Bytecode Verification and its Applications

June 6, 2006

Contents

Chapter 1

Specification language for Java bytecode programs

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 / can 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 / 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 1.1, we give an overview of the main features of JML. A detailed overview of BML is given in section 1.2. 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 1.5. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

1.1 A quick overview of JML

JML [7] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications. JML follows the design-by-contract approach (see [2]), where classes are annotated with class invariants and method with pre- and postconditions. Specification inside methods is also possible; for example one can specify loop invariants, or assertions that must hold at specific program points.

Over the last few years, JML has become the de facto specification language for Java source code programs. Different tools exist to verify or generate JML specifications (see for an overview [4]). 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. Specifications are written using preconditions, postcondition, class invariants and other annotations, where the different predicates are side-effect free Java expressions, extended with specification-specific keywords (*e.g.* logical quantifiers and a keyword to refer to the return value of a method). Other important factors for the success of JML are its expressiveness and flexibility.

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 one has to use the keywords requires and ensures respectively, modifies keyword is followed by all the locations that can be modified by the method, loop_invariant, not surprisingly, stands for loop invariants, 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.

JML can be used for either static checking of Java programs by tools such as JACK, the Loop tool, ESC/Java [8] or dynamic checking by tools such as the assertion checker jmlrac [6]. An overview of the JML tools can be found in [4].

Figure 1.1 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.

```
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 1.1: class `ListArray` with JML annotations

1.2 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 Def. 1.2.2 gives the formal grammar of BML. The formal grammar of BML is given in the next definition.

1.2.1 Notation convention

- Nonterminals are written with a *italics* font
- Terminals are written with a **boldface** font
- Keywords are writtem with a sans serif font

- brackets [ ] surround optional text.

1.2.2 BML Grammar

$$
\begin{array}{ll}
\textit{constants} & ::= \textit{intLiteral} \mid \textit{signedIntLiteral} \mid \mathsf{null} \mid \textit{ident} \\
\textit{signedIntLiteral} & ::= +\textit{nonZerodigit}[\textit{digits}] \mid -\textit{nonZerodigit}[\textit{digits}] \\
\textit{intLiteral} & ::= \textit{digit} \mid \textit{nonZerodigit}[\textit{digits}] \\
\textit{digits} & ::= \textit{digit}[\textit{digits}] \\
\textit{digit} & ::= \mathbf{0} \mid \textit{nonZerodigit} \\
\textit{nonZerodigit} & ::= \mathbf{1} \mid \ldots \mid \mathbf{9} \\
\textit{ident} & ::= \#\ \textit{intLiteral} \\
\textit{boundVar} & ::= \#\ \textit{bv\_intLiteral} \\
\\
\mathcal{E} & ::= \textit{constants} \\
 & \mid \mathrm{reg}(\textit{digits}) \\
 & \mid \mathcal{E}\textbf{.}\textit{ident} \\
 & \mid \textit{ident} \\
 & \mid \textit{arrayAccess}(\mathcal{E}, \mathcal{E}) \\
 & \mid \mathcal{E}\ \textit{op}\ \mathcal{E} \\
 & \mid \textbf{cntr} \\
 & \mid \textbf{st}(\mathcal{E}) \\
 & \mid \backslash\mathsf{typeof}(\mathcal{E}) \\
 & \mid \backslash\mathsf{type}(\textit{ident}) \\
 & \mid \backslash\mathsf{elemtype}(\mathcal{E}) \\
 & \mid \backslash\mathsf{old}(\mathcal{E}) \\
 & \mid \mathtt{EXC} \\
 & \mid \backslash\mathsf{result} \\
 & \mid \textit{boundVar} \\
\\
\textit{op} & ::= + \mid \textbf{-} \mid \textbf{mult} \mid \textbf{div} \mid \textbf{rem} \\
\\
\mathcal{R} & ::= ==\mid\neq\mid\leq\mid\leq\mid\geq\mid>\mid<:
\end{array}
$$

$\mathcal{F}^{bc}$ ::= $\mathcal{E}_1\ \mathcal{R}\ \mathcal{E}_2$
| *true*
| *false*
| *not* $\mathcal{F}^{bc}$
| $\mathcal{F}^{bc} \wedge \mathcal{F}^{bc}$
| $\mathcal{F}^{bc} \vee \mathcal{F}^{bc}$
| $\mathcal{F}^{bc} \Rightarrow \mathcal{F}^{bc}$
| $\forall\ boundVar, \mathcal{F}^{bc}$
| $\exists\ boundVar, \mathcal{F}^{bc}$

classSpec ::= ClassInv $\mathcal{F}^{bc}$
| ClassHistoryConstr $\mathcal{F}^{bc}$
| declare ghost *ident ident*

intraMethodSpec ::= atIndex *nat*;
assertion;

assertion ::= *loopSpec*
| assert $\mathcal{F}^{bc}$
| set $\mathcal{E}\ \mathcal{E}$

loopSpec ::= loopInv $\mathcal{F}^{bc}$;
loopModif *list*;
loopDecreases $\mathcal{E}$;

methodSpec ::= *specCase*
| *specCase* also *methodSpec*

specCase ::= requires $\mathcal{F}^{bc}$;
modifies *list locations*;
ensures $\mathcal{F}^{bc}$;
exsuresList

exsuresList ::= [] | exsures (*ident*) $\mathcal{F}^{bc}$; *exsuresList*

locations ::= $\mathcal{E}$**.***ident*
| reg(i)
| *arrayModAt*($\mathcal{E}$, *specIndex*)
| **everything**
| **nothing**

specIndex ::= all | $i_1..i_2$ | i

$$
\begin{array}{rl}
bmlKeyWords & ::= \text{requires} \\
& \mid \text{ensures} \\
& \mid \text{modifies} \\
& \mid \text{assert} \\
& \mid \text{set} \\
& \mid \text{exsures} \\
& \mid \text{also} \\
& \mid \text{ClassInv} \\
& \mid \text{ClassHistoryConstr} \\
& \mid \text{atIndex} \\
& \mid \text{loopInv} \\
& \mid \text{loopDecreases} \\
& \mid \text{loopModif} \\
& \mid \backslash \text{ typeof} \\
& \mid \backslash \text{ elemtype} \\
& \mid \text{TYPE} \\
& \mid \backslash\text{result}
\end{array}
$$

1.2.3 Interpretation of the BML grammar

In the following, we will discuss informally the interpretation of the syntax structures of BML.

BML expressions

Most of the expressions supported in BML have their counterpart in JML. However there are few constructs that do not have analogs in JML. As we will see hereafter, BML allows to express field access, array access, method parameters and local variables, stack expressions etc. Let's look now in more detail at the BML expressions that can be translated in JML and viceversa.

- *constants* represents the constants in BML. A constant is either a signed or unsigned integer, or an identificator. Integers and identificators are defined as usually. Identificators correspond to indexes in the constant pool of a Java class.

- reg(i) a local variable in the array of local variables of a method at index i. Note that the array of local variables of a method on bytecode level is the list of formal parameters of the variables declared locally in the method. This is slightly different from the Java language where difference is made between method parameters and variables declared locally to a method.

- $\mathcal{E}.ident$ stands for accessing the field which is at index *ident* in the class constant pool. for the reference denoted by the expression $\mathcal{E}$.

- $arrayAccess(\mathcal{E}_1, \mathcal{E}_2)$ stands for an access to the element at index $\mathcal{E}_2$ in the array denoted by the expression $\mathcal{E}_1$. This corresponds to the Java notation $\mathcal{E}_1[\mathcal{E}_2]$
- $\mathcal{E}$ *op* $\mathcal{E}$ stands for the usual arithmetic operations. *op* ranges over the standard arithmetic operations $+, -, *, div, rem$
- $\backslash\mathsf{typeof}(\mathcal{E})$ denotes the dynamic type of the expression $\mathcal{E}$
- $\backslash\mathsf{type}(ClassName)$ denotes the class with class name $ClassName$
- $\backslash\mathsf{elemtype}(\mathcal{E})$ denotes the type of the elements of the array $\mathcal{E}$
- $\backslash\mathsf{old}(\mathcal{E})$ denotes the value of $\mathcal{E}$ in the pre state of a method. This expression is usually used in the postcondition of a method and thus, allows that the postcondition predicate relate to the prestate
- `EXC` is a special specification identifier which denotes the thrown exception object in exceptional postconditions

The expressions that cannot be translated in JML 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. Let's see how stack expressions are represented in BML:

- **cntr** represents the stack counter.
- $\mathbf{st}(\mathcal{E})$ stands for the element in the operand stack at position $\mathcal{E}$ Differently from the JML, our bytecode specification language has to take into account the operand stack and its counter. Of course, those expressions may appear in predicates that refer to intermediate instructions in the bytecode. For instance, the element below the stack top is represented with $\mathbf{st}(\mathbf{cntr} - 1)$

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 $\mathcal{F}^{bc}$. 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 $\neq, =, \leq, \leq \ldots$

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:

```
class C {
   int a ;

   invariant a > 0;
   historyConstraint a >= old(a);

   public void decrease(int b) {
      ...
   }
}
```

Figure 1.2: An example for class specification

- 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. Class history constraints is a property which states a relation between the pre state and poststate of every method in the corresponding class.
- 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.

We give in Fig.1.2 an example of a class specification in Java source code. Note, that we give these examples on source code for the sake of clarity. The specification from the example declares one invariant which states that the field `a` must always be greater than `0`. This means for instance, that whenever the method `decrease` is called the invariant must hold and when the method terminates execution the invariant once again should hold. In the example we also have specified a history constraint which states that the value of the instance variable `a` in the prestate of any method of class `C` must be greater or equal to its value in the state when the method terminates execution. For instance, the history constraint is established by the method `decrease`

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

Method specification case

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

- requires $\mathcal{F}^{bc}$ 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 $\mathcal{F}^{bc}$ 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*) $\mathcal{F}^{bc}$. 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 $\mathcal{F}^{bc}$ 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*

If a method has only one specification case this means that the precondition of the method is always the precondition of the unique specification case. If a method has several specification cases then, when the method is invoked at least the precondition of one of the specification cases must hold. If the precondition of a particular specification case holds in the prestate this requires that in the

```
class C {
   int a ;

   invariant a > 0;
   historyConstraint a >= old(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 1.3: An example for an intra method specification

poststate the postcondition of the same specification case holds and only the locations mentioned in the frame condition of this specification case may be modified during method execution. For instance, the example in Fig. 1.3 shows the method `decrease` which is now specified with two specification cases. The specification cases describe two different behaviours of the method. The first specification case states that if the method is invoked with parameter smaller than the instance variable `a` then the method will modify `a` by decreasing it with the value of the parameter `b`. The other specification case describes the behavior of the method in case the actual parameter of the method is greater than the instance variable `a`.

Intra — method specification

As we can see from the formal grammar in subsection 1.2.2, 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. We would like to note here that the style of specification in BML is slightly different from the JML style. First, JML specification is written directly in the source code in comments at the point in the program text where the specification must hold. Second, as the Java source language is structured, JML allows to specify a particular program structure. For instance, in Fig. 1.1 the reader may notice that the loop specification refers to the control structure which follows after the specification and which corresponds to the loop. However, on bytecode level we could not write ditrectly in the bytecode of a method body, as this will corrupt the performance of any standard Java Virtual Machine. That's why specification is written outside the bytecode text and contains also information about the instruction to which the specification refers. Then, as bytecode does not have control structures specification will always refer to a particular instruction in the bytecode. For instance, loops on bytecode are identified by a unique loop entry instruction and thus, a loop invariant must hold basically every time the corresponding loop entry instruction is reached.
- *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 $\mathcal{F}^{bc}$ where $\mathcal{F}^{bc}$ 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 and end), all the expressions not mentioned in the loop frame condition must have the same value.
    - loopDecreases $\mathcal{E}$ specifies the expression $\mathcal{E}$ which guarantees loop termination. The values of $\mathcal{E}$ must be from a well founded set (usually from int type) and the values of $\mathcal{E}$ should decrease at every iteration
  - assert $\mathcal{F}^{bc}$ specifies the predicate $\mathcal{F}^{bc}$ that must hold at the corresponding position in the bytecode
  - set $\mathcal{E}$ $\mathcal{E}$ 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.

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:

- $\mathcal{E}$.*ident* states that the method / loop modifies the value of the field at index *ident* in the constant pool for the reference denoted by $\mathcal{E}$
- reg(i) states that the local variable may modified by a loop. Note that this kind of modified expression 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. However, for loops this is not the case.
- $arrayModAt(\mathcal{E}, specIndex)$ states that the components at the indexes specified by *specIndex* in the array denoted by $\mathcal{E}$ 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, $arrayModAt(\mathcal{E}, i)$ 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: $0 \leq i < \text{arrLength}(\mathcal{E})$
  - all specifies that all the components of the array may be modified, i.e. the expression $arrayModAt(\mathcal{E}, \text{all})$ is a syntactic sugar for

    $$\forall\, i, 0 \leq i < \text{arrLength}(\mathcal{E}) \Rightarrow arrayModAt(\mathcal{E}, i)$$

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

    $$\forall\, i, i_1 \leq i \wedge i \leq i_2 \Rightarrow arrayModAt(\mathcal{E}, i)$$

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

    $$0 \leq i_1$$
    $$i_2 < \text{arrLength}(\mathcal{E})$$

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

1.3 BML type checker

In the previous section, we described the BML syntax. However, we are going to accept a strict subset of the specifications that can be written in BML. In particular, we define here a typechecker for the BML specifications

to be continued ...

1.4 Background information of the class file format

This section is not obligatory if the reader is familiar with the class file format. However, for those who do not know much about it, this chapter is useful for understanding the JML compiler presented in the next section 1.5.

In particular, our specification compiler will make use of the **Constant_pool_table** data structure as well as of several optional data structures contained in the class file: the **Local_variable_table** and the **Line_Number_Table**. In what follows, we give a consize description of these class file data structures.

1.4.1 The Constant_pool_table

Recall that a class file defines a single class or interface and contains information about the class name, interfaces implemented by the class, super class, methods and fields declared in the class and references. The JVM [9] mandates that the class file contains data structure usually referred as the **Constant_pool_table** which is used to construct the runtime constant pool upon class or interface creation. The runtime constant pool serves for loading, linking and resolution of references used in the class.

Thus, the **Constant_pool_table** contains elements describing :

- every field which is dereferenced in any of the methods of the current class. The corresponding data structure which stands for a particular field constant reference is **CONSTANT_Fieldref_info**. Its structure is given in Fig.1.4. The figure shows that a constant field reference data structure contains information about the class or interface where the field is declared (the second element of the data structure, **class_index**) as well a description of its name and type (the field **name_and_type_index**)
- every class referenced in the current class. A class constant is stored in the constant pool in a **CONSTANT_Class_info** data structure
- every method which is called in any of the methods of the current class. It is represented by a **CONSTANT_Method_info** data structure
- every string constant that appears in the current class file.
- etc...

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

- **tag** is a tag of one byte whose value is 7 and determines without ambiguity that the current attribute describes a field reference constant
- **class_index** The value of this item must be a valid index into the **Constant_pool_table**table. The **Constant_pool_table**entry at that index must be a **CONSTANT_Class_info** structure representing the class or interface type that contains the declaration of the field or method.
- **name_and_type_index** The value of the item must be a valid index into the **Constant_pool_table**table. The **Constant_pool_table**entry at that index must be a **CONSTANT_NameAndType_info** (see for more detailed explanation [9], section 4.4) structure. This **Constant_pool_table**entry indicates the name and descriptor of the field.

Figure 1.4: STRUCTURE OF THE **CONSTANT_Fieldref_info** ATTRIBUTE

1.4.2 Representation of method in the class file format

Each method, including each instance initialization method and the class or interface initialization method, is described by a special data structure called **method_info**. Every **method_info** structure is supplied with obligatory and optional attributes. For instance, an obligatory attribute is the data structure called **Code**. A **Code** attribute contains the Java virtual machine instructions and auxiliary information for a single method, instance initialization method, or class or interface initialization method. The **Code** attribute has a list of optional attributes which should be ignored by the JVM but which can be used by other tools, as for instance debuggers. The JVM specification defines two attributes — the **Local_variable_table** and the **Line_Number_Table**, which may appear as attributes in the **Code** data structure. We give hereafter a description of the last two data structures as they will play a role in the JML2BML compilation process.

The Local_variable_table attribute

The **Local_variable_table** attribute is an optional variable-length attribute of a method **Code** attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. There may be no more than one **Local_variable_table** attribute per local variable in the Code attribute.

give the data structure and explain what are the assumptions?

The Line_Number_Table attribute

The LineNumberTable attribute is an optional variable-length attribute in the attributes table of a method **Code** attribute. It may be used by debuggers to determine which part of the Java virtual machine code array corresponds to a given line number in the original source file. If **Line_Number_Table** attributes are present in the attributes table of a given **Code** attribute, then they may appear in any order. Furthermore, multiple **Line_Number_Table** attributes may together represent a given line of a source file; that is, **Line_Number_Table** attributes need not be one-to-one with source lines.

1.5 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" [1] 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. 1.5. Its desugared form corresponds to the method given in Fig. 1.3

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

```
class C {
   int a ;

   invariant a > 0;
   historyConstraint a >= old(a);

      /*@ public_behaviour
        @ requires a > b;
        @ modifies a;
        @ ensures  a == \old(a) - b;
        @
        @ also
        @ requires a <= b;
        @ ensures  a == \old(a);
        @*/
       public void decrease(int b) {
          if ( a > b) {
          a = a - b;
      }
   }
}
```

Figure 1.5: An example for a method with two normal behaviours specified in JML

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. 1.5 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 specifica-

tion parts like the loop invariants and the assertions which should hold at a certain point in the source program must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. The compilation of an expression is a tag followed by the compilation of its subexpressions.

Another 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.1.1 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[2].

Finally, the compilation of the postcondition of method `isElem` is given in Fig. 1.6. 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.1.1 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`).

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. 1.7. 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

[2]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

$$\backslash\mathsf{result} = 1$$

$$\Longleftrightarrow$$

$$\exists var(0).\left(\begin{array}{l} 0 \leq var(0) \wedge \\ var(0) < len(\#19(reg_0)) \wedge \\ \#19(reg_0)[var(0)] = reg_1 \end{array}\right)$$

Figure 1.6: The compilation of the postcondition in Fig. 1.1

```
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 1.7: Structure of the Loop Attribute

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 for the compiler. 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.

Bibliography

[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.