Java Bytecode Specification and Verification

Lilian Burdy
INRIA Sophia-Antipolis
2004, Route des Lucioles, BP 93,
06902 Sophia-Antipolis, France
Lilian.Burdy@sophia.inria.fr

Mariela Pavlova
INRIA Sophia-Antipolis
2004, Route des Lucioles, BP 93,
06902 Sophia-Antipolis, France
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 [9] 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 [11] 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 components: a bytecode specification language, a compiler from source program annotations into bytecode annotations and a verification condition generator over Java bytecode.

In a client-producer scenario, these features bring 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 client is supplied with a procedure 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 [7] 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. 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) [4].

The full specifications of the JML compiler, the weakest precondition predicate transformer definition and its proof of correctness can be found in [12].

The remainder of the paper is organized as follows: Section 2 reviews scenarios in which the architecture may be appropriate to use; Section 3 presents the bytecode specification language BCSL and the JML compiler; Section 4

SAC'06 April 23-27, 2006, Dijon, France

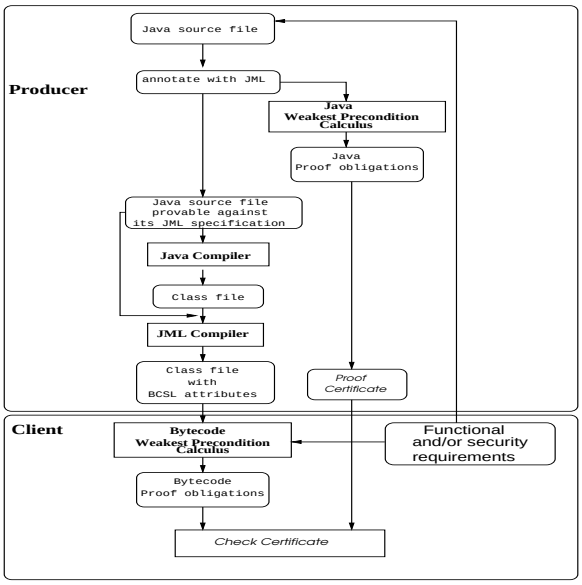

Figure 1: The overall architecture for client producer scenarios

discusses the main features of the *wp*(short for weakest precondition calculus); Section 5 discusses the relationship between the verification conditions for JML annotated source and BCSL annotated bytecode; Section 6 concludes with future work.

2. APPLICATIONS

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, a full automatization of the verification process is impossible. 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.

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:

- Typical functional requirements can be a specified interface describing the application to be developed. In that case, the client specifies in JML the features that have to be implemented by the code producer.

- Client security requirements can be a restricted access to some method from the API expressed as a finite state machine. 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 [13].

Usually, 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 annotated class files. 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 [4] 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 *wp*calculus which underlines the bytecode verification condition generator.

3. BYTECODE SPECIFICATION LANGUAGE

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) [7], which is a behavioral interface specification language following the design by contract approach [2].

Before going further, we give a flavor 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. As the figure shows, JML annotations are written in comments and thus 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 the method precondition (introduced by `requires`) and that the method return value will be true if and only if the internal array `list` contains the object referenced by the argument `obj` in the method 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`.

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

3.1 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

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

```
public class ListArray {
  Object[] list;
  //@requires list != null;
  //@ensures \result == (\exists int i;0<=i &&
            i < list.length && list[i]==obj);
  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

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), inter method specification, as for instance loop invariants and loop frame conditions(this is not a standard feature of JML but we were inspired for this by the JML extensions in JACK [4]). We also support specification inheritance and thus behavioral subtyping as described in [6]. Most of the Java expressions like field access expressions, local variables, etc can be mentioned in the BCSL specification. BCSL supports the standard JML specification operators as for example, `\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, `\typeof(`$\mathcal{E}$`)` which stands for type of $\mathcal{E}$ etc.

3.2 Compiling JML into BCSL

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 (JVM) [10] 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 JVM allows to add to the class file user specific information([10], ch.4.7.1). This is done by defining user specific attributes (their structure is predefined by JVM).

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 [10], 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]not to be confused, Gary Leavens also calls his tool jmlc JML compiler, which transforms JML into runtime checks and thus generates input for the jmlrac tool

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

Figure 3: THE COMPILATION OF THE POSTCONDITION OF METHOD ISELEM

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 cp (short for constant pool) or the array of local variables described in the **Local_Variable_Table** attribute. If a field identifier, for which no cp index exists, appears in the JML specification, a new index is added in the cp and the field 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 interesting point in this stage of the JML compilation is how the type differences on source and bytecode level are treated. The JVM does not provide a direct support for integral types like byte, short, char, neither for boolean. Those types are rather encoded as integers in the bytecode. The JML compiler performs transformation on specifications that involve Java boolean values and variables. We illustrate this by an example in Fig. 3, which shows the resulting compilation of the postcondition of method `isElem` in Fig. 2. 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 cp 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 compiled specifications components in 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 single loop from the method source code. 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,

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

Figure 4: STRUCTURE OF THE LOOP SPECIFICATION ATTRIBUTE

the invariant associated to this loop and the decreasing expression in case of total correctness,

The most problematic part of the specification compilation is the identification of which loop in the source 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 into the middle of the loops from outside; graph reducibility allows to establish the same order between loops in the bytecode and source code level and to find the right places in the bytecode where the loop invariants must hold.

4. WEAKEST PRECONDITION CALCULUS FOR JAVA BYTECODE

In this section, we define a bytecode logic in terms of a weakest precondition calculus. 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. The verification condition generator applied to a method bytecode generates a proof obligation for every execution path by applying first the weakest predicate transformer to every `return` instruction, `athrow` instruction and end of a loop instruction and then following in a backwards direction the control flow up to reaching the entry point instruction. In an extended version of the present paper [12], we show that the *wp* function is correct.

In Fig. 5, we show the *wp* rule for the `Type_load i` instruction. As the example shows 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` and instruction index `ind` returns the corresponding exceptional postcondition $\psi^{exc}(\texttt{Exc}, \texttt{ind})$. One can also notice that the rule involves the stack expressions `ct`(stands for the counter of the method execution stack) and `st( i )` (stands for the element at ind `i` from the stack top). This is because the JVM is stack based and 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 `ct` and loads on the stack top the

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

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

Figure 5: EXAMPLES FOR BYTECODE WP RULES

contents of the local variable `lv[i]`.

In the following, we consider how instance fields, loops exception handling and subroutines are treated. We omit here aspects like method invocation and object creation because of space limitations but a detailed explanation can be found in [12].

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 updates when assigning a value to a field reference as, for instance in [3]. In Fig.5, we give the *wp* rule for the instruction `putfield Cl.f`, which updates the field `Cl.f`[3] of the object referenced by the reference stored in the stack below the stack top `st(ct-1)` with the value on the stack top `st(ct)`. Note that the rule takes in account the possible exceptional termination of the instruction execution.

Loops. Identifying 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 analyze the control flow graph in order to find them. The analysis consists in looking for the backedges in the control flow graph using standard techniques, see [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 graph.

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 the 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 ex-

[3] `Cl.f` stands for the field `f` declared in class `Cl`

ception 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} 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[4]. First, the instructions of every subroutine are identified. To this end, we assume that the code has passed the bytecode verification and that every subroutine terminates with a `ret` instruction(usually, the compilation of subroutines ends with a `ret` instruction but it is not always the case). Thus, by abstract inlining, we mean that whenever the *wp* function is applied to an instruction `jsr ind`, a postcondition ψ and an exceptional postcondition function ψ^{exc}, its precondition $wp^{\texttt{jsr ind}}$ is calculated as follows: the *wp* is applied to the bytecode instructions that represent the subroutine which starts at instruction `ind`, the postcondition ψ and the exceptional postcondition function ψ^{exc}.

5. RELATION BETWEEN VERIFICATION CONDITIONS ON SOURCE AND BYTECODE LEVEL

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 [8]. The proof obligations were the same modulo names and short, byte and boolean values as well as hypothesis names. The proof obligations on bytecode and source level that we proved interactively in Coq produced proof scripts which were also equal modulo the names of hypothesis. This means that if an appropriate encoding of proof obligations on source and bytecode level is found, where the names of the source and bytecode hypothesis are the same, the produced proof script for a source proof obligation can be applied to the corresponding one on bytecode.

The equivalence between source and bytecode proof obligations can be applied to PCC scenarios, as discussed in Section 1 in cases where the client policy is complex and a complete automatic certification will not work and the producer has to generate the program certificate interactively.

6. CONCLUSION AND FUTURE WORK

This article describes the bytecode specification language BCSL, a compiler from the JML language to BCSL and a bytecode *wp* calculus. A proof obligation generator based on the wp calculus and a JML compiler to BCSL have been implemented and are part of the Jack 1.8 release[5]. Those components have been applied to Java to native code optimization of realistic examples [5].

We conclude with future work directions. Currently, we have a framework for Java program verification which is the first step towards a PCC infrastructure. First, we aim to establish formally that non optimizing source compilation preserves proof obligations modulo names and basic types. This equivalence can be used to complete the PCC framework where the proof certificates will be made interactively on source level. Second, we would like to perform more real case studies. Finally, we are interested in the extension of the framework applying previous research results in automated annotation generation for Java source programs (see [13]). The client thus will establish that the code respects his security policy by first propagating automatically annotations in the loaded code and then verifying the resulting annotated code.

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

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

7. REFERENCES

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

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

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

[4] 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: ISFME*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[5] A. Courbot, M. Pavlova, G. Grimaud, and J.-J. Vandewalle. A low-footprint java-to-native compilation scheme using formal methods. submitted to Cardis06.

[6] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267, 1996.

[7] 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`.

[8] 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*, lncs, pages 202–219. Springer, Berlin, 2003.

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

[10] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] 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.

[12] M. Pavlova. Java bytecode logic and specification. Preliminary version. Available from `http://www.inria.fr/everest/Mariela.Pavlova`.

[13] 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.