean expressions as integers (which is trivially true). This means that the proof obligations have also the same shape.

Another important issue is the impact of simple optimizations like dead code elimination on the relationship between source and bytecode proof obligations. In this case, the compiler does not generate the dead code and the bytecode verification condition generator will neither "see" it. Even though the source con-

tains the never taken branch as the condition is equivalent to false, this will result in a trivially true verification condition which the JACK source verification condition generator will discard.

The equivalence between source and bytecode proof obligations can be exploited in PCC scenarios, as we discussed in Section 1 where the producer generates the program certificate over the source code in scenarios where a complete automatic certification (e.g. the certifying compiler) will not work.

We aim to formally give evidence that the proof obligations on non optimized bytecode and source programs are syntactically the same (modulo names and types).

7. Conclusion and Future Work

This article describes a bytecode weakest precondition calculus applied to a bytecode specification language (BCSL). BCSL is defined as suitable extensions of the Java class file format. Implementations for a proof obligation generator and a JML compiler to BCSL have been developed and are part of the Jack 1.8 release[7]. At this step, we have built a framework for Java program verification. This validation can be done at source or at bytecode level in a common environment: for instance, to prove lemmas ensuring bytecode correctness all the current and future provers plugged in Jack can be used.

We envisage several directions for future work:

- perform case studies
- build the missing part of the PCC architecture described in Section 1 where the proofs are done interactively over the source code and then compiled down to bytecode. As we discussed in Section 6 in the performed tests the proof obligations generated over a source program and over its compilation with non optimizing compiler are syntactically the same modulo names and types. We aim to establish formally this property.
- an extension of the framework applying previous research results in automated annotation generation for Java bytecode (see [17]). The client thus will have the possibility to verify a security policy by propagating properties in the loaded code and then by verifying that the code verify the propagated properties.

7 http://www-sop.inria.fr/everest/soft/Jack/jack.html

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers-Principles, Techniques and Tools*. Addison-Wesley: Reading, 1986.

[2] F. Bannwart and P. Muller. A program logic for bytecode. In *Bytecode 2005*, 2005.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In Springer, editor, *in CASSIS workshop proceedings*, 2004.

[4] N. Benton. A typed logic for stack and jumps. draft, 2004.

[5] B.Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 revised edition, 1997.

[6] R. Bornat. Proving pointer programs in Hoare Logic. In *MPC*, pages 102–126, 2000.

[7] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[8] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Jver: A java verifier. In *CAV*, pages 144–147, 2005.

[9] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Dec. 1997. Also in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 258–267. Available by anonymous ftp from ftp.cs.iastate.edu.

[10] G.T.Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[11] B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects, volume 2852 of Lecture Notes in Computer Science*, pages pages 202–219. Springer, Berlin, 2003.

[12] X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[13] T. Lindholm and F. Yellin. Java virtual machine specification. Technical report, Java Software, Sun Microsystems, Inc., 2004.

[14] G. Necula. *Compiling With Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[15] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 17-19 June 1998. Montreal, Canada.

Hypothesis on bytecode:	**Hypothesis on source level:**
$\texttt{lv[2]}\_at\_ins\_20 \geq$ $len(\#19(\texttt{lv[0]}))$	$\texttt{i}\_at\_ins\_26 \geq$ $len(ListArray.list(\texttt{this}))$
$\#19(\texttt{lv[0]}) \neq \texttt{null}$	$ListArray.list(\texttt{this}) \neq \texttt{null}$
$\texttt{lv[2]}\_at\_ins\_20) \leq$ $len(\#19(\texttt{lv[0]}))$	$\texttt{i}\_at\_ins\_26 \leq$ $len(ListArray.list(\texttt{this}))$
$\texttt{lv[2]}\_at\_ins\_20 \geq 0$	$\texttt{i}\_at\_ins\_26 \geq 0$
$\forall var(0).\ 0 \leq var(0) \wedge$ $var(0) < \texttt{lv[2]}\_at\_ins\_20 \Rightarrow$ $\#19(\texttt{lv[0]})[var(0)] = \texttt{lv[1]}$	$\forall var(0).\ 0 \leq var(0) \wedge$ $var(0) < \texttt{i}\_at\_ins\_26 \Rightarrow$ $ListArray.list(\texttt{this})[var(0)] = \texttt{obj}$
$typeof(\texttt{lv[0]}) <: ListArray$	$typeof(\texttt{this}) <: ListArray$
$0 = 0 \vee 0 = 1$	
Goal on bytecode:	**Goal on source level:**
$false \iff$ $\exists var(0).\ 0 \leq var(0) \wedge$ $var(0) < len(\#19(\texttt{lv[0]})) \wedge$ $\#19(\texttt{lv[0]})[var(0)] = \texttt{lv[1]}$	$false \iff$ $\exists var(0).\ 0 \leq var(0) \wedge$ $var(0) < len(ListArray.List(this)) \wedge$ $ListArray.List(this)[var(0)] = \texttt{obj}$

Note: $\mathcal{E}\_at\_ins\_n$ denotes the value of expression $\mathcal{E}$ at the bytecode instruction at index (source line) n

Figure 7. Comparison of source and bytecode verification conditions

[16] M. Pavlova. Java bytecode logic and specification. Technical report, INRIA, Sophia-Antipolis, 2005. Draft version. Available from `http://www.inria.fr/everest/Mariela.Pavlova`.

[17] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In *CARDIS 2004*. Springer-Verlag, 2004.

[18] C. Quigley. A programming logic for Java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[19] A. Raghavan and G. Leavens. Desugaring JML method specification. Report 00-03d, Iowa State University, Department of Computer Science, 2003.