Specification language for Java bytecode programs

Contents

1 Introduction **2**

2 A quick overview of JML **2**

3 The subset of JML supported in BML **5**

3.1 Notation convention 6
3.2 BML Grammar 7
3.3 Syntax and semantics of BML 8
3.3.1 BML expressions 9
3.3.2 BML predicates 10
3.3.3 Class Specification 10
3.3.4 Inter — method specification 10
3.3.5 Method specification case 10
3.3.6 Intra — method specification 11
3.3.7 Frame conditions 13

4 Well formed BML specification **14**

5 Compiling JML into BML **14**

1 Introduction

This section presents a bytecode level specification language, called for short BML and a compiler from a subset of the high level Java specification language JML to BML.

Before going further, we discuss what advocates the need of a low level specification language. Traditionally, specification languages were tailored for high level languages. Source specification allows to express complex functional or security properties about programs. Thus, they are successfully be used for software audit and validation. Still, source specification in the context of mobile code does not help a lot for several reasons.

First, the executable or interpreted code may not be accompanied by its specified source. Second, it is more reasonable for the code receiver to check the executable code than its source code, especially if he is not willing to trust the compiler. Third, if the client has complex requirements and even if the code respects them, in order to establish them, the code should be specified. Of course, for properties like well typedness this specification can be inferred automatically, but in the general case this problem is not decidable. Thus, for more sophisticated policies, an automatic inference will not work.

It is in this perspective, that we propose to make the Java bytecode benefit from the source specification by defining the BML language and a compiler from JML towards BML.

BML supports the most important features of JML. Thus, we can express functional properties of Java bytecode programs in the form of method pre and postconditions, class and object invariants, assertions for particular program points like loop invariants. To our knowledge BML does not have predecessors that are tailored to Java bytecode.

In section 2, we give an overview of the main features of JML. A detailed overview of BML is given in section 3. As we stated before, we support also a compiler from the high level specification language JML into BML. The compilation process from JML to BML is discussed in section 5. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

2 A quick overview of JML

JML [7] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications which follows the design-by-contract approach (see [2]).

Over the last few years, JML has become the de facto specification language for Java source code programs. Several case studies have demonstrated that JML can be used to specify realistic industrial examples, and that the different tools allow to find errors in the implementations (see e.g. [3]). One of the reasons for its success is that JML uses a Java-like syntax. Other important factors for the success of JML are its expressiveness and flexibility.

JML is supported by several verification tools. Originally, it has been designed as a language of the runtime assertion checker [6] created by G.T. Leavens and . The JML rac compiles both the Java code and the JML specification into executable bytecode and thus, in this case, the verification consists in executing the resulting bytecode. Several static checkers based on formal logic exist which use JML as a specification language. Esc/java [8] whose first version used a subset of JML [1] is among the first tools supporting JML. Among the static checkers with JML are the Loop tool developed by the Formal group at the University of Nijmegen, the Jack tool developed at Gemplus, the Krakatoa tool created by the Coq group at Inria France. The tool Daikon [**?**] tool uses a subset of JML for detecting loop invariants by run of programs. A detailed overview of the tools which support JML can be found in [4].

who are the others

Specifications are written using different predicates which are side-effect free Java expressions, extended with specification-specific keywords. JML specifications are written as comments so they are not visible by Java compilers. The JML syntax is close to the Java syntax: JML extends the Java syntax with few keywords and operators. For introducing method precondition and postcondition the keywords **requires** and **ensures** are used respectively, **modifies** keyword is followed by all the locations that can be modified by the method, **loop_invariant**, not surprisingly, stands for loop invariants, the **loop_modifies** keyword gives the locations modified by loop invariants etc. The latter is not standard in JML and is an extension introduced in [5]. Special JML operators are, for instance, **\result** which stands for the value that a method returns if it is not void, the **\old(expression)** operator designates the value of **expression** in the prestate of a method and is usually used in the method's postcondition. JML also allows the declaration of special JML variables, that are used only for specification purposes. These variables are declared in comments with the **model** modificator and may be used only in specification clauses.

Figure 1 gives an example of a Java class that models a list stored in a private array field. The method `replace` will search in the array for the first occurence of the object `obj1` passed as first argument and if found, it will be replaced with the object passed as second argument `obj2` and the method will return true; otherwise it returns false. The loop in the method body has an invariant which states that all the elements of the list that are inspected up to now are different from the parameter object `obj1`. The loop specification also states that the local variable `i` and any element of the array field `list` may be modified in the loop.

A useful feature of JML is that it allows two kinds of method specification, a *light* and *heavy* weight specification. An example for a *light* specification is the annotation of method `replace` in Fig. 1. The specification in the example states what is the expected behavior of the method and under what desired conditions it might be called. The user, however, has also the possibility to write very detailed method specifications in JML. This style of specification is called a heavy weight specification. It is introduced by the JML keywords

[1]the current version of the tool esc/java 2 supports almost all JML constructs

```
public class ListArray {

  private Object[] list;

  //@ requires list != null;
  //@ ensures \result ==(\exists int i;
  //@ 0 <= i && i < list.length &&
  //@ \old(list[i]) == obj1 && list[i] == obj2);
  public boolean replace(Object obj1,Object obj2){
    int i = 0;

    //@ loop_modifies i, list[*];
    //@ loop_invariant i <= list.length && i >=0
    //@ && (\forall int k;0 <= k && k < i ==>
    //@      list[k] != obj1);
    for (i = 0; i < list.length; i++ ){
      if ( list[i] == obj1){
        list[i] = obj2;
        return true;
      }
    }
    return false;
  }
}
```

Figure 1: CLASS ListArray WITH JML ANNOTATIONS

normal_behavior and **exceptional_behavior**. As the keywords suggest every of them specifies a specific normal or exceptional behavior of a method. (see [**?**]). The keyword **normal_behavior** introduces a precondition which guarantees that if it holds in the prestate of the method execution then the method will terminate normally and the postcondition introduced by the keyword **ensures** will hold. Note that this clause guarantees that the method will not terminate on an exception and thus the exceptional postcondition for any kind of exception (i.e. for the exception class `Exception`) is **false**. An example for a *heavy* weight specification is given in Fig. 2. In the example, method `divide` has two behaviors, one in case the method terminates normally and the other in case the method terminates by throwing an object reference of `ArithmeticException`. The exceptional postcondition is ommitted in the normal behavior specification clause as by default if the precondition `b > 0` holds this assures that no exceptional termination is possible. Another observation over the example is that the exceptional behavior is introduced with the JML keyword **also**. The keyword `also` serves for introducing every new behavior of a method. Note that

```
public class C {
    int a;

    //@ public instance invariant a > 0 ;

    //@ requires val > 0 ;
    public C(int val){
       a = val ;
    }

    //@ public normal_behavior
    //@ requires b > 0;
    //@ modifies a;
    //@ ensures  a == \old(a) / b;
    //@
    //@ also
    //@ public exceptional_behavior
    //@ requires b == 0;
    //@ modifies \nothing;
    //@ exsures (ArithmeticException) a == \old(a);
    public void divide(int b) {
        a = a / b;
    }
}
```

Figure 2: An example for a method with a heavy weight specification in JML

the keyword **also** is used in case a method overrides a method from the super class. In this case, the method specification is preceded by the keyword **also** to indicate that the method should respect also the specification of the super method.

JML can be used to specify not only methods but also properties of a class or interfaces. A Java class may be specified with an invariant or history constraints. An invariant of a class is a predicate which holds at all visible states of every object of this class (see for the definition of visible state in the JML reference manual [**?**]). A Class history constraints is a property which relates the initial and terminal state of every method in the corresponding class. The class `C` in Fig.2 has also an invariant which states that the instance variable `a` is always greater than `0`.

3 The subset of JML supported in BML

BML corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties. The following Section 3.1 gives the notation conventions adopted here and Section 3.2 gives the formal grammar of BML as well as an informal description of its semantics.

3.1 Notation convention

- Nonterminals are written with a *italics* font
- Terminals are written with a **boldface** font
- brackets [ ] surround optional text.

3.2 BML Grammar

$constants_{bml}$ ::= $intLiteral \mid signedIntLiteral \mid$ **null** $\mid ident$

$signedIntLiteral$ ::= $+nonZerodigit[digits] \mid -nonZerodigit[digits]$

$intLiteral$ ::= $digit \mid nonZerodigit[digits]$

$digits$ ::= $digit[digits]$

$digit$ ::= **0** $\mid nonZerodigit$

$nonZerodigit$ ::= **1** $\mid \ldots \mid$ **9**

$ident$ ::= $\#\ intLiteral$

$boundVar$ ::= **bv**_$intLiteral$

E_{bml} ::= $constants_{bml}$
| **reg**($digits$)
| E_{bml}.$ident$
| $ident$
| **arrayAccess**(E_{bml}, E_{bml})
| $E_{bml}\ op\ E_{bml}$
| **cntr**
| **st**(E_{bml})
| **old**(E_{bml})
| **EXC**
| **result**
| $boundVar$
| **typeof**(E_{bml})
| **type**($ident$)
| **elemtype**(E_{bml})
| **TYPE**

op ::= **+** | **-** | **mult** | **div** | **rem**

$\mathcal{R}$ ::= $= \mid \neq \mid \leq \mid \leq \mid \geq \mid > \mid <:$

P_{bml} ::= $E_{bml}\ \mathcal{R}\ E_{bml}$
| **true**
| **false**
| $not\ P_{bml}$
| $P_{bml} \wedge P_{bml}$
| $P_{bml} \vee P_{bml}$
| $P_{bml} \Rightarrow P_{bml}$
| $P_{bml} \iff P_{bml}$
| $\forall\ boundVar, P_{bml}$
| $\exists\ boundVar, P_{bml}$

$classSpec$ ::= **ClassInv** P_{bml}
| **ClassHistoryConstr** P_{bml}
| **declare ghost** $ident\ ident$

$intraMethodSpec$ ::= **atIndex** nat;
$assertion$;

$assertion$ ::= $loopSpec$
| **assert** P_{bml}
| **set** E_{bml}, E_{bml}

methodSpec	$::=$ *specCase*
	$\mid$ *specCase* **also** *methodSpec*
specCase	$::=$ **requires** P_{bml}; **modifies** *list locations*; **ensures** P_{bml}; *exsuresList*
exsuresList	$::=$ [] $\mid$ **exsures** (*ident*) P_{bml}; *exsuresList*
locations	$::= E_{bml}.ident$
	$\mid$ **reg**(i)
	$\mid$ *arrayModAt*(E_{bml}, *specIndex*)
	$\mid$ **everything**
	$\mid$ **nothing**
specIndex	$::=$ all $\mid i_1..i_2 \mid i$
bmlKeyWords	$::=$ **requires**
	$\mid$ **ensures**
	$\mid$ **modifies**
	$\mid$ **assert**
	$\mid$ **set**
	$\mid$ **exsures**
	$\mid$ **also**
	$\mid$ **ClassInv**
	$\mid$ **ClassHistoryConstr**
	$\mid$ **atIndex**
	$\mid$ **loopInv**
	$\mid$ **loopDecreases**
	$\mid$ **loopModif**
	$\mid$ \ **typeof**
	$\mid$ \ **elemtype**
	$\mid$ **TYPE**
	$\mid$ **result**

3.3 Syntax and semantics of BML

In the following, we will discuss informally the semantics of the syntax structures of BML. Note that most of them are identical with construction from JML. In the following, we will concentrate more on the specific features of BML and will just briefly comment the BML features which it inherits from JML like preconditions and which we have not mentioned already, [2]. We will discuss

[2]because we have already discussed in Section 2 the JML constructs for pre and postconditions, loop invariants, operators like **old**, **result**, etc. we would not return to them anymore

the specifities of the different part of the BML grammar starting with BML expressions.

3.3.1 BML expressions

Among the common features of BML and JML are the following expressions: field access expressions $E_{bml}.ident$, array access ($\mathbf{arrayAccess}(E^1_{bml}, E^2_{bml})$), arithmetic expressions ($E_{bml}\ op\ E_{bml}$). Like JML, BML may talk about expression types as shown in the grammar. As the BML grammar shows, $\backslash\mathbf{typeof(}E_{bml}\mathbf{)}$ denotes the dynamic type of the expression E_{bml}, $\backslash\mathbf{type(}ident\mathbf{)}$ is the class described at index *ident* in the constant pool of the corresponding class file. The construction $\backslash\mathbf{elemtype(}E_{bml}\mathbf{)}$ denotes the type of the elements of the array E_{bml}, and $\backslash\mathbf{TYPE}$, like in JML, stands for the Java type `java.lang.Class`.

A difference between JML and BML expressions is the encoding of identifiers and more particularly, the encoding of local variables, method parameters and field identifiers. In JML all these constructs are represented by their names in the Java source file. This is not the case in BML.

We first look at the encoding of method local variables and parameters. The class file format stores information for them in the array of local variables. That is why, both method parameters and local variables are represented in BML with the construct $\mathbf{reg}(i)$ which refers to the local variable in the array of local variables of a method at index i. Note that the `this` JML expression in BML is encoded as $\mathbf{reg}(0)$. This is because the reference to the current object is stored at index 0 in the array of local variables.

Field identifiers in BML are encoded by the respective number in the constant pool table of the class file. For instance, the syntax for field access expressions in BML is $E_{bml}.ident$ which stands for the value in the field at index *ident* in the class constant pool of the class for the reference denoted by the expression E_{bml}.

A particular feature of BML is that it supports stacke expressions which do not have a counterpart in JML. These expressions are related to the way in which the virtual machine works, i.e. we refer to the stack and the stack counter. Because intermediate calculations are done by using the stack, often we will need stack expressions in order to characterise the states before and after an instruction execution. Stack expressions are represented in BML as follows:

- **cntr** represents the stack counter.
- $\mathbf{st}(E_{bml})$ stands for the element in the operand stack at position E_{bml}. For instance, the element below the stack top is represented with $\mathbf{st}(\mathbf{cntr} - 1)$ Note that those expressions may appear in predicates that refer to intermediate instructions in the bytecode.

---

as their semantics is exactly the same with the JML semantics

3.3.2 BML predicates

The properties that our bytecode language can express are from first order predicate logic. The formal grammar of the predicates is given by the nonterminal P_{bml}. From the formal syntax, we can notice that BML supports the standard logical connectors $\wedge, \vee, \Rightarrow$, existential $\exists$ and universal quantification $\forall$ as well as standard relation between the expressions of our language like $\neq, =, \leq, \leq \ldots$

3.3.3 Class Specification

Class specifications refer to properties that must hold in every visible state of a class. Thus, we have two kind of properties concerning classes: HERRRRRRRRRRRRRRRR

- **ClassInv**. Class invariants are predicates that must hold in every visible state of a class. This means that they must hold at the beginning and end of every method as well as whenever a method is called.
- **ClassHistoryConstr**.
- **declare ghost** *ident ident* declares a special specification variable which we call ghost variable. These variables do not change the program behaviour although they might be assigned to as we shall see later in this section. Ghost variables are used only for specification purposes and are not "seen" by the Java Virtual Machine.

3.3.4 Inter — method specification

In this subsection, we will focus on the method specification which is visible by the other methods in the program. We call this kind of method specification an inter method specification as it exports to the outside the method contracts. In particular, a method exports a precondition, a normal postcondition, a list of exceptional postconditions for every possible exception that the method may throw and the list of locations that it may modify. Those four components is one specification case, i.e. they describe a particular behaviour of the method, i.e. that if in the prestate of the method the specified precondition holds, then when the method terminates normally, the specified normal postcondition holds and if it terminates on an exception `E` then the specified exceptional postcondition for `E` will hold in the poststate of the method.

We also allow that a method might have several specification cases. Note that the specification cases that BML supports is actually the desugared version of the different behaviours of a method as well as its inherited specification. reference to JML desugaring

3.3.5 Method specification case

A specification case *specCase* consists of the following specification units:

- **requires** P_{bml} which represent the precondition of the specification case. If such a clause is not explicitely written in the specification, then the default precondition *true* is implicite

- **ensures** P_{bml} which stands for the normal postcondition of the method in case the precondition held in the prestate. In case this clause is not written in the specification explicitely, then the default postcondition *true* must hold.

- **modifies** *list locations* which is the frame condition of the specification case and denotes the the locations that may be modified by the method if the precondition of this specification case holds in the prestate. This in particular means that a location that is not mentioned in the **modifies** clause may be modified. If the modifies clause is omitted, then the default modifies specification is **modifies everything**

- *exsuresList* is the list of the exceptional postconditions that should hold in this specification case. In particular, every element in the list of exceptional postconditions has the following structure **exsures** (*ident*) P_{bml}. Note that at index *ident* there is a constant which stands for some exception class `Exc`. The semantics of such a specification expression is that if the method containing the exceptional postcondition terminates on an exception of type `Exc` then the predicate denoted by P_{bml} must hold in the poststate. Note that the list of exceptional postcondition may be empty. Also the list of exceptional postconditions might not be complete w.r.t. exceptions that may be thrown by the method. In both cases, for every exception that might be thrown by the method for which no explicite exceptional postcondition is given, we take the default exceptional postcondition *false*

give the bytecode version of the example with heavy weight specification

3.3.6 Intra — method specification

As we can see from the formal grammar in subsection 3.2, BML allows to specify a property that must hold at particular program point inside a method body. The nonterminal which describes the grammar of assertions is *intraMethodSpec*. Let us see in detail what kind of specifications can be supported in BML:

- **atIndex** *nat* specifies the index of the instruction which identifies the instruction to which the specification refers.

- *assertion* specifies the property that must hold in every state that reaches the instruction at the index specified by **atIndex** *nat*. We allow the following local assertions:

  - *loopSpec* gives the specification of a loop. It has the following syntax:
    - **loopInv** P_{bml} where P_{bml} is the property that must hold whenever the corresponding loop entry instruction is reached during execution
    - **loopModif** *list loc* is the list of locations modified in the loop. This means that at the borders of every iteration (beginning

```
class C {
   int a ;

   /@*
     * invariant a > 0;
     * historyConstraint   old(a) >= a;
     *@/

   /@*
     * requires a > b;
     * modifies a;
     * ensures  a == \old(a) - b;
     * exsures (Exception) false;
     *
     * also
     * requires a <= b;
     * modifies nothing;
     * ensures  a == \old(a);
     * exsures (Exception) false;
     *@/
   public void decrease(int b) {
     if ( a > b) {
       a = a - b;
   }
 }
}
```

Figure 3: An example for a heavy weight specification in BML

and end), all the expressions not mentioned in the loop frame condition must have the same value.

* **loopDecreases** E_{bml} specifies the expression E_{bml} which guarantees loop termination. The values of E_{bml} must be from a well founded set (usually from **int** type) and the values of E_{bml} should decrease at every iteration

- **assert** P_{bml} specifies the predicate P_{bml} that must hold at the corresponding position in the bytecode
- **set** E_{bml} E_{bml} is a special expression that allows to set the value of a specification ghost variable. This means that the first argument must denote a reference to a ghost variable, while the second expression is the new value that this ghost variable is assigned to.

3.3.7 Frame conditions

As we already saw, method or loop specifications might declare the locations that are modified by the method / loop. We use the same syntax in both of the cases where the modified expressions for methods or loops are specified with **modifies** *list locations*;. The semantics of such a specification clause is that all the locations that are not mentioned in the **modifies** list must be unchanged. The syntax of the expressions that might be modified by a method is determined by the nonterminal *locations*. We now look more closely what a modified expression can be:

- $E_{bml}.ident$ states that the method / loop modifies the value of the field at index *ident* in the constant pool for the reference denoted by E_{bml}
- **reg**(i) states that the local variable may modified by a loop. Note that this kind of modified expression makes sense only for expressions modified in a loop. However a modification of a local variable does not make sense for a method frame condition, as methods in Java are called by value, and thus, a method can not cause a modification of a local variable that is observable by the rest of the program.
- $arrayModAt(E_{bml}, specIndex)$ states that the components at the indexes specified by *specIndex* in the array denoted by E_{bml} may be modified. The indexes of the array components that may be modified *specIndex* have the following syntax:
  - i is the index of the component at index i. For instance, $arrayModAt(E_{bml}, i)$ means that the array component at index i might be modified. Of course, in order that such a specification make sense the following must hold: $0 \leq i < \mathrm{arrLength}(E_{bml})$
  - all specifies that all the components of the array may be modified, i.e. the expression $arrayModAt(E_{bml}, \mathrm{all})$ is a syntactic sugar for

    $$\forall\ i, 0 \leq i < \mathrm{arrLength}(E_{bml}) \Rightarrow arrayModAt(E_{bml}, i)$$

  - $i_1..i_2$ specifies the interval of array components between the index i_1 and i_2. Thus, the modified expression $arrayModAt(E_{bml}, i_1..i_2)$ is a syntactic sugar for

    $$\forall\ i, i_1 \leq i \wedge i \leq i_2 \Rightarrow arrayModAt(E_{bml}, i)$$

    Here, once again the following conditions must hold, otherwise the expression does not make sense :

    $$\begin{array}{l} 0 \leq i_1 \\ i_2 < \mathrm{arrLength}(E_{bml}) \end{array}$$

- **everything** states that every location might be modified by the method / loop
- **nothing** states that no location might be modified by a method / loop

4 Well formed BML specification

In the previous Section 3, we gave the formal grammar of BML. However, we are interested in a strict subset of the specifications that can be generated from this grammar. In particular, we want that a BML specification is well typed and respects few structural constraints.

Let's see few examples of type constraints that a valid BML specification must respect :

- the array expression $\mathbf{arrayAccess}(E^1_{bml}, E^2_{bml})$ must be such that E^1_{bml} is of array type and E^2_{bml} is of integer type
- the field access expression $E_{bml}.ident$ is such that E_{bml} is of subtype of the class where the field described by the constant pool element at index *ident* is declared
- For any expression $E^1_{bml}\, op\, E^2_{bml}$, E^1_{bml} and E^2_{bml} must be of a numeric type
- ...

Example for structural constraint are :

- All references to the constant pool must be to an entry of the appropriate type. For example: the field access expression $E_{bml}.ident$ is such that the *ident* must reference a field in the constant pool; or for the expression **\type(***ident***)**, *ident*must be a reference to a constant class in the constant pool
- every *ident* in a BML specification must be a correct index in the constant pool table.

Actually, an extension of the bytecode verifier may perform the checks if a BML specification respects this kind of structural and type constraints. However, we are not going farther in this subject as it is out of the scope of the present thesis. For the curious reader, it will be certainly of interest to turn to the Java Virtual Machine specification [9] which contains the official specification of the Java bytecode verifier or to the existing literature on bytecode verification (see the overview article [**?**])

5 Compiling JML into BML

We now turn to explaining how JML specifications are compiled into user defined attributes for Java class files. As we shall see, the compilation consists of several phases where in the final phase The JVMS allows to add to the class file user specific information([9], ch.4.7.1). This is done by defining user specific attributes (their structure is predefined by JVMS). Thus the "JML compiler" [3]

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

compiles the JML source specification into user defined attributes. The compilation process has the following 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 [9], 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. Desugaring of the JML specification
   BML supports less specification clauses than JML for the sake of keeping compact the class file format. In particular BML does not support heavy weight behaviour specification clauses or nested specification, neither an incomplete method specification(see [7]). Thus, a step in the compilation of JML specification into BML specification is the desugaring of the JML heavy weight behaviours and the expanding of a light - weight non complete specification into its full default format. This corresponds to the standard JML desugaring as described in [10] For instance, a Java method which has two normal behaviours is given in Fig. **??**. Its desugared form corresponds to the method given in Fig. 3

3. Linking with source data structures
   When the JML specification is desugared, we are ready for the linking and resolving phases. In this stage, the JML specification gets into an intermediate format in which the identifiers are resolved to data structures standing for the data that it represents. For instance, consider once again the example in Fig. **??** and particularly, let's look at the first specification case of method `m` whose precondition `a`¿ b contains the identifier `a`. In the linking phase, this identifier is resolved to the field named `a` which is declared in the same class as shown in the figure. Also in this precondition, the identifier `b` which is resolved to the parameter of method `m`.

4. Compilation of the JML specification into BML
   In this stage, the desugared JML specification from the source file is compiled into BML specification. The Java and JML source identifiers are linked with their identifiers on bytecode level, namely with the corresponding indexes either from the constant pool or the array of local variables described in the **Local_Variable_Table** attribute. If, in the JML specification a field identifier appears for which no constant pool (cp) index exists, it is added in the constant pool and the identifier in question is compiled to the new cp index. It is also in this phase that the specification parts like the loop invariants and the assertions which should hold at

$$
\begin{array}{l}
\backslash\mathbf{result} = 1 \\
\\
\Longleftrightarrow \\
\exists \mathbf{bv\_0}, \left(\begin{array}{l} 0 \leq \mathbf{bv\_0} \wedge \\ \mathbf{bv\_0} < len(\#19(\mathbf{reg}(0))) \wedge \\ \mathbf{arrayAccess}(\#19(\mathbf{reg}(0)), \mathbf{bv\_0}) = \mathbf{reg}(1) \end{array} \right)
\end{array}
$$

Figure 4: THE COMPILATION OF THE POSTCONDITION IN FIG. 1

a certain point in the source program must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. The compilation of an expression is a tag followed by the compilation of its subexpressions.

Another important issue in this stage of the JML compilation is how the type differences on source and bytecode level are treated. By type differences we refer to the fact that the JVM (Java Virtual Machine) does not provide direct support for integral types like byte, short, char, neither for boolean. Those types are rather encoded as integers in the bytecode. Concretely, this means that if a Java source variable has a boolean type it will be compiled to a variable with an integer type. For instance, in the example for the method `isElem` and its specification in Fig.1 the postcondition states the equality between the JML expression **result** and a predicate. This is correct as the method `isElem` in the Java source is declared with return type boolean and thus, the expression **result** has type boolean. Still, the bytecode resulting from the compilation of the method `isElem` returns a value of type integer. This means that the JML compiler has to "make more effort" than simply compiling the left and right side of the equality in the postcondition, otherwise its compilation will not make sense as it will not be well typed. Actually, if the JML specification contains program boolean expressions that the Java compiler will compile to bytecode expression with an integer type, the JML compiler will also compile them in integer expressions and will transform the specification condition in equivalent one[4].

Finally, the compilation of the postcondition of method `isElem` is given in Fig. 4. From the postcondition compilation, one can see that the expression **result** has integer type and the equality between the boolean expressions in the postcondition in Fig.1 is compiled into logical equivalence. The example also shows that local variables and fields are respectively linked to the index of the register table for the method and to the corresponding index of the constant pool table (#19 is the compilation of the field name `list` and **reg**(1) stands for the method parameter `obj`).

[4]when generating proof obligations we add for every source boolean expression an assumption that it must be equal to 0 or 1. Actually, a reasonable compiler will encode boolean values in this way

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

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

Figure 5: Structure of the Loop Attribute

5. Encoding BML specification into user defined attributes
   Method specifications, class invariants, loop invariants are newly defined attributes in the class file. For example, the specifications of all the loops in a method are compiled to a unique method attribute whose syntax is given in Fig. 5. This attribute is an array of data structures each describing a single loop from the method source code. Also for each loop in the source code there must be a corresponding element in the array. More precisely, every element contains information about the instruction where the loop starts as specified in the **Line_Number_Table**, the locations that can be modified in a loop iteration, the invariant associated to this loop and the decreasing expression in case of total correctness,

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

References

[1] AV, Sethi R, and Ullman JD. *Compilers-Principles, Techniques and Tools.* Addison-Wesley: Reading, 1986.

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

[3] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 2004. To appear.

[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. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 2003)*, volume 80 of *ENTCS*. Elsevier, 2003.

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

[6] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java modeling language. In *Software Engineering Research and Practice (SERP'02)*, CSREA Press, pages 322–328, June 2002.

[7] G.T.Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual.* technical report.

[8] R.K. Leino. escjava. `http://secure.ucd.ie/products/opensource/ESCJava2/docs.html`.

[9] Tim Lindholm and Frank Yellin. Java virtual machine specification. Technical report, Java Software, Sun Microsystems, Inc., 2004.

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