Java Bytecode Specification and Verification

Lilian Burdy Mariela Pavlova

August 22, 2005

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

The present paper addresses the problem of establishing trust in software components that originate from untrusted or unknown producers. The question concerns important areas like smart card applications, mobile phones, bank cards, ID cards and whatever scenario where untrusted code should be installed and executed.

In particular, depending on what is the level of trust the code receiver wants to establish, 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. The bytecode verification technique proposes another solution, which does not require to trust the compiler. The bytecode verifier performs the 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 [**?**, **?**, **?**] are another

alternative. In this architecture, the 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 verification framework with the following features:

- translating source program annotation into bytecode annotation. Thus bytecode can benefit from the source specification and does not need to be accompanied by its source code. Moreover, this approach is suitable for scenarios where the consumer requirements are potentially complex and a full automatic specification inference will fail.

- establishing the correctness of Java bytecode programs using a verification condition generator over Java bytecode, which supports the bytecode annotation.

The scheme can potentially be used in a PCC framework. In particular, we aim at compiling source proofs into bytecode proofs.

In particular, our approach is tailored to Java bytecode. The Java technology finds a large application in 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. In this article we propose a static verification technique using formal methods for sequential Java bytecode programs.

The aforementioned scheme is composed by several components. 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 like exceptions, references, method calls and subroutines. We define a bytecode specification language, called BCSL, and supply a compiler from the high level Java specification language JML [**?**] 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, thus making not only the code mobile but also its specification. These class file extensions do not affect the JVM performance. The scheme makes the Java bytecode benefit from the specification written at source level. 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 (JACK) [**?**]. The full specifications of the JML compiler and the weakest precondition predicate transformer definition can be found in [**?**] and [**?**] respectively.

The remainder of the paper is organized as follows: Section 2 reviews scenarios in which the architecture is appropriate to use; Section 4 presents the bytecode specification language BCSL and the JML compiler; Section 5 explains how the weakest precondition calculus works illustrating it with definitions and example; in this section we also give the verification conditions that are generated for proving program correctness; Section 6 concludes with future work.

2 Framework

Figure 1 presents the proposed overall architecture for ensuring Java bytecode correctness.

It describes a process that allows a client to trust a code produced by an untrusted code producer. This approach is especially suitable in cases where the client policy involves non trivial functional or safety requirements and thus, an automatic specification inference cannot be applied.

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. In this case, 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 [**?**].

In the development process, the producer verifies if the client requirements are respected by generating verification conditions over the source code and usually, he has to add JML annotations for this e.g. loop invariants, class invariants, method preconditions and postconditions etc. It is usually only after specifying enough the source code that the annotated Java source and class files are fed to the JML compiler.

If the annotations are sufficient to prove the code, the Java file is normally compiled with a Java compiler to obtain a class file. This class file is then extended with user defined attributes that contain the BCSL specification, resulting from the compilation of the JML specification in the Java

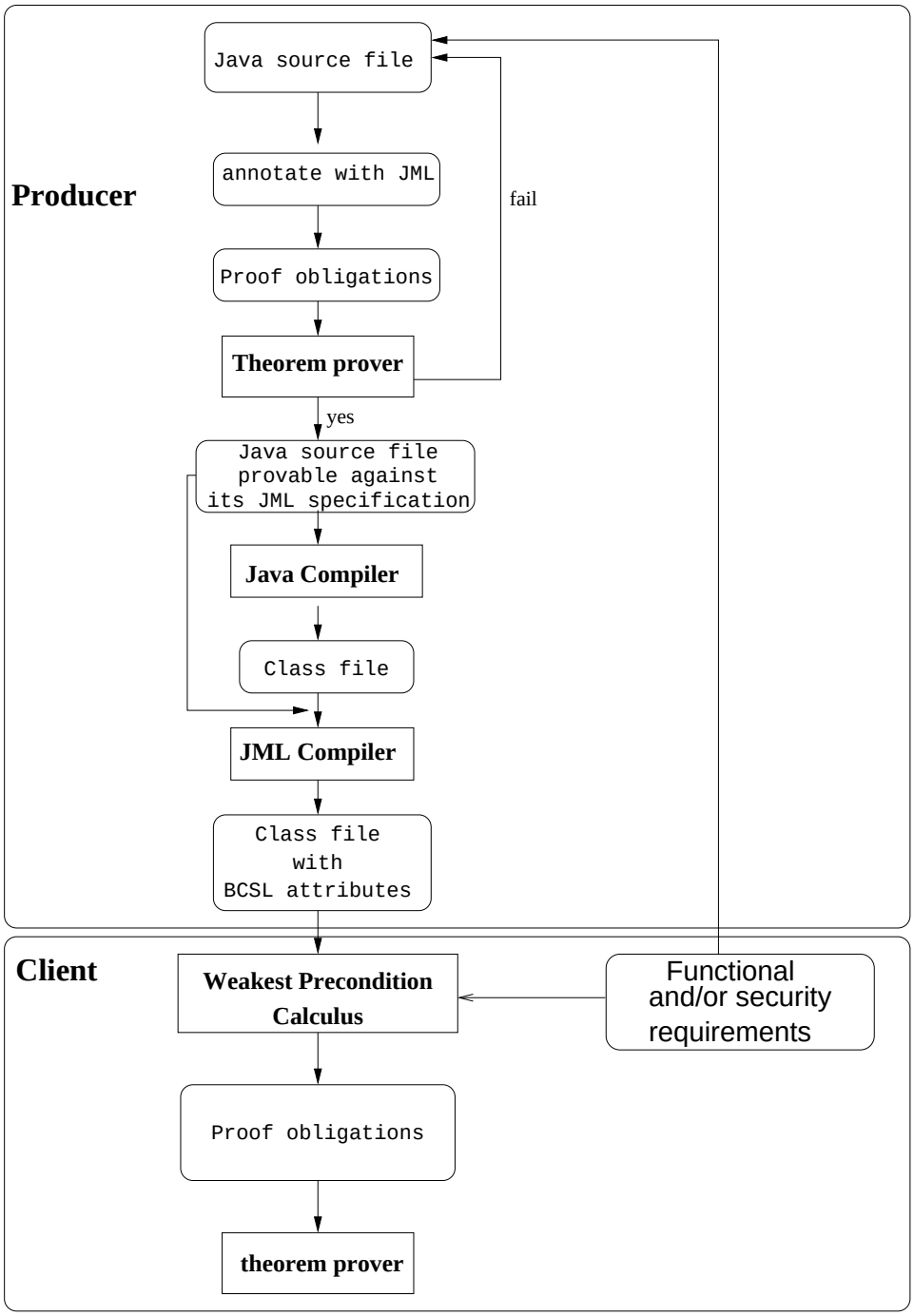

Figure 1: The overall architecture for annotating and verifying code

source file. At this stage, the Java class files contain all the information that will allow the client to check if the bytecode does not violate his requirements. In particular, the client will generate proof obligations from the untrusted annotated bytecode and his security requirements (expressed in a suitable form) as shown in figure 1. Proof obligations are formulas which, if provable, guarantee the bytecode correctness. The latter are then proved with a theorem prover (possibly interactively). If the client succeeds in proving the verification conditions, he can trust the unknown code.

Currently the framework does not support sending both the proof and the bytecode to the client this being the next step in our work.

To implement this architecture we use JACK 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 following we introduce the BCSL language, the JML compiler and the bytecode weakest precondition calculus which underlines the bytecode verification condition generator.

3 Related Work

There are several fields which are related to the present work: bytecode verification, logic for unstructured program languages, attaching specification to the compiled code, architecture for checking untrusted components.

Bytecode verification is concerned with establishing that a bytecode is well typed (every instruction is applied to operands of the correct type) and well formed (e.g. no jumps to an un-existing bytecode index), differently from the goals of the present work where program correctness is defined in terms of functional correctness. The Java Virtual Machine (JVM), for example, is provided with a bytecode verifier. The field is well researched and for more information one can look at [**?**].

Floyd is among the first to work on program verification for unstructured languages (see [**?**]). Few works have been dedicated to the definition of a bytecode logic. In [**?**], Quigley defines a Hoare logics for bytecode programs. This work is limited to a subset of the Java virtual machine instructions and does not treat for example method calls, neither exceptional termination. The logic is defined by searching a structure in the bytecode control flow graph, which gives an issue to complex rules.

A work close to ours is presented in [**?**] by P.Muller and F.Bannwart. The authors define a Hoare logic over a bytecode language with objects

[1] http://www.eclipse.org

and exceptions. A compiler from source proofs into bytecode proofs is also defined. As in our work, they assume that the bytecode has passed the bytecode verification certification. The bytecode logic aims to express functional properties. To our knowledge subroutines are not treated. Invariants are inferred by fixpoint calculation, differently from the approach presented here, where invariants are compiled from the high level JML specification (see section 4.2). However, inferring invariants is not a decidable problem. The work of P.Muller and F.Bannwart is inspired by the Nick Benton's work (see [**?**]). In the latter a bytecode logic for a stack based language is defined which checks programs both for well — typedness and functional correctness. The language does not support objects, references, exceptions neither subroutines.

In [**?**], M. Wildmoser and T. Nipkow describe a framework for verifying Jinja (a Java subset) bytecode against arithmetic overflow. The annotation is written manually, which is not comfortable, especially on bytecode. Here we propose a way to compile a specification written in a high level language, allowing specification to be written at source level, which we consider as more convenient.

The Spec# ([**?**]) programming system developed at Microsoft proposes a static verification framework where the method and class contracts (pre, post conditions, exceptional postconditions, class invariants) are inserted in the intermediate code . Spec# is a superset of the C# programming language, with a built-in specification language, which proposes a verification framework (there is a choice to perform the checks either at runtime or statically). The static verification procedure involves translation of the contract specification into metadata which is attached to the intermediate code and the verification is performed over the bytecode by the Boogie theorem prover.

Another topic related to the present work is PCC. PCC and the certifying compiler were proposed by Necula (see [**?**, **?**, **?**]). PCC is an architecture for establishing trust in untrusted code in which the code producer supplies a proof for correctness with the code. The initial idea for PCC was that the producer automatically infers annotation for properties like well typedness, correct read/writes and automatically generates the proof for their correctness using the certifying compiler. Such properties guarantee that a program do the things correctly and not that it does the right things. The present work is targeting at a framework for establishing complex functional and interface properties whose automatic checking is hard and even impossible. From the above cited papers, [**?**] is aiming also at building PCC for guaranteeing not trivial properties. As we stated in section 1, our framework currently does not support adding proofs to bytecode which but we consider this point as a future work.

4 Bytecode Specification Language (BCSL)

In this section, we itroduce a bytecode specification language which we call BCSL. We define a compiler from the high level specification language JML to BCSL. The specification compilation results in a class file extension. In the following we give the grammar of BCSL and sketch the specification compiler.

4.1 Grammar

BCSL (short for ByteCode Specification Language) is based on the design principles of JML (Java Modeling Language), i.e. it is a behaviorial interface specification language and follows the design by contract approach, (see [**?**]).

In order to give a flavour of what JML specifications look like, in Fig. 2 we show 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 basically says that whenever the loop entry is reached the elements inspected already by the loop are all different from `obj`.

BCSL corresponds to a representative subset of JML. We sketch the bytecode specification language grammar in Fig. 3. We omit some of the definitions because of space constraints, e.g. the grammar for arithmetic expressions (which is defined in a standard way).

The language defined here is expressive enough for most purposes including the description of non trivial functional and security properties. We now discuss some of the specification clauses, for the rest their semantics is the same as in JML and can be found in [**?**, **?**].

As the grammar shows, BCSL supports like JML:

- class specification, e.g class invariants
- method pre and postconditions (normal and exceptional) and method frame conditions (the locations that may be modified by the method). We also support behavioral subtyping by specification inheritance (the keyword also)
- inter method specification, for instance loop invariants. One can notice that the loop invariant specification are tagged with the index of

```
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

the instruction at which the loop invariant must hold (the loop entry instruction). this is different from JML where loop invariants are written at the beginning of the declaration of the loop statement, while the BCSL specification are separated from the bytecode

- predicates from first order logic
- expressions from the programming language, like field access expressions (`field_cp_index(`$\mathcal{E}$`)`), local variables (`lv[i]`), 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 rather inspired for this by [**?**].
- stack expressions - `c` which stands for the stack counter and `st(Arithmetic_Expr)` standing for a stack element at position `Arithmetic_Expr`. These ex-

ClassSpec	::= class invariant $\mathcal{P}$ | history constraint $\mathcal{P}$ | model `ClassName` *id*
MethodSpec ::=	SpecCase | SpecCase also MethodSpec;
SpecCase	::= requires $\mathcal{P}$; modifies $list(\mathcal{E})$; ensures $\mathcal{P}$; exsures (`ExceptionClass`) $\mathcal{P}$;
InterMethodSpec	::= loopSpec | assertSpec
loopSpec	::= pc_ index *int*; loop_modifies $list(\mathcal{E})$; loop_invariant $\mathcal{P}$; loop_decreases $\mathcal{E}$;
assertSpec	::= pc_ index *int*; assert $\mathcal{P}$;
$\mathcal{P}$	::= `true` | `false` | $\mathcal{E}$ *predSymbol* $\mathcal{E}$ | $\mathcal{P} \wedge \mathcal{P}$ | $\mathcal{P} \vee \mathcal{P}$ | $\mathcal{P} \Rightarrow \mathcal{P}$ | $\forall$(`boundVar` : *JavaType*)$\mathcal{P}$ | $\exists$(`boundVar` : *JavaType*)$\mathcal{P}$
$\mathcal{E}$	::= `Arithmetic_Expr` | `lv[i]` | `ref` | `int_literal` | `field_cp_index(`$\mathcal{E}$`)` | $\mathcal{E}[\mathcal{E}]$ | `\result` | `\old(`$\mathcal{E}$`)` | `EXC` | `\typeof(`$\mathcal{E}$`)` | `null` | `this` | `c` | `st(Arithmetic_Expr)` ...

Figure 3: BCSL grammar

pressions are needed in BCSL as the Java bytecode language is stack based. They do not appear in the specification. Later we shall see how they are used.

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) [**?**] 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([**?**], 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. compile 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 [**?**], 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. from the source file and the resulting class file compile the JML specification. 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, such 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 source program point must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. Basically 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

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

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

Figure 4: The compilation of the postcondition in Fig. 2

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 a variable in a Java source can have boolean type but 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 contains an equivalence between the JML expression `\result` and a predicate. As the method `isElem` is declared with return type boolean the expression `\result` has type boolean on source level. Still, the bytecode resulting from the compilation of the method `isElem` has type integer. This means that the compiler method has to make more effort than simply compiling the left and right side of the equivalence in the postcondition, otherwise the postcondition will not make sense (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` in is given in Fig. 4.

From the postcondition compilation, one can see that `\result` is compiled to integer and the equivalence 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`, `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

[3] when generating proof obligations we add for every source boolean expression an assumption that it must be equal to 0 or 1. This, actually, must be always equal to true as we assume that the programs are well typed

newly defined attributes in the class file. For example, the specification of all the loops in a method are compiled to a unique method attribute: whose syntax is given in figure 5. This attribute is an array of data structures each describing a single 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 invariant associated to this loop, the decreasing expression in case of total correctness, the expressions that can be modified. For the full specification of the compiler see [**?**].

```
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 modified expressions.
- **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 5: Structure of the Loop Attribute

The JML compiler does not depend on any specific Java compiler, but it requires the presence of a debug information, namely the presence of the **Line_Number_Table** attribute for the proper 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 find the program points where the loop invariants must hold. This basically means that one has 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 [**?**]); intuitively this means 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 correctly the invariants to the proper 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, thus the calculus is concerned only with program functional properties. We also assume that code is generated by a non optimizing compiler.

The proposed weakest precondition has those features:

- it supports all Java 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
- it supports BCSL annotations (section 4), i.e. bytecode method's specification like pre- and postconditions, class invariants, assertions at particular program point among which loop invariants (if there is no explicite specification the default one is taken into account: preconditions, postconditions and invariants are taken to be true, exceptional postcondition is false) is taken into account.

The calculus is defined over the control flow graph of the program and has two levels of definitions — the first one is the set of rules for sequential Java bytecode instructions (discussed in subsection 5.1) and the second one takes into account how control flows in the bytecode (subsection 5.3). Loops are treated by transforming the control flow graph into an abstract acyclic graph (by eliminating the backedges). As we mentioned earlier we assume that every method is specified enough, i.e. for each loop, the corresponding invariant is present. Thus, the eliminated edges are replaced by the corresponding loop invariant. The verification conditions are generated over the abstract control flow graph. Subsection 5.2 discusses how the abstract control flow graph is generated.

We have the proof of soundness of the *wp* predicate transformer. The proof is done w.r.t. to the operational semantics of the sequential Java bytecode subset and establishes that if the verification conditions are provable this means that the method implementation respects the method specification.

5.1 Weakest Precondition for bytecode instructions

We define a weakest precondition (*wp*) predicate transformer function which takes into account normal and exceptional termination. *wp* takes three arguments: an instruction, a predicate that is the instruction's normal postcondition ψ, and a function from exception types to predicates ψ^{exc} (it returns the specified postcondition in the `exsures` clause for a given exception; see section 3). The *wp* function returns the weakest predicate such that if it holds in the state when the instruction starts its execution the following conditions are met:

1. if the instruction terminates its execution normally the predicate ψ holds in the poststate
2. if it terminates with an exception `E` then the predicate ψ^{exc}(`E`) holds in the poststate.

The Java bytecode language is stack based, i.e. the instructions take their arguments from the method execution stack and put the result on the stack. In Fig. 6 we show the *wp* rules for some bytecode instructions where the expression `c` and `st(c)` stand respectively for the the stack counter and the element on the top of 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]`.

The rules also take into account the possible abnormal execution of the instruction. For example, in Fig. 6, the rule for the instruction `putField` has two conjuncts - one in the case when the dereferenced object is not null and the instruction execution terminates normally; the other one stands for the case when this is not true. Note, that if the exception thrown is not handled, we substitute the special specification variable `EXC` (see Subsection 4.1) in the exceptional postcondition by the thrown exception object.

5.1.1 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 [**?**]. In Fig. 6 the rule for `putField` substitutes the corresponding field function `Cl.f` with `Cl.f` updated for object o, in case the dereferenced object is not null. The definition of update function is given in figure 7.

5.1.2 Method calls

Method calls are handled by using their specification. A method specification is a contract between callers and callees — the precondition of the

$wp(\texttt{Type\_load}\ i,\ \psi,\ \psi^{exc}) =$
$\psi[\texttt{c} \leftarrow \texttt{c}+1][\texttt{st(c+1)} \leftarrow \texttt{lv[i]}]$
where i is a valid local variable index

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

where the predicate ϕ is the precondition of the exception handler protecting the instruction against `NullPointerException` *if it exists, otherwise if the* `NullPointerException` *is not handled*
$\phi = \psi^{exc}(\texttt{NullPointerException})[\texttt{EXC} \leftarrow \texttt{st(c)}]$

Figure 6: rules for some bytecode instructions

$$(\texttt{Cl.f}) \oplus [e2 \rightarrow e1](o) = \begin{cases} e1 & if\ e2 = o \\ \texttt{Cl.f}(o) & else \end{cases}$$

Figure 7: Overriding Function

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 on a non-void instance method is given in figure 8. 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.

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

$\psi^{pre}(\texttt{m})$ – *the specified precondition of method* m

$\psi^{post}(\texttt{m})$ – *the specified postcondition of method* m

$\psi_{\texttt{m}}^{exc}$ – *the exceptional function for method* m

$\texttt{numArgs}(\texttt{m})$ – *the number of arguments of* m

$e_j, j = 1..s$ – *the locations modified by method* m

$\texttt{Exc}_\texttt{i}, \texttt{i} = \texttt{1..k}$ – *the exceptions that* m *may throw*

$\phi_{\texttt{Exc}_\texttt{i}}, i = 1..k$ – *is the precondition of the exception handler protecting the instruction against* $\texttt{Exc}_\texttt{i}$ *if it exists, otherwiseif the* $\texttt{Exc}_\texttt{i}$ *is not handled*
$\phi = \psi^{exc}(\texttt{Exc}_\texttt{i})[\texttt{EXC} \leftarrow \texttt{st(c)}]$

Figure 8: *wp* rule for a call to an instance non void method

5.2 Abstracting The Control Flow Graph

In this section we discuss how the control flow graph of a bytecode is transformed in an acyclic control flow graph. We assume that the bytecode is provided with sufficient specification and in particular loop invariants. Under this assumption, we can "cut" the control flow graph at every program point where an invariant must hold and "place" at that point the invariant. This requires the introduction of several definitions.

A method body is an array of bytecode instructions. We write i_k the $k-th$ instruction in a method body. We assume that method's bytecode has exactly one entry point(every execution of a method starts at the entry point instruction) and we denote it with $i_{\texttt{entry}}$. Using standard terminology (see [**?**]), a basic block is a code segment that has no unconditional jump or conditional branch statements except for possibly the last statement, and none of its statements, except possibly the first, is a target of any jump or

branch statement. We denote a block starting at instruction i_j with $\mathtt{b}^{\mathtt{j}}$. The block starting at the entry instruction is denoted with $\mathtt{b}^{\mathtt{entry}}$. The execution relation $\mathtt{b}^{\mathtt{j}} \rightarrow \mathtt{b}^{\mathtt{k}}$ states that block $\mathtt{b}^{\mathtt{k}}$ may be executed immediately after $\mathtt{b}^{\mathtt{j}}$ in some execution path of the method. For example if instruction $\mathtt{i}_{\mathtt{k}} = \mathtt{athrow}$ is the last of the block $\mathtt{b}^{\mathtt{j}}$ then $\mathtt{b}^{\mathtt{j}} \rightarrow \mathtt{b}^{\mathtt{n}}$, where $\mathtt{b}^{\mathtt{n}}$ is the first block of an exception handler that protects $\mathtt{i}_{\mathtt{k}}$

Definition 1 *Loop Assume we have a bytecode* Π. *We say that* $\mathtt{b}^{\mathtt{e}}$ *is the entry block of a loop* l *in* Π *and* $\mathtt{b}^{\mathtt{f}}$ *is an end block of* l *and we note this with* $\mathtt{b}^{\mathtt{f}} \rightarrow^{\mathtt{l}} \mathtt{b}^{\mathtt{e}}$ *if:*

- *every path in the control flow graph starting at the entry block* $\mathtt{b}^{\mathtt{entry}}$ *of* Π *and that reaches* $\mathtt{b}^{\mathtt{f}}$*, passes through* $\mathtt{b}^{\mathtt{e}}$
- *there is a path in which* $\mathtt{b}^{\mathtt{e}}$ *is executed immediately after the execution of* $\mathtt{b}^{\mathtt{f}}$*,* $\mathtt{b}^{\mathtt{f}} \rightarrow \mathtt{b}^{\mathtt{e}}$

We abstract the execution relation $\rightarrow$ to the acyclic execution relation $\rightarrow^A$ which is an abstraction of $\rightarrow$ where the backedges $\rightarrow^l$ are removed: $\rightarrow^A = \rightarrow \setminus \rightarrow^l$. The weakest precondition is applied over the resulting abstraction of the control flow graph, where the eliminated edges are replaced by the appropriate loop invariant.

5.3 Bytecode Weakest Precondition

In the following we overload the function symbol *wp* applying it to a method and sequence of bytecode instructions. The *wp* function runs in a backwards direction starting from the blocks that do not have successors up to reaching the entry point instruction. We distinguish the *wp* calculus over a method's body bytecode, a bytecode block and a sequence of bytecode instructions.

The weakest precondition $wp(\mathtt{m})$ *for method* $\mathtt{m}$ is the weakest precondition of its entry block $\mathtt{b}^{\mathtt{entry}}$.

$$wp(\mathtt{m}) = wp(\mathtt{b}^{\mathtt{entry}})$$

The weakest precondition for a bytecode block is calculated splitting the block in two parts: its last instruction and its sequential part (the instruction sequence without the last one). We note with $\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}$ the sequential part of block $\mathtt{b}^{\mathtt{i}}$ and $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}})$ the predicate that must hold after the execution of $\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}$ and before its last instruction; the weakest precondition of $\mathtt{b}^{\mathtt{i}}$ is then defined as follows:

$$wp(\mathtt{b}^{\mathtt{i}}) = wp(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}, post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}), \psi^{exc})$$

Concerning the sequential part, the *wp* is calculated applying the standard *wp* rule for compositional statements :

$$wp(instrList; i_j, \psi, \psi^{exc}) =$$
$$wp(instrList, wp(i_j, \psi, \psi^{exc}), \psi^{exc})$$

The postcondition $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}})$ of the block $\mathtt{b}^{\mathtt{i}}$ depends on its last instruction and the respective predicates that must hold between $\mathtt{b}^{\mathtt{i}}$ and its successor blocks. Those predicates are determined by the function *pre*, which for any two blocks $\mathtt{b}^{\mathtt{i}}$ and $\mathtt{b}^{\mathtt{n}}$, such that $\mathtt{b}^{\mathtt{i}} \rightarrow \mathtt{b}^{\mathtt{n}}$ gives the predicate $pre(b^i, b^n)$ that must hold after the execution of $\mathtt{b}^{\mathtt{i}}$ and before the execution of $\mathtt{b}^{\mathtt{n}}$.

We define the postcondition of the sequential part of a block $\mathtt{b}^{\mathtt{i}}$ as follows:

Definition 1 *Block's postcondition* $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}})$. *Let* i_s *be the last instruction of* $\mathtt{b}^{\mathtt{i}}$ *then:*

- *if* i_s = `if_cond n`

$$post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) = \begin{cases} cond(\mathtt{st(c)}, \mathtt{st(c-\ 1)}) \Rightarrow \\ \quad pre(b^i, b^n)[\mathtt{c} \leftarrow \mathtt{c} - 2] \\ \wedge \\ not(cond(\mathtt{st(c)}, \mathtt{st(c-\ 1)})) \Rightarrow \\ \quad pre(b^i, b^{s+1})[\mathtt{c} \leftarrow \mathtt{c} - 2] \end{cases}$$

- *if* i_s = `goto n`
  $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) = pre(b^i, b^n)$

- *if* i_s = `athrow`

  1. *if there exists a block* $\mathtt{b}^{\mathtt{e}}$ *such that* $\mathtt{b}^{\mathtt{i}} \rightarrow \mathtt{b}^{\mathtt{e}}$ *(i.e. an exception handler protects the type of the exception thrown) then :*
     $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) =$
     $pre(b^i, b^e)[\mathtt{c} \leftarrow 0][\mathtt{st(0)} \leftarrow \mathtt{st(c)}]$.
  2. *Otherwise the thrown exception is not handled and then* $\mathtt{b}^{\mathtt{i}}$ *must respect the postcondition determined by the exceptional postcondition function* ψ^{exc} *for this exceptional type:*
     $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) =$
     $\psi^{exc}(\mathtt{st(c)})[\mathtt{c} \leftarrow 0][\mathtt{st(0)} \leftarrow \mathtt{st(c)}][\mathtt{EXC} \leftarrow \mathtt{st(c)}]$.
     see in section 4.1 for the meaning of `EXC`.

- *if* i_s = `return`

$$post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) = \psi[\backslash \mathtt{result} \leftarrow \mathtt{st(c)}]$$

  where ψ *is the specified method postcondition.*

- *else*

  $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) =$
  $wp(i_s, pre(b^i, b^{s+1}), \psi^{exc})$

The function *pre* determines what is the property that should hold between two blocks that execute one after another, depending on if they determine a cycle or not (see definition 1 in the previous Section 5.2). This definition gives us the right to abstract the control flow to an acyclic one, as discussed in the previous subsection 5.2 and perform on the latter the weakest precondition calculus.

Definition 1 *Predicate between consecutive blocks. Assume that* $b^i \rightarrow b^n$. *The predicate* $pre(b^i, b^n)$ *must hold after the execution of* $\mathtt{b^i}$ *and before the execution of* $\mathtt{b^n}$ *and is defined as follows:*

- *if* $\mathtt{b^i} \rightarrow^{\mathtt{l}} \mathtt{b^n}$ *then the corresponding loop invariant must hold:*

$$pre(b^i, b^n) = I$$

- *else if* $\mathtt{b^n}$ *is a loop entry then the corresponding loop invariant I must hold before* $\mathtt{b^n}$ *is executed, i.e. after the execution of* $\mathtt{b^i}$. *We also require that I implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations* $m_i, i = 1..s$ *that may be modified in the loop.*

$$pre(b^i, b^n) = I \ \wedge \ \forall_{i=1..s} m_i.(I \Rightarrow wp(b^n))$$

- *else the normal precondition is taken into account:*

$$pre(b^i, b^n) = wp(b^n)$$

Subroutines are treated by in-lining. Exception handlers are treated by identifying the instructions that belong to the handler (the class file format provides information about the exception handlers for all methods, in particular where starts and ends the region they protect and at what index the handler starts at); the precondition of the handler bytecode is calculated upon the normal postcondition of the method.

5.4 Verifying Java Bytecode Programs

Bytecode programs represent a set of Java classes. Establishing the correctness of Java bytecode program w.r.t. to their specification thus, consists of generating verification conditions for every method appearing in every class of the bytecode program. The verification procedure for a method $\mathtt{m}$ consists of calculating the weakest precondition $wp(\mathtt{m})$ upon its specification: precondition $\psi^{pre}(m)$, postcondition $\psi^{post}(\mathtt{m})$ and the mapping between exceptional types and predicates $\psi^{exc}(\mathtt{m})$ and then prove the condition:

$$\psi^{pre}(m) \Rightarrow wp(m)$$

The verification procedure does not trust neither the bytecode specification, nor the bytecode; in both cases — wrong specification or incorrect implementation will result in verification conditions that are not provable

5.5 Results

We have an implementation of the JML compiler (subsection 4.2) and the bytecode verification condition generator based on the weakest precondition calculus (Section 5) which are integrated in JACK. Both of the verification condition generators perform the same simplifications over the verification conditions , e.g. eliminate verification conditions that contain contradictory hypothesis or trivial goals (equal to true).

The performed tests show that JML compilation augments around twice the file size. For the example in Fig. 2, the class file without the specification extensions is 548 bytes, and the class with the BCSL extension BCSL is 954 bytes. The size of the bytecode specification is proportional to the source specification: the bigger is the source specification, the greater will be the size of the class file.

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. Basically, they are the same modulo the names of the program variables. Another issue that deserves the attention is that even non optimizing compilers perform optimizations, as for example dead code elimination of a conditional branch which is never taken. 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 contains 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.

We return now to our example from the previous sections and give the proof obligations on source and bytecode level respectively. In particular, in Fig. 9 compares the source and bytecode proof obligation concerning the postcondition correctness (as there are several return instructions).

The verification conditions on bytecode and source level for the postcondition correctness given in Fig. 9 have the same shape modulo names (see Section 4.2 for how method local variables and field names are compiled). In Section 4.2 we showed that postcondition in the example was compiled by performing structural transformations in an equivalent specification expression and we gave it in Fig. 4.

Despite those changes, 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).

One of the future directions is to formally give evidence that the proof

obligations on non optimized bytecode and source programs are syntactically the same (modulo names and types).

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 9: Source and Bytecode verification condition for one case of the postcondition correctness

6 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[4]. 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 are now aiming to complete our architecture for establishing trust in untrusted code - in particular extending the present work to a PCC architecture for establishing non trivial requirements. In this way, several important directions for future work are:

- perform case studies and strengthen the tool with more experiments.
- find an efficient representation and validation of proofs in order to construct a PCC framework for Java bytecode. We would like to build

[4]http://www-sop.inria.fr/everest/soft/Jack/jack.html

a PCC framework where the proofs are done interactively over the source code and then compiled down to bytecode. Actually, as we stated in Section 5.5 the proof obligations generated over a source program and over its compilation with non optimizing compiler are syntactically equivalent modulo name and types.

- an extension of the framework applying previous research results in automated annotation generation for Java bytecode (see [**?**]). 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.