

Supplementing Java Bytecode with Specifications

Jędrzej Fulara and Krzysztof Jakubczyk and Aleksy Schubert

Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw, Poland

Abstract. The proof-carrying code (PCC) techniques allow the executable code to be augmented with a proof that the code obeys certain policies (e.g. the program does not store passwords in clear in a file). BML (Bytecode Modelling Language) can be regarded as a part of the PCC architecture which allows to express detailed properties of Java bytecode programs. As most of the programming is done at the source code level, it is desirable to have a way to translate properties expressed at the source code level (in our case written in Java Modeling Language, JML) to the bytecode level. In this paper we present a *JML2BML* compiler, a tool that for a given Java source file annotated with specifications generates class files with BML.

1 Introduction

The bytecode verification process performed by Java Virtual Machine (JVM) ensures vital properties of programs such as that all operand arguments on the operand stack are correct legal, there are enough arguments on the stack etc. In certain situations, these guarantees are not sufficient. In particular, many of the security guarantees are ensured at runtime. For instance, a mobile application asks the user to acknowledge the sending of data over the mobile network. However, the user may get bored with many such prompts and disable this security feature. After this, she or he may easily be made to send data to e.g. an expensive premium number. It would be more secure to give the user a load- (or download-) time guarantee that the code sends data only to expected receivers. Currently, this is partially done through digital certificates. The certificates, however, do not assure that the software is indeed safe. They only certify who takes (not so well defined) responsibility for the problems caused by the program. In fact, there were cases that a signed code caused serious security problems [2].

Another guarantee which is not assured by the traditional bytecode verification procedure is the lack of code flaws such as occurrences of null-pointer exceptions, array index out of bounds exceptions etc. In many cases, these flaws can be eliminated with the help of some additional information, e.g. @NonNull annotations, suggested in [18] (much in the way non-termination can be eliminated with help of the number of steps the underlying machine can take). Additionally, if the extended verification procedure would verify the lack of the

null-pointer exceptions then the checks for null values could be eliminated from the execution of bytecode instructions which would give a performance boost.

The goal of the proof-carrying code (PCC) technique [23, 12] is to give the user certain guarantees of the code to be executed at the moment of its execution. The check of the expected policy is performed by the user (or by user's execution environment) before the code is executed (see Fig. 1).

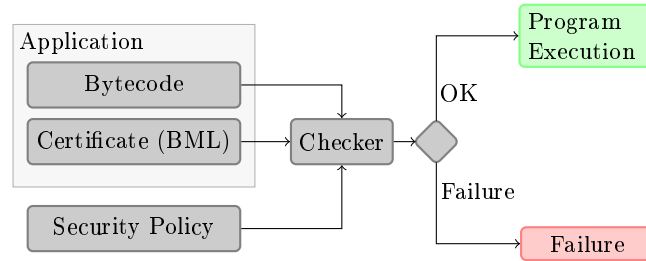


Fig. 1. Proof-Carrying Code architecture for bytecode

One of the possible ways to realise the PCC architecture is developed under the European project MOBIUS.¹ The logic-based methods developed in the project [3] rely on specification of object-oriented code in the fashion governed by the design-by-contract principles [22]. Bytecode Modeling Language (BML) is a specification language which realises the methodology at the bytecode level [7]. It is designed as a counterpart of the Java Modeling Language (JML), an established specification language dedicated to formally describe properties of Java programs in the design-by-contract style [20].

Multiple languages can now be compiled to Java bytecode: Python, JavaScript, Scala, etc. At SugarCon 2008, Sun Microsystems President and CEO Jonathan Schwartz said "we are just going to take the 'J' off the 'JVM' and just make it a 'VM'". Therefore there will be a global trend with support of companies to use JVM with languages other than Java. It is, thus, important to be able to express specifications at the bytecode level—in case the development is done in many languages the only common platform is the platform of the executable code. This explains the efforts concerning BML specification language.

The JML specification language exists already for several years. A lot of code have been annotated with the specifications in this language (see [6] for an overview). Except for that, it is easier to understand and specify the code in the source form than in the bytecode form. In this light, it is desirable to be able to translate these specifications from JML to BML. Moreover, the code producer in PCC scenarios, who has to produce a correctness proof, will often prefer to construct it rather in terms of the source code than in terms of the bytecode, and then compile the specification and the proof into the level of executable code.

¹ See <http://mobius.inria.fr>

In a broader perspective, the full infrastructure to support the use of BML annotated programs, for which complicated properties are checked at the user's end, requires the following items:

- PCC checker tools for BML annotations combined with PCC certificates,
- tools which enable the construction of PCC certificates,
- procedures to safely distribute the desired properties to be checked by PCC infrastructure,
- modelling languages (such as JML) for other programming languages,
- compilers that transform programs to JVM bytecode with annotations.

The *JML2BML* compiler described in this paper is designed to be a part of this scheme which translates the policies and specifications to the bytecode format.

Organisation of the paper In Sect. 2, we present the specification languages JML and BML. An example which illustrates the work of the compiler is presented in Sect. 3. Sect. 4 overviews the design of the *JML2BML* compiler. The problem of placement of the loop specification in the bytecode is discussed in Sect. 5. The related work is presented in Sect. 7 and we conclude in Sect. 8.

2 Specification languages

2.1 JML

The Java Modelling Language (JML) is a behavioural specification language for Java programs [20]. It allows to write specifications in the *design-by-contract* fashion [22]. Data types and method behaviour can be precisely documented using JML annotations. They describe the invariant properties that are maintained by objects, the input method requirements (preconditions), what we can expect at the output of method (postconditions) and also some lower level properties of the code (such as loop invariants). JML annotations are written in standard Java comments, so they do not disturb standard Java compilers.

The software development process must be divided into implementation and integration of multiple smaller components. JML allows to precisely specify the desired behaviour of the components which is crucial in case they must be combined with as little implementers' intervention as possible [24]. JML is also a language which abstracts from certain implementation details and in this way it can serve as a language to document *what* is written in code, but to hide *how* it is done. Moreover, it is possible to automatically check that the documentation expressed in JML is up to date [17].

JML stays as close as possible to the Java syntax and semantics to ensure the readability of its specifications and its tool support is rich (see [6]). In particular, there are tools that check JML specification at runtime [8], in extended static checking fashion [11], and allow to perform software certification [21]. There are also tools that support annotation generation [10, 15].

2.2 BML

The Bytecode Modelling Language (BML) is a specification language for Java bytecode. It was proposed by Burdy et al. in [7].

The design of BML directly follows the fundamental concepts of JML. It inherits most constructions and keywords from the JML syntax. As BML is developed within the MOBIUS [1] project and the main target of the project are Java-enabled mobile devices such as mobile phones, the current version of BML assumes some simplifications of the Java bytecode which are present in the J2ME mobile platform.

The class files with BML annotations are regular Java class files, executable by all Java tools. The annotations are stored within additional attributes. The BML related attributes start with the prefix `org.bmlspecs` and according to the specification of JVM they should be ignored in normal execution, since their names are not part of the original JVM specification.

Following the logical structure of class files, class specifications are stored as class attributes, method specifications, as attributes of corresponding methods and specifications inserted in the bytecode are subattributes of the JVM Code attributes.

2.3 Overview of annotations

The structure of annotations in BML and JML is very similar. We have two main types of annotations: method annotations and data type (class and interfaces) annotations.

Method annotations The most important type of method annotations are *method specifications* describing the input-output behaviour of the method. These are preconditions (**requires**), defining conditions that should be fulfilled before entering the method and postconditions (**ensures**) telling what we can expect after the method finishes. One can define also which fields are modified (clause **modifies**) and which exceptions might be thrown (clause **signals**).

The other type of method annotations are specifications appearing between mnemonics, such as:

- Assert clauses which state some facts about fields, variables etc. that should hold at this point of program execution.
- Loop specifications that describe the loop invariants (**loop_invariant**), loop termination measures (**decreases**), and modification descriptions (**modifies**).
- Declarations of local **ghost** variables—variables that are visible only in the specifications. Their values can be modified only using special **set** instructions.
- **Set** instructions are similar to Java assignments, but they operate on ghost fields and variables.

Data type annotations Class (and interface) specifications describe the behaviour of a class as a whole (in the **static** version) or of objects (**instance**). The most important type of *class specifications* are class invariants. They describe the property that should hold for all objects of this class in all *visible*

states, i.e. after all constructors and before and after all methods. For example, having a field `Object[] list`, one can write an invariant that the list is never null and its length is 10. Class invariants can be seen as additional, implicit preconditions and postconditions for all methods in the class.

Other important class specifications are declarations of `ghost` and `model` fields. Ghost fields are similar to local ghost variables, but are visible in the whole class scope. Model fields are present only in the specifications, representing some more complicated formulae. For example one can create a model field representing the property that a collection does not contain nulls. More details can be found in [19] and [9].

```

1 public class List {
    private Object[] list;

5  /*@ requires list != null;
   @ ensures \result ==(\exists int i;
   @ 0 <= i && i < list.length &&
   @ \old(list[i]) == o1 && list[i] == o2);
   @*/
9  public boolean replace(Object o1, Object o2){
   /*@
   @ loop_invariant i <= list.length
13  @ && i >=0 && (\forall int k;0 <= k
   @ && k < i ==> list[k] != o1);
   @ decreases list.length - i;
   @*/
17  for (int i = 0; i < list.length; i++) {
       if (list[i] == o1) {
           list[i] = o2;
           return true;
21     }
       }
       return false;
25 }

```

Fig. 2. An example class `List.java` containing single method `replace`

3 An example of using the compiler

3.1 Source code

Consider the class presented on Fig. 2. This is an excerpt of a class which implements a sequence of objects. We present here only one method that replaces in the `list` array the first occurrence of its first parameter with the second one. True will be returned, if and only if such an element was found.

The presented code, apart from standard Java statements, contains also specifications in JML. There is a precondition (`requires ...`) for the method `replace` defined. It requests that every time the method is invoked, the field `list` it not null². The next three lines (starting with `ensures`) constitute the method postcondition. It states that, if the precondition was fulfilled, then the

² In general a method can have multiple `requires-ensures` pairs. In this case the method can be invoked only in places where a condition specified by at least one of the `requires` clauses holds.

method result is true if and only if there was an element in the `list` the value of which has been updated from `o1` to `o2`. This is the guarantee which is ensured by the method call. Note that the postcondition makes use of some JML features, such as `\result`, `\old` or `\exists`. This postcondition does not describe all desirable properties of this method. For example an implementation that replaces all elements in the `list` up to the first occurrence of `o1` with `o2` will fulfil this specification.

```

/*@
@ requires this.list != null
@ ensures \result ==
@   (\exists int i; 0 <= i &&
@     i < this.list.length &&
@     old_this.list[i] == o1 &&
@     this.list[i] == o2)
@*/
public boolean replace(Object o1, Object o2)
0:   iconst_0
1:   istore_3
2:   goto      #27
5:   aload_0
6:   getfield   main.List.list
9:   iload_3
10:  aaload
11:  aload_1
12:  if_acmpne  #24
15:  aload_0
16:  getfield   main.List.list
19:  iload_3
20:  aload_2
21:  astore
22:  iconst_1
23:  ireturn
24:  iinc      %3  1
27:  iload_3
28:  aload_0
29:  getfield   main.List.list
32:  arraylength
/*@
@ loop_specification
@ modifies everything
@ invariant i <= this.list.length &&
@   i >= 0 &&
@   (\forall int k; 0 <= k &&
@     k < i ==> this.list[k] != o1)
@ decreases this.list.length - i
@*/
33:  if_icmplt  #5
36:  iconst_0
37:  ireturn

```

Fig. 3. The method `replace` in the `List.class`

In addition to the specification of the input-output behaviour of the method, also the main loop of the `replace` method is annotated. The `loop_invariant` clause contains the invariant: a formula that holds at the beginning of the loop body at each loop iteration. In this example it states that in iteration `i` there are no occurrences of `o1` in `list` on positions before `i`. The annotation `decreases` describes the loop variant. It gives an expression (in this case `list.length - i`) the value of which is decreased in each loop iteration by at least one.

3.2 Bytecode

In this section we describe the result of translating the source code from Fig. 2. The actual result of the compilation is a class file enriched with the attributes which contain the representation of BML specifications. However, the binary class files are not human readable. Thus, we rely for the current presentation on its textual representation obtained from Umbra [25]. Fig. 3 shows the translated `replace` method together with BML annotations inserted by our *JML2BML* compiler. The bytecode instructions labelled with 0 and 1 correspond to the initialisation `i = 0`. The loop is located between lines 2 and 33. Lines 5–12 represent the `if` statement, 15–23 correspond to lines 19–20 from the source code. Loading loop condition parameters is located in lines 27–32 and 33 performs the loop condition comparison.

The JML `requires-ensures` pair is directly translated to a corresponding `requires-ensures` pair at the BML level. We can see that the translation is almost literal copy of the original formula. The only difference is in the way the objects are referenced (BML requires to add explicit `this` prefix to fields) and the `\old` operator becomes a part of `this` reference).

Loop specifications are located after the line labelled with 32 in the presented listing. The *JML2BML* compiler detects loops in the bytecode and inserts the annotation before the bytecode representing the final conditional jump. In this case it is the `if_icmplt` instruction comparing `i` and `list.length` (detecting loops is described in Sect. 5.) The `modifies` clause describes a set of variables modified by the loop. We can see it in the listing as this clause is obligatory in BML. Currently, because of OpenJML limitations, it is not supported by our compiler so we see the default value (`everything`) inserted at this point.

4 JML2BML compiler design

JML2BML takes as input a Java source file with JML annotations together with the corresponding class file and outputs the class file with inserted proper BML annotations. Our compiler uses an enhanced Abstract Syntax Tree (AST) for the Java source code, taken from the OpenJML³ compiler (a Java compiler with JML checker based upon OpenJDK). For different types of JML clauses, there are separate translation rules defined. At each node of the AST, all translation rules are applied. If some rule succeeds to translate this node, the result is stored in the class file, using the BMLLib library [25]. This approach makes the compiler easily extensible. It is enough to just write a new translation rule to support additional features of the JML language. Moreover, the acyclic structure of the code should make the future maintenance more feasible and facilitate the understanding of the code mechanisms by future contributors [13].

Currently, the *JML2BML* compiler focuses on a subset of JML called JML Level 0 [19, Sect. 2.9]. Due to external libraries limitations not all desired features are translated, for example the loop `modifies` clause is not supported by

³ Available from <http://sourceforge.net/projects/jmlspecs>

OpenJML. The *JML2BML* compiler is designed to be compatible with other bytecode level tools, such as the bytecode editor *Umbra*.

4.1 Translation mechanism

Translation rules The *JML2BML* compiler uses a set of translation rules, each responsible for relatively small, independent piece of translation. For example we have separate rule for translating *assert* and another one for translating *loop_invariant*. Each rule is responsible for translation of a part of the AST tree resulting from a particular non-terminal of the input grammar (in most cases it boils down to the situation that each rule is applied exactly to one AST node). Translation rule may write results of the translation to the output class file (using BMLLib). It can, however, only collect some translated data that may be used by other translation rules. For example both—the *assert* and *loop_invariant* annotations contain expressions. Therefore we created an expression translation rule that makes translation of an expression but it does not write anything to output file—just returns the translated expression that will be used in other translation rules.

It is relatively easy to extend translation using the translation rule concept. One simply provides new implementation of a translation rule and registers it in the manager. Translation rule key design features are:

- the concept falls into a visitor design pattern,
- translation rule is an extension of a simple abstract class,
- translation process can be broken into smaller, independent pieces,
- extending translation is simple.

Example of a translation rule The rules in the implementation of *JML2BML* are based on abstract compilation rules described in [9, Chap.9] We present here a rule which translates a JML invariant into a BML invariant. The main non-trivial work of the rule is to combine all the JML invariants in the current class into a single invariant in the resulting BML representation (the specification in BML can contain only one instance invariant for a single class).

The logic of the translation rule can be described in an abstract way as follows:

```
Tr(invariant_keyword predicate, translation_ctxt) =
  replace(translation_ctxt,
    getInvariant(translation_ctxt),
    consInvariant(
      getInvExpression(
        getInvariant(translation_ctxt)) &&*
        getExpression(Tr(predicate, translation_ctxt))))
```

where $\text{Tr} : \text{JMLAST} \times \text{Ctxt} \rightarrow \text{Ctxt}$ is the function which defines the translation. It takes a JML AST node and a translation context and transforms this into a new translation context which contains the result of the translation.

The replace function replaces in the given translation context the item from the second argument with the item on the third argument. In our case, it replaces the current invariant (obtained using the `getInvariant` function from the current context) with the newly generated one. The newly generated invariant is constructed (using `consInvariant`) from the conjunction of the expression obtained from the already accumulated invariant (obtained using `getInvExpression`) with the expression being the result of the translation of the predicate in the currently translated invariant (`predicate`). One remark must be made about the operation of the conjunction. In case the first argument is undefined (it happens when we translate the first invariant in the class), the operation `&&*` results just in its second argument.

4.2 Translating expressions

In order to be able to translate any JML specification, one needs to translate JML expressions, so one of the most substantial tasks in writing the compiler was to write a translation rule for expressions. The syntax of both, JML and BML expressions follows the one of Java expressions which makes them easily accessible for typical Java programmers. The translation of expressions is in most cases straightforward. We face a more complicated situation when identifiers are translated, because one has to distinguish between fields, ghost fields, local variables, and bound variables. All these kinds of variables are resolved in different ways at the bytecode level which makes their compilation convoluted.

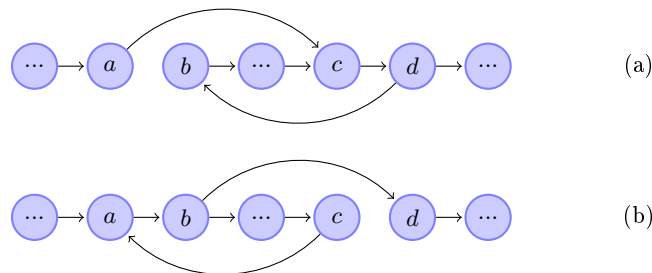


Fig. 4. Two ways of compiling loops

5 Detecting loops in bytecode

In some cases, it is crucial to use for the translation the content of the original class file, provided by the user. There is a need to link instructions from the source code with corresponding bytecode instructions from the provided class file. The most difficult and important part is to detect loops in the bytecode. A loop can be translated in one of the ways presented on Fig. 4.

In the first scenario (shown on Fig. 4(a)), an unconditional jump (goto) from the vertex a to the vertex c is done (vertex c denotes the start of the loop exit condition). In vertex d , the condition check is executed, and if it is fulfilled, we jump back to b . The instructions between b and c constitute the loop body. The annotation should be added to the vertex d . In the second approach (shown on Fig. 4(b)), the condition is tested at the beginning (a starts the sequence of instructions which evaluates the loop condition and b represents the actual check). If the loop condition is fulfilled, we enter the loop body which starts right after b , otherwise we jump out of the loop to the vertex d . In the vertex c , an unconditional jump back to the vertex a is done to check again the loop condition after the body is executed. The BML annotation should be associated with the instruction in the vertex b .

Another difficulty is posed by `do-while` loops and loops the exit condition of which is the constant `true` condition which prevents the loop to finish in a normal way (i.e. the loops of the shape `while(true){...}` or `for(;;){...}` loops). The loops of this kind are usually compiled in the way presented on Fig. 5.

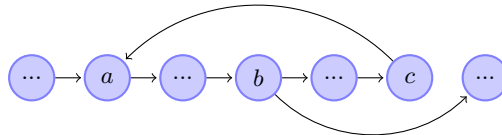


Fig. 5. Compiling `do-while` loop

Before entering the loop (between a and c), no condition is checked. There might be some `break` inside (vertex b). In these cases, the annotation should be associated with the vertex a (the start of the loop).

The *JML2BML* compiler covers all the cases described above. It tries to detect the first kind of a loop from Fig. 4. If it fails then it tries to detect the second one. At the end, the `do-while` case is checked.

5.1 Matching bytecode loops with source code loops

It is not enough to discover a loop in the bytecode. We have to find out which loop specification is pertinent to this particular piece of the code. The *line number table*⁴ is used to match the bytecode loop with its source code counterpart. To this end we calculate, with the help of the *line number table*, the range of the lines where the loop is located. In order to do that, we examine all the instructions in the bytecode loop and retrieve for them their corresponding source code lines from the table. The range we are interested in is delimited by the minimal and the maximal values found in this procedure.

⁴ In some cases this requires the compiler to be called with a proper flag.

This is, however, not enough as the translation starts from the source code. Therefore we have to take into the consideration the fact that a single source code loop may be represented in the bytecode several times (e.g. when a loop is located in a `finally` block of a Java `try-catch` statement—one of the translation strategies here is to copy the `finally` block to the end of both `try` and `catch` blocks). Thus, we find for every source code loop with `loop_invariant` present the best matching bytecode loops. This is done using the following algorithm:

- Pick the bytecode loops for which the calculated source code line range is in the range of the source code loop in question.
- Take the ones with the maximal range.
- Check if they have the same range of the source code lines.
- If so annotate all of them.
- Otherwise, fail.

The translation of assert clauses It is worth mentioning that the *line number table* is also crucial for the translation of JML `assert` expressions. They are assigned to the first bytecode instruction which realises the particular line. A potential `assert` formula to be inserted is determined using the correspondence from *line number table*.

6 The application of the compiler

The generation of class files with BML specifications can be conducted both by the application developers and the code distributors (the code distributors may want to add the specifications and proofs for instance to ensure that the programs do not disturb their infrastructure). In the latter case, the limited access to the source code makes the augmenting of the existing class files with BML attributes a sensible approach. One can just decompile the class files then specify them and subsequently move the specifications to the original class files with the help of *JML2BML*. In this case, it is even not strictly necessary that the specifications will be placed in exactly the right point as the bytecode will be examined by human anyways (e.g. using the BML editor Umbra [25]).

In the former case, the unlimited access to the source code may suggest that a better solution would be to extend a compiler backend to insert BML attributes along the normal bytecode. However, even in that case, the use of a compiler which augments the class files with BML attributes in a postprocessing stage may be useful. For instance, the development process may be tightly bound to a particular toolset (e.g. to Eclipse Java compiler because the environment supports easy refactoring or to Sun's Java compiler because the resulting code is better handled by virtual machines with JIT compilation). It is easier, then, to adopt a tool which adds attributes than to change the compilation infrastructure.

6.1 Performance

We did a performance experiment to establish the effectiveness of our tool. We compiled a project called Passwords which contained 23 classes with 177 methods that contributed altogether 1634 lines of code (988 non-commenting lines of code as calculated by Metrics tool). This project was annotated with JML specifications that allowed to secure the lack of the runtime exceptions and a simple security policy that the code does not print a secret data to public I/O channels. It took our compiler 1320ms to compile JML annotations to BML (average of 10 experiments). This value can be compared with 800ms of the compilation time that took Sun's compiler to transform the source code to byte code.

This running time can be explained by the fact that our compiler must manipulate two files instead of a single one. However, the result is satisfactory as the whole JML to BML compilation task is not supposed to be done in continuous fashion throughout the development of the project, but rather only once at the end of the process.

7 Related work

Another approaches to proof-carrying code for Java bytecode have been proposed [16, 26]. A notable technique consists in the development of a proof transforming compiler which translates the source code specifications together with corresponding correctness proofs to the formulae and proofs which correspond to the bytecode program (see e.g. [5]). The main difference with our approach is that the use of BML allows to have a single fixed representation of the properties to be guaranteed. These properties can then be proved by different proving technologies.

An example of an attribute with information that can be further exploited to check that the code indeed obeys the required policy is the StackMap attribute introduced as obligatory to Java 6. Also the attributes proposed by JSR305 and JSR308 specifications [18, 14] exploit the idea. The attribute format proposed by BML is more appropriate for the specification task as it allows to associate formulae not only to types or code points but also to classes and methods.

An early version of the *JML2BML* compiler has been realised in the Java Applet Correctness Kit (JACK) verification environment [4]. However, BML specification language and the attribute format have evolved largely since the implementation there was finished. Moreover, we propose here to attach the invariants not at the place where the loop control expression begins, but where the expression ends. This solution is semantically equivalent in case the loop condition is side-effect free and consistent with the current JML semantics in that case [19, Section 12.2.1] (moreover, the semantics is not precisely determined in case of conditions with side effects). We think that in case the condition is not side effect free our solution is better as then the invariant holds after the exit from the loop (which is not the case when the invariant is attached before the expression). One more important difference is that the algorithm used in JACK to discover the location of the loop invariant is based on line number table only

whereas we base our algorithm on the control flow of the graph. This means that our procedure can be used even in case the line number table is absent from the class file. In this case, however, the algorithm described in Sect. 5.1 should be replaced with something different e.g. an algorithm which constructs the control flow graph of the source code and matches the graph with the bytecode graph.

8 Conclusion

We have presented *JML2BML* compiler that transforms JML annotations into BML. The resulting annotations are inserted in binary format into the class file (using the BMLLib library). The compiler is an important step in building a common verification platform for all languages compiled to Java bytecode.

9 Acknowledgements

This work was partly supported by the Information Society Technologies programme of the European Commission, under the IST-2005-015905 MOBIUS project and Polish Ministry of Science grant 177/6.PR UE/2006/7. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

References

1. MOBIUS — Mobility, Ubiquity and Security. Enabling proof-carrying code for Java on mobile devices.
2. Anti-piracy CD problems vex Sony. BBC News web page, 8 December 2005. <http://news.bbc.co.uk/2/hi/technology/4511042.stm>.
3. Scenarios and requirements for PCC. MOBIUS web page, September 2006. http://mobius.inria.fr/twiki/pub/DeliverablesList/WebHome/deliv4_1.pdf.
4. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, LNCS. Springer-Verlag, 2007.
5. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, LNCS, Seoul, Korea, August 2006. Springer-Verlag.
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005. The original publication is available at <http://www.springerlink.com> from <http://dx.doi.org/10.1007/s10009-004-0167-4>.
7. L. Burdy, M. Huisman, and M. Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. *Fundamental Approaches to Software Engineering (FASE 2007)*, pages 215–229, 2007.
8. Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language. *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02)*, 2002.
9. J. Chrzęszcz, M. Huisman, J. Kiniry, M. Pavlova, and A. Schubert. *BML Reference Manual*, 2008.

10. M. Cielecki, J. Fulara, K. Jakubczyk, E. Jancewicz, A. Schubert, J. Chrzęszcz, and Ł. Kamiński. Propagation of JML non-null annotations in Java programs. *PPPJ'06: Proceedings of the 4th international symposium on Principles and Practices of Programming in Java*, pages 135–140, 2006.
11. D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, pages 108–128, 2005.
12. C. Colby, P. Lee, and G. C. Necula. A proof-carrying code architecture for Java. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 557–560, London, UK, 2000. Springer-Verlag.
13. E. W. Dijkstra. Notes on structured programming. Technical Report 70-WSK-03, Technological University Eindhoven, April 1970.
14. M. Ernst. Jsr 308: Annotations on Java types. JCP.org, November 5 2007. JSR 308.
15. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCammant, C. Pacheco, M. S. Tschantz, and C. Xiao. Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.
16. S. Gilmore and M. Prowse. Proof-carrying bytecode. In *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005)*, Edinburgh, Scotland, April 2005.
17. A. Goyal and S. Sankar. The application of formal specifications to software documentation and debugging. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging. First International Workshop, AADEBUG '93 Linköping, Sweden, May 3-5 1993, 1993 Proceedings*, volume 749 of *LNCS*, pages 333–349. Springer, 1993.
18. D. Hovemeyer and W. Pugh. Status report on JSR-305: annotations for software defect detection. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 799–800, New York, NY, USA, 2007. ACM.
19. G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, May 2008. <http://www.eecs.ucf.edu/~leavens/JML/OldReleases/jmlrefman.pdf>.
20. G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*, chapter 12, pages 175–188. Kluwer, 1999.
21. C. Marche, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML, 2004.
22. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, second edition edition, 1997.
23. G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.
24. K. Periyasamy and J. Chidambaram. Software reuse using formal specification of requirements. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative Research*, page 31. IBM Press, 1996.
25. A. Schubert, J. Chrzęszcz, T. Batkiewicz, J. Paszek, and W. Wąs. Technical aspects of class specification in Java byte code. *Proceedings of Bytecode'08*, 2008.
26. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J.-J. Levy, E. Mayr, and J. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, TC1 Third Int. Conf. on Theoretical Computer Science (TCS2004)*, pages 333–347, Dordrecht, 2004. Kluwer.