Supplementing Java Bytecode with Specifications

Jędrzej Fulara
Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw, Poland
fulara@mimuw.edu.pl

Krzysztof Jakubczyk
Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw, Poland
kjk@mimuw.edu.pl

Aleksy Schubert
Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw, Poland
alx@mimuw.edu.pl

ABSTRACT

Java class file is an interoperable format that can serve not only to transfer the compiled versions of Java programs, but also to embed into the files additional information which can be exploited by the execution environment of user's machine to speed up the execution of the application or to ensure certain vital properties of the code. The latter goal is specifically the aim of the proof-carrying code (PCC) techniques in which the executable code is supplied with a proof that the code obeys certain policy (e.g. the program does not store password cleartext in a file). BML (Bytecode Modelling Language) can be regarded as a part of the PCC architecture which allows to express detailed properties of bytecode programs, in particular the policies they must obey.

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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Verification

Keywords

Java, bytecode, JML, BML

1. INTRODUCTION

Each Java Virtual Machine is required to perform so called bytecode verification process on each class file which is loaded to be executed [12]. This verification process ensures vital properties of the loaded programs such as that all arguments on the operand stack are legal, all types of variables passed to methods are correct, all load and store operations have correct types, etc.

In certain situations, these guarantees are not sufficient. In particular, many of the current security guarantees are ensured at runtime of an application. 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 property. After this, she or he may easily be made to send data to e.g. an expensive premium number. In this case, 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 signatures. The signatures, however, do not assure that the software is indeed secure. They only certify who takes (not so well defined) responsibility for the problems caused by the program. In fact, there were cases when many users were deceived by respected companies e.g. rootkits were installed when audio CD was inserted [2].

Another guarantee which is not secured by the traditional bytecode verification procedure is the lack of code inconsistencies typically considered to be programming errors i.e. null-pointer exceptions, array index out of range exceptions etc. In many cases, these inconsistencies can be eliminated with the help of some additional information (e.g.@NonNull annotations, suggested in [14]) which makes the checking process feasable.

The goal of the proof-carrying code (PCC) technique [19] is to give the user certain guarantees of the code to be executed at the moment of its execution. In this paradigm, the executable code (in case of Java, the bytecode) is augmented with an additional piece of information, a PCC certificate, which together with the code makes possible an automatic check that the code indeed obeys the expected policy (e.g. sends data to authorised targets only). This check is performed by the user (or by user's execution environment) right before the code is executed (see Figure 1). In fact, the usual Java bytecode verification procedure can be viewed as an example of a forerunner of thich technique. Although the certificate is in this case empty (or supplementary in case the StackMap attributes are employed), the execution environment indeed checks a vital code property before the code is executed. In more complicated cases of the proof-carrying code, one needs the additional information to make

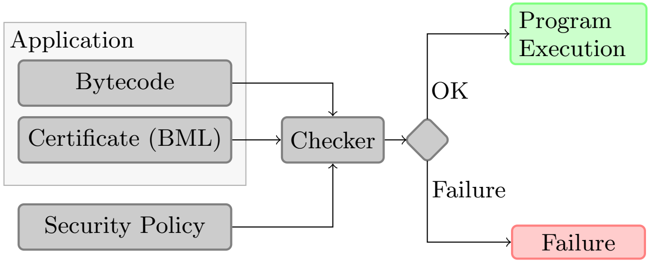

Figure 1: Proof-Carying Code architecture for bytecode.

the checking of the required property feasible or even algorithmically possible.

One of the possible ways to realise the PCC architecture is developed under the european project MOBIUS.[1] This project plans to develop both type-based and logic-based PCC program certification techniques [3]. The logic-based methods rely on specification of the object-oriented code in the fashion governed by the design-by-contract principles [18]. Bytecode Modeling Language (BML) is a specification language which realises the methodology at the bytecode level [5]. 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 [16].

The use of specification formalisms such as JML or BML has another important application. Modern, high level programming languages support creating software in a modular way. Applications can be divided into smaller parts that can be developed independently. In large systems, it is a common practice to outsource some well defined subsystems to external companies. The main problem in this approach is that the pieces of software developed by different programming teams often are not fully compatible. To avoid this incompatibility and useless code that is produced, there is a need to specify precisely the desired behaviour of the components, what they require and what can we expect from them [22]. The solution is to define formally implementation contracts in languages such as JML or BML and check automatically the compliance of the code with these descriptions.

Specification languages are useful in describing the system components behaviour. They are not only helpful in dividing the problem into smaller pieces, but also focus on *what* is expected from each part, without saying about *how* should it be done. The specification languages are designed to be simple enough to be understood by programmers, so they can play role of code documentation. Using specification language for documenting code has the advantage that it is possible to automatically verify that the source code implements the documented features [13].

The distributed development of software results in a situation where different software modules are implemented in different languages. This is facilitated by the fact that more and more languages are compiled to the same Java bytecode.

[1]See `http://mobius.inria.fr`

A few examples:

- Jython: the Python Java implementation,
- JRuby: the Ruby Java implementation,
- Jacl: the Tcl Java implementation,
- Rhino: the JavaScript Java implementation,
- Scala: a functional programming language compiled to Java bytecode.

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. This is an important reason, why it is important to be able to translate the source code specifications into lower level language specifications— 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. In the course of the time, a lot of code have been annotated with the specifications in this language (see [4] 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 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.

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 that understand 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 Java) for other programming languages,
- compilers that compile programs to JVM bytecode along with annotation compilers,

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 Section 2, we present the specification languages JML and BML. An example which illustrates the work of the compiler is presented in Section 3. Section 4 overviews the design of the *JML2BML* compiler. The most difficult problem of the specification compilation is the placement of the loop invariants. This issue is discussed in Section 5. The related work is presented in Section 6 and we conclude in Section 7.

2. SPECIFICATION LANGUAGES

2.1 JML

The Java Modelling Language (JML) is a behavioural specification language for Java prigrams [16]. It allows to write specifications according to the *design-by-contract* principles [18]. Data types and method behaviour can be precisely commented 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 (i.e. loop invariants, loop variants etc). JML annotations are written in standard Java comments, so they do not affect the normal work of any Java compiler.

An important goal in the design of JML is that it should be easily understandable by Java programmers. It is achieved by staying as close as possible to the Java syntax and semantics. The tool support for JML is rich (see [4] for an overview). In particular, there are tools that check JML specification at runtime [6], in extended static checking fashion [9], and allow to perform software certification [17]. There are also tools that support annotation generation [8],[11].

The works on JML was started by Gary Leavens at Iowa State University. Since then it became an open project, multiple groups around the world are writing tools supporting JML and developing the language itself.

2.2 BML

The Bytecode Modelling Language (BML) is a specification language for the bytecode. It was proposed by Burdy et al. in [5]. The design of BML directly follows the fundamental concepts of JML. It inherits most constructions and keywords from the JML syntax. As the 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 platform—the Java platform for mobile devices with restricted resources.

The class files representing bytecode 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 the Java Virtual Machine they should be ignored by the Machine, since their names are not part of the original JVM specification.

Of course, following the logical structure of class files, class specifications are stored as class attributes, method specifications, as attributes of corresponding method and specifications inserted in the code are attributes of the JVM Code attribute of the given method.

The document is organized as follows. In Section 2 we describe an annotation language. In Section 3 we describe implementation of our compiler, give details on the architecture and present the main principles of the translation. Then an example of using our tool is presented. In Section 5 and 6 we describe algorithms for detecting loops in the bytecode and matching them with the source code. At the end conclusions are given and future work is discussed.

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.

2.3.1 Method annotations

The most important type of method annotations are *method specifications* describing the input-output behaviour of the method. This 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 `signal`).

The other type of method annotations are specifications elements appearing in the code, like:

- Assert instructions that 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 variants, like `decreases` to prove the loop's termination or `modifies` that tells which fields or variables can be changed in this loop.
- Declarations of local `ghost` variables—variables that exist only in the specification. Their values can be modified only using special `set` instructions.
- `Set` instructions are similar to Java assingments, but they operate on ghost fields and variables.

2.3.2 Data type annotations

Class (and interface) specifications describe the behaviour of a class as a whole (in the `static` version) or of objects of that class (`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 postcondtitions for all methods in the class.

Other important class specifications are:

- Declarations of `ghost` fields. They are similar to local ghost variables, but are visible in the whole class scope.

- Model fields—fields present only in the specifications, representing some more complicated formulas. For example one can create a model field representing the property that a collection does not contain nulls.

More details can be found in [15] and [7].

```
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 }
```

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

3. AN EXAMPLE OF USING THE COMPILER

This section provides an example demonstrating the result of launching the *JML2BML*.

3.1 Source Code

Consider the class presented on Figure 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 occurence 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 the 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 `null`. The next three lines (starting with `ensures` constitute the method postcondition. It states that, if the precondition was fullfiled, then the method result is true if and only if there was an element in the `list` which value has been updated from `o1` to `o2`. Note that the postcondition makes use of some JML features, like `\result`, `\old` or `\exists`. This postcondition does not describe all properties of this method. For example an implementation that replaces all elements in the `list` up to the first occurence of `o1` with `o2` will fulfill this specification.

In addition to specification describing input-output behaviour of the method, also the loop implementing the `replace` method is annotated. The `loop_invariant` clause contains the invariant: a formula that should hold at the beginning of the loop body at each loop iteration. In this example it states that in iteration `i` there are no occurences of `o1` in `list` on positions before `i`. The annotation `decreases` describes the loop variant. It specifies an expression (in this case `list.length - i`) which value 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 Figure 2. The actual result of the compilation is a class file enriched with the attributes which contain the representation of BML specifications. This means that the binary class files are not human readable. Thus, we rely for the current presentation on its textual representation obtained from the BMLLib. The Figure 4.1 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 initialization `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 `requires-enusers` pair is translated into input-output behaviour BML specifications located just before the method code. We can see that the BML code contains two places where the counterpart of the JML `requires` clause can be placed. The first one is right after the `requires` keyword and the second one is after the `precondition` keyword. In fact, both JML and BML allow one to specify many pairs of the input-output specifications. The semantics of the multiple pairs is such that for each pair such that the requires formula holds at the entry to the method, the corresponding ensures formula must hold at the exit (so effectively the conjunction of all the ensures formulas holds). However, we often want to specify that a particular method can be called only in specific context (e.g. with first parameter being non-null).and except for that it should obey the input-output behaviour described in multiple `requires-enusers` pairs. In this situation, it is more convenient to distinguish a single additional clause which should hold in all the cases. In our JML code, this clause is implicit and equivalent to `true` whereas in BML this is made explicit as the formula after the `requires` keyword. This also is why the actual JML precondition is translated to the internal `precondition` statement (it is one of potentially many preconditions).

Loops specifications are located after line 32 in the presented listing. The *JML2BML* compiler detects loops in the bytecode and inserts the annotation before the statement representing the loop condition. In this case it is the `if_icmplt` instruction comparing `i` and `list.length`. For more details about detecting loops refer to Section 5. The modifies clause describes set of variables modified by the loop. Currently, because of OpenJML limitations, it is not supported by our compiler (the default value `everything` will be inserted).

4. JML2BML COMPILER DESIGN

In our work we have designed and implemented a tool called *JML2BML* that compiles JML specifications into BML. It 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 OpenJML [2] compiler (a JML checker based upon the OpenJDK Java tool set). 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 to the class file, using the BMLLib library [23]. This approach makes the compiler easily extensible. One can simply write new translation rule to support additional features of the JML language.

Currently, the *JML2BML* compiler focuses on subset of the JML called JML Level 0. Due to external libraries limitations not all desired features are translated, for example the `loop_modifies` clause is not supported by the OpenJML.

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

4.1 Architecture Description

In this section we present the overview of the architecture of *JML2BML* compiler. The *JML2BML* compiler uses OpenJML to parse the Java source code together with the JML annotations. To insert generated BML annotations, the BMLLib library is used. The dependencies between internal packages of *JML2BML* and OpenJML and BMLLib are presented in Figure 4.1.

The `jml2bml.main` package provides the entry point to the application. JML annotations from given source file will be translated and inserted into corresponding `class` file. Functions to access some bytecode information are located in `jml2bml.bytecode` and helpers to BMLLib are collected in `jml2bml.bmllib`. The `jml2bml.rules` package contains translation rules for different aspects of JML. It should be easy to add new rules in the future. Classes for traversing the Java abstract syntax tree can be found in `jml2bml.ast`. In `jml2bml.symbols` implementation of symbol table can be found. The `jml2bml.engine` package contains the core translating mechanism.

4.2 Translation Mechanism

The full translation consists of a set of independent translation rules. Having the set of rules the AST tree of the source code with annotations is traversed. For each visited node all translation rules are applied. For most nodes translation rules do nothing—each is responsible for few node types. The translation mechanism allows to register new rules in a very simple way, what is an important issue in case of implementing new features in the future.

4.3 Translation Rules

The *JML2BML* compiler uses a set of translation rules. The concept of translation rule is that it should be responsible for relatively small, independent piece of translation. For example we have separate rule for translating *assert* and another one for translating *loop_invariant*. Translation rule may write results of translation to the output class file (using BMLLib). It can hovewer only collect some translated

[2]Available from `http://sourceforge.net/projects/jmlspecs`

```
/*@
  @ requires true
  @  {|
  @   precondition list != null
  @   ensures \result ==
  @     (\exists int i; 0 <= i &&
  @                 i < list.length &&
  @                 old_list[i] == o1 &&
  @                 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:   aastore
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 <= list.length &&
  @       i >= 0 &&
  @       (\forall int k; 0 <= k &&
  @           k < i ==> list[k] != o1)
  @   decreases list.length - i
  @*/
33:   if_icmplt   #5
36:   iconst_0
37:   ireturn
```

Figure 3: The method `replace` in the `List.class`

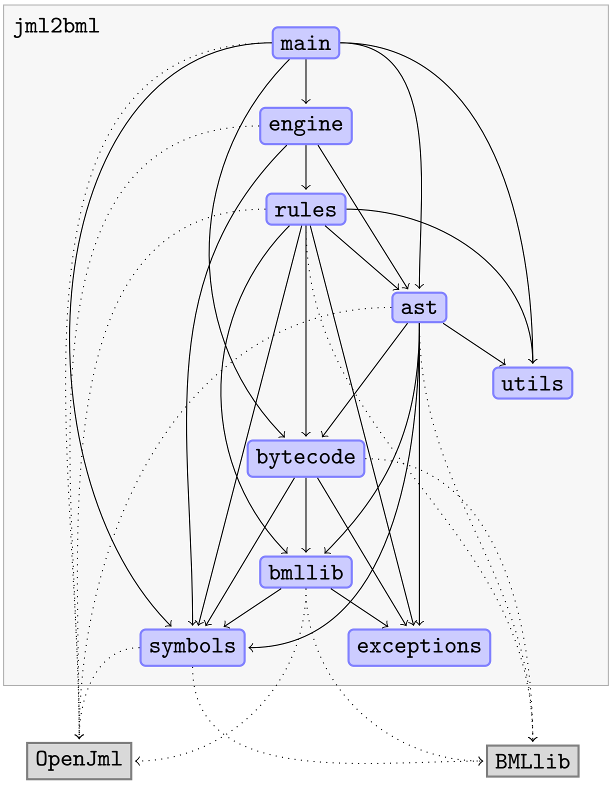

Figure 4: The dependency graph of the *JML2BML* packages. Dotted lines denote access to external libraries.

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. For example if we would like to translate an annotation that was not already translated we should create a new translation rule implementation and include it in the translation mechanism by registering the rule in the translation manager. When implementing the translation rule we should first find out which AST nodes are important and then override proper methods of the base class.

Translation rule key 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

4.4 Example of Translation Rule

```java
public class TypeClauseExprRule
  extends TranslationRule<String, Symbols> {
  /** Context object. */
  private final Context context;

  /**
   * Constructor of the rule.
   * @param context context object
   */
  public TypeClauseExprRule(Context context) {
    super();
    this.context = context;
  }

  /**
   * Main translation method.
   *
   * @param node node to be translated
   * @param symb symbol table
   * @return empty string
   */
  @Override
  public String visitJmlTypeClauseExpr(
      JmlTypeClauseExpr node, Symbols symb) {
    if (node.token == JmlToken.INVARIANT) {
      BCClass clazz =
          context.get(BCClass.class);
      AbstractFormula formula =
          TranslationUtil.getFormula(
                      node.expression,
                      symb, context);

      ClassInvariant classInvariant =
          clazz.getInvariant();
      if (classInvariant == null) {
        classInvariant =
          new ClassInvariant(clazz, formula);
      } else {
        AbstractFormula newFormula =
            new Formula(Code.AND,
                classInvariant.getInvariant(),
                formula);
        classInvariant =
            new ClassInvariant(clazz,
                              newFormula);
      }
      clazz.setInvariant(classInvariant);
      return "";
    } else
      return null;
  }
}
```

Figure 5: Example implementation of translation rule for class invariants.

Figure 5 contains example implementation of translation rule - for class invariants. The rule extends `TranslationRule` class. It has one attribute `context`, which holds the context in which rule will be executed. Class invariants in OpenJML abstract syntax tree are represented by `JmlTypeClauseExpr` class nodes. As we use visitor pattern, we have to override method responsible for these nodes - `visitJmlTypeClauseExpr`.

Implementation of translation method is simple. First we check if this is the node type we are interested in - line 25. We get an object representing the class which we want to add invariant to (line 26). Next, in line 28, we translate the content of invariant - rule for translating expression is used here. In line 34 existing class invariant is taken from the class. If there was no invariant - new one is created (line 37), otherwise we connect old invariant formula with new one using `and` (line 42) to create new class invariant - line 45. At the end, in line 47 we set new invariant to the class.

4.5 Translating Expressions

To be able to translate any JML specification, one needs to translate JML expressions, so the fundamental task in writing the compiler was to write a translation rule for expressions. As BML is based on JML, the syntax of expressions is similar in both languages and includes:

- Binary arithmetic operations (+,-,*,/)
- Boolean operations
- Relational operators ($<$, $\leq$, $\geq$, $!=$, etc.)
- Logical formulae containing
  - special expressions such as reference to old value (`\old`), reference to the result of method (`\result`),
  - implications, conjunctions, alternatives, etc.,
  - quantifiers (with bound variables).

As in standard Java, also in JML and BML the expressions can contain local variables, references to fields (both standard and ghost), method invocations, array access etc. There can also appear constructions specific for JML and BML, like \`old` clause. The translation of expressions is in many cases straightforward. Translation of identifiers is more complicated, because one has to distinguish between fields, ghost fields, local variables and bound variables and resolve them properly at the bytecode level.

5. DETECTING LOOPS IN BYTECODE

In some cases it is crucial to use the original, provided by user class file, the Jml2Bml compiler is therefore not allowed to compile the source code on its own. 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. To be able to compile the JML loop invariants, one should detect in the bytecode the corresponding loop. The created BML annotation should be associated with the bytecode instruction that represents the loop condition. Note that the loop condition is translated into multiple bytecode instructions. We are interested in the last one (comparison). A loop can be translated in one of the ways presented on Figure 6.

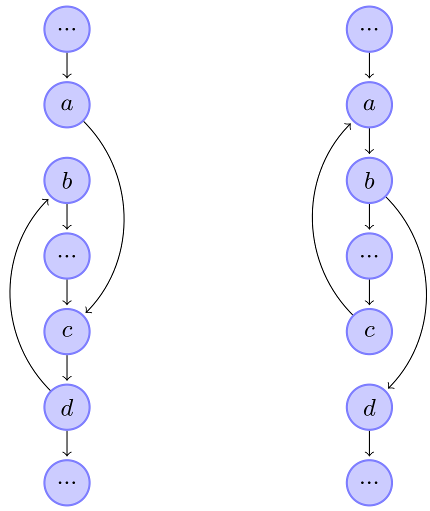

Figure 6: Two ways of compiling loops.

In the first scenario, in the vertex a, an unconditional jump (goto) to the vertex c is done (vertex c denotes loading the condition). In d the condition is checked, and if it is fulfilled, we jump back to b. Between b and c is the loop body. The annotation should be added to the vertex d. In the second approach, the condition is tested at the beginning (a puts the condition on the stack and b checks it. If it is fulfilled, we enter the loop, otherwise we jump out). In c an unconditional jump back to a is done. The BML annotation should be associated with the instruction in the vertex a.

`Do-while` loops and loops with always true condition (i.e. `while(true){...}` or `for(;;){...}` are usually compiled in the way presented in Figure 5

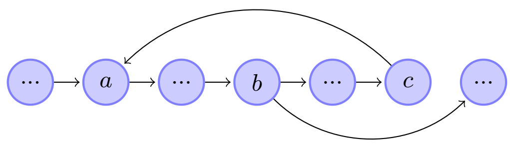

Figure 7: Compiling `do-while` loop.

Before entering the loop (between a and c), no condition is checked. There might be some `break` inside (vertex b). In this cases, the annotation should be added to a (start of the loop). The *JML2BML* compiler covers all the cases described above. It tries to detect the first kind of loop. If it fails, tries to detect the second one. At the end checks the `do - while` case. In the first case:

- assume that the tested instruction is in vertex c. Consider all incoming edges that start in a vertex v, which is before c
- if there are no such vertices, return null (tested instruction is not the c vertex in the first kind loop)

- else take this v that has the longest jump to c (other jumps come from some continue instructions inside the loop). This is a from our graph,
- find the first instruction after a that has an incoming backward edge starting in an vertex after c. This is our b. Find the longest backward jump. It is our vertex d
- return d

If no loop of the first kind was detected for an instruction, try to detect the second kind:

- assume that the tested instruction is a.
- if the instruction has less than two incomming edges, return null
- find v that is after a and has the longest jump to it. This is c from our graph.
- look at the next instruction d.
- find such u that there exist an edge (u,d) and u is between a and d and there is no such u' that u' is between a and u and there is an edge (u',d). This is candidate for b
- if at u is an unconditional jump (goto), then this is a break; this is the case of loop with always true condition. Return a
- else (at u is a conditional jump); u is really our b. Return it.

If both cases described above fail, the algorithm tries to detect the `do - while` loop. We simply check if

- there is a backward jump from the tested instruction to some a
- if yes, assuming that cases 1 and 2 failed, return a as the beginning of the loop

5.1 Matching Bytecode Loops with Source Code Loops

Let us take any source method that has loop with JML invariant and bytecode corresponding to this method. The compiler used to generate the bytecode may have used some optimizations. Unfortunately this causes some problems. Here are some exemplary loop optimizations:

- loop unwinding (loop unrolling)
  In this case invariant should be put in every copy of the loop. But the invariant may need a change.
- loop interchange
  If internal loop invariant depended on external loop variables—this invariant must also be translated.
- code-motion
  Some part of code may be moved before the loop. What if invariant depended on it?

When we want to add BML specifications to optimized bytecode, we have to know the optimizations that were used. There are two solutions of this problem:

- Include the JML to BML compiler in existing Java compiler application.
- Use non-optimizing compiler.

The first solution has the advantage that even optimized bytecode may be annotated. Unfortunately it would have to use an existing compiler infrastructure so every change in the compiler might reflect a change in translating annotations. This would be very difficult and complicated.

In second solution translations are more predictible and simplier compared to the optimizing compiler. Different compiler implementations may be used eg. Jikes or the reference one.

In both cases class level annotations (method pre-post conditions, class invariants) can be translated in the same way. Therefore when we limit to only these annotations we can use *JML2BML* compiler even with an optimizing compiler.

For matching source loops with detected bytecode loops we use *line number table* so the source java file must be compiled with the proper flag. Using the *line number table* is cruitial for translating JML `assert` expressions. Without the knowledge how Java compiler works—how it translates every Java expression to bytecode, it would be impossible to locate proper place in bytecode where the BML `assert` should be placed. Knowing where in bytecode loops are placed and having *line number table* in input Java class file, we can detect for evely such loop the line range in source file where the loop is placed. To get the beginning line number and ending line number corresponding to a given bytecode loop in source file, we search all instructions of the loop and use the *line number table* to get line number of the instruction. When we already have source file line ranges for every bytecode loop we have to take into consideration fact, that source loop can be present in bytecode more than once. This may happen when loop is placed in `finally` block of Java `try-catch` statement—translated `finally` block can be coppied to the end of both `try` and `catch` blocks. To match source loop to bytecode loops for every source code loop with `loop_invariant` present, we search for the best matching bytecode loop:

- bytecode loop line range must be in the source loop line range
- in bytecode there cannot be any other loop that has line range bigger but still in source loop line range
- there can be more than one bytecode loops matching but every all of them must have equal source line range

6. RELATED WORK

JML2BML compiler uses external libraries that are still in developement (BMLLib, OpenJML). They do not provide whole functionality needed to compile the JML Level 0.

When they get more powerful, also our compiler should be enhanced to support more sophisticated specifications (and therefore get more adequate for the end user). The most urgent are the `modifies` clauses and `set` instructions. The compiler should be also provided as a Eclipse plugin.

7. CONCLUSION

We have presented *JML2BML* compiler that deals with JML annotations and translates them 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 the Java bytecode.

8. 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] 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.

[5] 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.

[6] 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.

[7] J. Chrząszcz, M. Huisman, J. Kiniry, M. Pavlova, and A. Schubert. *BML Reference Manual*, 2008.

[8] M. Cielecki, J. Fulara, K. Jakubczyk, Ł. Jancewicz, A. Schubert, J. Chrząszcz, 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.

[9] 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.

[10] M. Ernst. Jsr 308: Annotations on Java types. JCP.org, November 5 2007. JSR 308.

[11] 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.

[12] A. Gal, M. Franz, and C. W. Probst. Java bytecode verification via static single assignment form.

[13] A. Goyal and S. Sankar. Automated and algorithmic debugging. first international workshop, aadebug '93 linköping, sweden, may 3̆20135, 1993 proceedings. volume 749 of *LNCS*, pages 333–349. Springer, 1993.

[14] 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.

[15] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, 2005.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*, chapter 12, pages 175–188. Kluwer, 1999.

[17] C. Marche, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML, 2004.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, second edition edition, 1997.

[19] 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.

[20] G. C. Necula. A scalable architecture for proof-carrying code. In *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming*, pages 21–39, London, UK, 2001. Springer-Verlag.

[21] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91, London, UK, 1998. Springer-Verlag.

[22] 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.

[23] A. Schubert, J. Chrząszcz, T. Batkiewicz, J. Paszek, and W. Wąs. Technical aspects of class specification in Java byte code. *Proceedings of Bytecode'08*, 2008.