Java ByteCode Specification Language(BCSL) and how to compile JML to BCSL

Mariela Pavlova

March 10, 2006

1 Introduction

This document is an overview of a bytecode level specification language, called for short BCSL and a compiler from a subset of the high level Java specification language JML to BCSL. BCSL 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. Before going further, we discuss what advocates the need of a low level specification. Traditionally, specification languages were tailored for high level languages. Source specification allows to express functional or security properties about a program and 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 but 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 BCSL language and a compiler from JML towards BCSL.

In what follows subsection 1.1 introduces the basic features of JML, subsection 2 gives the formal grammar of the specification language and subsection 3 describes the compilation process from JML to BCSL. 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 pre- and postconditions. Specification inside methods is also possible; for example one can specify loop invariants, or assertions predicates that must hold at specific program points.

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 Java 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 [4]. 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 [5]. An overview of the JML tools can be found in [3].

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.

```
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

2 Features of BCSL

BCSL corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties.

Specification clauses in BCSL that are taken from JML and inherit their semantics directly from JML include:

- class specification, i.e. class invariants and history constraints

- ghost variables, which are special specification variables not seen by the virtual machine, used by tools that support BCSL, e.g. a verification condition generator for Java bytecode. Those variables can be assigned by using the special BCSL operator set.
- method specification cases. Every method specification case specifies a method precondition, normal and exceptional postconditions, method frame conditions (the locations that may be modified by the method).
- inter method specification, for instance loop invariants
- predicates from first order logic
- expressions from the programming language, like field access expressions, local variables, etc.
- specification operators. For instance *old(*E*)* which is used in method postconditions and designates the value of the expression E in the prestate of a method, *result* which stands for the value the method returns if it is not void

BCSL has few particular extra features that JML lacks :

- loop frame condition, which declares the locations that can be modified during a loop iteration. We were inspired for this by the JML extensions in JACK [4]
- stack expressions - *cntr* which stands for the stack counter and *st*(`ArithmeticExpr`) standing for a stack element at position `ArithmeticExpr` . These expressions are needed in BCSL as the Java Virtual Machine (JVM) is stack based.

The formal grammar of BCSL is given in Fig. 2

3 Compiling JML into bytecode specification language

We now turn to explaining how JML specifications are compiled into user defined attributes for Java class files. Recall that a class file defines a single class or interface and contains information about the class name, interfaces implemented by the class, super class, methods and fields declared in the class and references. The Java Virtual Machine Specification (JVMS) [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. 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 three stages:

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

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. Compilation of the JML specification from the source file and the resulting class file. In this phase, Java and JML source identifiers are linked with their identifiers on bytecode level, namely with the corresponding indexes either from the 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 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 intergral 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 $\backslash result$ and a predicate. This is correct as the method `isElem` in the Java source is declared with return type boolean and thus, the expression $\backslash result$ 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. 3. From the postcondition compilation, one can see that the expression $\backslash result$ has integer type and the equality between the boolean

[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

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`).

3. add the result of the JML compilation in the class file as 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. 4. 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 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.

A Specification of the Bytecode Specification Compiler

A.1 Class annotation

The following attributes can be added (if needed) only to the array of attributes of the `class_info` structure.

A.1.1 Ghost variables

Ghost_Field_attribute {

```
    u2 attribute_name_index;
    u4 attribute_length;
    u2 fields_count;
    {  u2 access_flags;
       u2 name_index;
       u2 descriptor_index;
    } fields[fields_count];
}
```

attribute_name_index
The value of the attribute_name_index item must be a valid index into the constant_pool table . The constant_pool entry at that index must be a CONSTANT_Utf8_info structure representing the string "Ghost_Field".

attribute_length
the length of the attribute in bytes = 2 + 6*fields_count.

access_flags
The value of the access_flags item is a mask of modifiers used to describe access permission to and properties of a field.

name_index
The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info structure which must represent a valid Java field name stored as a simple (not fully qualified) name, that is, as a Java identifier.

descriptor_index
The value of the descriptor_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8 structure which must represent a valid Java field descriptor.

A.1.2 Class invariant

```
JMLClassInvariant_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    formula attribute_formula;
}
```

attribute_name_index
The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info structure representing the string "ClassInvariant".

attribute_length
the length of the attribute in bytes - 6.

attribute_formula
code of the formula that represents the invariant, see (A.4) for formula grammar

A.1.3 History Constraints

```
JMLHistoryConstraints_attribute {
    u2 attribute_name_index;
```

```
    u4 attribute_length;
    formula attribute_formula;
}
```

attribute_name_index
The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info structure representing the string "Constraint".

attribute_length
the length of the attribute in bytes - 6.

attribute_formula
code of the formula that is a predicate of the form P$state, old(state)$ that establishes relation between the prestate and the postate of a method execution. see (A.4) for formula grammar

A.2 Method annotation

A.2.1 Method specification

The JML keywords `requires, ensures, exsures` will be defined in a newly attribute in Java VM bytecode that can be inserted into the structure `method_info` as elements of the array `attributes`.

```
JMLMethod_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    formula requires_formula;
    u2 spec_count;
    {   formula spec_requires_formula;
        u2 modifies_count;
        formula modifies[modifies_count];
        formula ensures_formula;
        u2 exsures_count;
        {   u2 exception_index;
            formula exsures_formula;
        } exsures[exsures_count];
    } spec[spec_count];
}
```

attribute_name_index
The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info structure representing the string "MethodSpecification".

attribute_length
The length of the attribute in bytes.

requires_formula
The formula that represents the precondition (in the subsection see Formulas)

spec_count
The number of specification case.

spec[]
Each entry in the spec array represents a case specification. Each entry must contain the following items:

spec_requires_formula
The formula that represents the precondition (in the subsection see Formulas)

modifies_count
The number of modified variable.

modifies[]
The array of modified formula.

ensures_formula
The formula that represents the postcondition (in the subsection see Formulas)

exsures_count
The number of exsures clause.

exsures[]
Each entry in the exsures array represents an exsures clause. Each entry must contain the following items:

exception_index
The index must be a valid index into the constant_pool table. The constant_pool entry at this index must be a CONSTANT_Class_info structure representing a class type that this clause is declared to catch.

exsures_formula
The formula that represents the exceptional postcondition (in the subsection see Formulas)
Note:
if the exsures clause is of the form:
`exsures` (`Exception_name` e) P(e) it is first transformed in : `exsures Exception_name` P(e)[e $\leftarrow$ EXCEPTION], where EXCEPTION is a special keyword for the specification language, for which in JML there is no correspondent one.

A.2.2 Set

These are particular assertions that assign to model fields.

Assert_attribute {

u2 attribute_name_index;
u4 attribute_length;
u2 set_count;
{ u2 index;
expression e1;
expression e2;
} set[set_count];
}

attribute_name_index
The value of the attribute_name_index item must be a valid index into the `constant_pool table` . The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "Set".

attribute_length
The length of the attribute in bytes.
set_count
The number of set statement.

set[]
Each entry in the set array represents a set statement. Each entry must contain the following items:

index
The index in the bytecode where the **assignment** is done.

e1
the expression to which is assigned a value. It must be a JML expression, i.e. a JML field, or a dereferencing a field of JML reference object an assignment expression see (**??**)

e2
the expression that is assigned as value to the JML expression

A.2.3 Assert

Assert_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 assert_count;
{ u2 index;
formula predicate;
} assert[assert_count];
}

attribute_name_index
The value of the attribute_name_index item must be a valid index into the `constant_pool table` . The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "Assert".

attribute_length

The length of the attribute in bytes.

assert_count

The number of assert statement.

assert[]

Each entry in the assert array represents an assert statement. Each entry must contain the following items:

index

The index in the bytecode where the **predicate** must hold

predicate

the predicate that must hold at index **index** in the bytecode ,see (A.4)

A.2.4 Loop specification

```
JMLLoop_specification_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 loop_count;
    {   u2 index;
        u2 modifies_count;
        formula modifies[modifies_count];
        formula invariant;
        expression decreases;
    } loop[loop_count];
}
```

attribute_name_index

The value of the attribute_name_index item must be a valid index into the `constant_pool table`. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "Loop_Specification".

attribute_length

The length of the attribute in bytes

loop_count

The length of the array of loop specifications

index

The index of the instruction in the bytecode array that corresponds to the entry of the loop

modifies_count

The number of modified variable.

modifies[]

The array of modified expressions.

invariant

The predicate that is the loop invariant. It is a formula written in the grammar specified in the section Formula ,see (A.4)

decreases
The expression whose decreasing after every loop execution will guarantee loop termination

A.2.5 Block specification

Here also the `LineNumberTable` attribute must be present.

Block_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 start_index;
u2 end_index;
formula precondition;
u2 modifies_count;
formula modifies[modifies_count];
formula postcondition;
}

attribute_name_index
The value of the attribute_name_index item must be a valid index into the `constant_pool table`. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` structure representing the string "_specification ".

attribute_length
The length of the attribute in bytes - 6, i.e. equals `n+m`.

start_index
The index in the `LineNumberTable` where the beginning of the block is described
end_index
The index in the `LineNumberTable` where the end of the block is described
precondition
The predicate that is the precondition of the block ,see (A.4)
modifies_count
The number of modified variable.

modifies[]
The array of modified formula.

postcondition
the predicate that is the postcondition of the block ,see (A.4)

A.3 Formula and Expression compiler function

The compiler function is denoted with $\ulcorner \urcorner_{\texttt{context}}$. It is defined inductively over the grammar of the specification language as defined in Section 2 and more particularly on Fig. **??**. The compiling function depends on the context `context` and in particular it is important for compiling field references and method call expressions. The context is the class where the method or field is declared. For example when compiling the fully qualified name `a.b` the two subexpressions `a` and `b` are compiled one after another. The subexpression `a` will be compiled in the context of `this` type ,i.e. in the context of the class where this expression appears and the subexpression `b` will be compiled in the context of the class of the subexpression `a` as it is a field of the class of the subexpression `a`.

A.4 Formulas

A.4.1 Translation of formulas

$\ulcorner\texttt{Formula}\urcorner_{\texttt{context}} ::= \ulcorner\texttt{Connector}\urcorner \ulcorner\texttt{Formula}_1\urcorner_{\texttt{context}} \ldots \ulcorner\texttt{Formula}_\texttt{n}\urcorner_{\texttt{context}} \mid$
$\ulcorner\texttt{Quantifier}\urcorner\ulcorner\texttt{Formula}_1\urcorner_{\texttt{context}}$
$\ulcorner\texttt{PredicateSymbol}\urcorner \ulcorner\texttt{Expression}\urcorner_{\texttt{context}} \ldots \ulcorner\texttt{Expression}\urcorner_{\texttt{context}} \mid$
$\ulcorner\texttt{True}\urcorner \mid$
$\ulcorner\texttt{False}\urcorner$

Remark: n is coded in 1 byte.
Every quantification that contains a range , i.e. every formula of the form : $\forall\ A\ a; P(a); Q(a)$ should be transformed into $\forall\ A\ a; P(a) \Rightarrow Q(a)$

A.4.2 Predicate constants

Predicate	*code*
True	0x00
False	0x01

Codes for the predicate constants `True`, `False`

$\ulcorner\ \texttt{True}\urcorner_{\texttt{context}} ::= code(\texttt{True})$
$\ulcorner\ \texttt{False}\urcorner_{\texttt{context}} ::= code(\texttt{False})$

A.4.3 Logical connectors

Connector	*code*
$\wedge$	0x02
$\vee$	0x03
$\Rightarrow$	0x04
!	0x05

Codes for the `Connector` symbols

$\ulcorner\texttt{Connector}\urcorner_{\texttt{context}} ::= code(\texttt{Connector})$

A.4.4 Quantifiers

$\ulcorner\texttt{Quantifier}\urcorner ::= \ulcorner\texttt{Quantificatorsymbol}\urcorner\ (\ulcorner\text{Type}\urcorner_{\texttt{context}}\ \ulcorner\text{BoundVar}\urcorner_{\texttt{context}})_{\text{n}},$
where n is the number of bound variables.

A.4.5 Bound Variables

$\ulcorner$`BoundVar`$\urcorner_{\texttt{context}}$ = code(`BoundVar`) `int`
where `int` is a fresh integer value.

A.4.6 Quantificator symbols

`Quantificator symbol`	*code*
$\forall$	0x06
$\exists$	0x07

Codes for `Quantification symbols`

$\ulcorner$ `Quantification symbol` $\urcorner_{\texttt{context}}$ ::= code(`Quantification symbol`)
The code of any bound variable *ident* is a fresh variable coded in *1 byte* that must replace any occurrence of *ident* in the predicate coming after the quantification expression
The type Type is a `fully qualified name` expression.

A.4.7 Predicate symbols

`PredicateSymbol`	*code*
==	0x10
>	0x11
<	0x12
<=	0x13
>=	0x14
`instanceof`	0x15
<:	0x16

Codes for the `Predicate Symbols` symbols

$\ulcorner$`PredicateSymbol`$\urcorner_{\texttt{context}}$ ::= *code*(`PredicateSymbol`)

A.5 Expressions

Here the grammar for well formed expressions is described. We use the prefixed representation of expressions , e.g + `Arithmetic_Expression Arithmetic_Expression` which stands for the infix representation `Arithmetic_Expression + Arithmetic_Expression`

A.5.1 Arithmetic Expressions

Operator Symbol	*code*
+	0x20
-	0x21
	0x22
/	0x23
%	0x24
-	0x25
int Literal i	0x40i
char Literal i	0x41i

Codes for Arithmetic operations

binary operations :

$$\ulcorner \texttt{op Expression}_1\ \texttt{Expression}_2 \urcorner_{\texttt{context}} = \\ \mathit{code}(\texttt{op}) \\ \ulcorner \texttt{Expression}_1 \urcorner_{\texttt{context}} \\ \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{context}}$$

unary operations :

$$\ulcorner \texttt{op Expression} \urcorner_{\texttt{context}} = \\ \mathit{code}(\texttt{op}) \\ \ulcorner \texttt{Expression} \urcorner_{\texttt{context}}$$

A.5.2 JML expressions

JML constant	*code*
\ typeof	0x50
\ elemtype	0x51
\ result	0x52
\ old	
∗	0x53
\ type	0x54
\ Type	0x55

Codes of JML constant

$$\ulcorner \backslash\texttt{typeof(Expression)} \urcorner_{\texttt{context}} = \\ \texttt{code}(\backslash\texttt{typeof}) \\ \ulcorner \texttt{Expression} \urcorner_{\texttt{context}}$$

$$\ulcorner \backslash\texttt{elemtype(Expression)} \urcorner_{\texttt{context}} = \\ \texttt{code}(\backslash\texttt{elemtype}) \\ \ulcorner \texttt{Expression} \urcorner_{\texttt{context}}$$

$$\ulcorner \backslash\texttt{result} \urcorner_{\texttt{context}} = \texttt{code(} \backslash \texttt{ result)}$$

$$\ulcorner [\ \texttt{Expression} * \ \urcorner_{\texttt{context}} = \ulcorner [\urcorner_{\texttt{context}} \ulcorner \texttt{Expression} \urcorner_{\texttt{context}} \ulcorner * \urcorner_{\texttt{context}}$$

$$\ulcorner \backslash\texttt{old(Expression)} \urcorner_{\texttt{context}} = \texttt{code}(\backslash\texttt{old}) \ulcorner \texttt{Expression} \urcorner_{\texttt{context}}$$

$$\ulcorner \backslash\texttt{type(Expression)} \urcorner_{\texttt{context}} = \ulcorner \backslash\texttt{type} \urcorner_{\texttt{context}} \\ \ulcorner \mathit{Expression} \urcorner_{\texttt{context}}$$

$$\ulcorner \backslash\texttt{TYPE} \urcorner_{\texttt{context}} = \mathit{code}(\texttt{TYPE })$$

see A.5.5, etc. see **??**, A.5.7

A.5.3 Array access

symbol	*code*
`[`	0x61

Code for array access symbol

$\ulcorner$`[`$\urcorner_{\texttt{context}} = code($`[`$)$

$\ulcorner$`[ Expression Arithmetic_Expression`$\urcorner_{\texttt{context}} =$
$\ulcorner$`[`$\urcorner_{\texttt{context}}$
$\ulcorner$`Expression`$\urcorner_{\texttt{context}}$
$\ulcorner$`Arithmetic_Expression`$\urcorner_{\texttt{context}}$

A.5.4 Cast expression

symbol	*code*
`cast`	0x62

Codes for cast symbol

$\ulcorner$`cast`$\urcorner_{\texttt{context}} = code($`cast`$)$

$\ulcorner$`cast Expression Expression`$\urcorner_{\texttt{context}} =$
$\ulcorner$`cast`$\urcorner_{\texttt{context}}$
$\ulcorner$`Expression`$\urcorner_{\texttt{context}}$
$\ulcorner$`Expression`$\urcorner_{\texttt{context}}$

A.5.5 References

A.5.6 Variable Names

Variable names denote either local variables (parameters) , class or instance fields, either JML ghost fields.

kind of name	compile name
Field name	0x80 *index*(Field Name)
Local Variable	0x90 *index*(Local Variable)
JML ghost Field name)	0xA0 *index*(JML ghost Field name)

The function *index*is defined as follows :

Variable Identifier	*index*(Name)
Field name	the constant pool index at which a ConstantFieldReference attribute describes the field
JML field name	the constant pool index at which a ConstantFieldReference attribute describes the field
Local Variable	the index of the registers of the method that represents this variable(+ start_ind + length)

Two remarks :

1. the function `index` has the same definition for JML ghost fields and Java fields. Note that Java compiler adds constant fields data structures in the constant_pool only for fields that are dereferenced. For any field that is mentioned in the specification but not dereferenced in the Java code a new constant field refernence will be added on JML compilation time.

2. Note that Java compilers may generate code that uses the same register to store values of different types at different states of execution (and consequently at different points in the bytecode). In the present specification we consider that any register contains exactly one type of values at any point in the code and that it hold not more than one method parameter at any point in the bytecode.

A.5.7 Java keywords

Java keyword	*code*
`this`	`0x8000`
`null`	`0x06`

Codes for Java keywords

$\ulcorner$ keyword $\urcorner_{context} = code$(keyword)

Note: for the reserved Java keyword `this`, the JVMS always puts the reference to the `this` object at position 0 in the array of local variables for any non static method.

A.5.8 Fully qualified names

symbol	*code*
.	0x63

$\ulcorner . \urcorner_{\texttt{context}} = code(\ .)$

$$\ulcorner .\ \texttt{Expression}_1\texttt{Expression}_2 \urcorner_{\texttt{context}} =$$

$$\begin{cases}
\ulcorner . \urcorner_{\texttt{context}} \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{type(this)}}\ \ulcorner \texttt{this} \urcorner_{\texttt{context}} & if\ \texttt{Expression}_1 == \texttt{this} \\
\ulcorner \texttt{Expression}_2 \urcorner_{\texttt{type(Expression}_1\texttt{)}} & if\ \texttt{Expression}_1 == \texttt{super} \\
\ulcorner \texttt{Expression}_2 \urcorner_{\texttt{Expression}_1} & if\ \texttt{Expression}_1\ \texttt{is a class name} \\
\ulcorner . \urcorner_{\texttt{context}} \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{typelocal(s)}}\ \ulcorner \texttt{local(s)} \urcorner_{\texttt{context}} & \begin{array}{l} if\ \texttt{Expression}_1 \\ is\ a\ local\ variable \\ \wedge \\ index_in_local_array(\texttt{Expression}_1) \\ == s \end{array} \\
\ulcorner . \urcorner_{\texttt{context}} \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{ret_type(expr)}}\ \ulcorner \texttt{Expression}_1 \urcorner_{\texttt{context}} & \begin{array}{l} if\ \texttt{Expression}_1 = \\ \quad (\\ \quad \texttt{expr} \\ \quad length(list_expr) \\ \quad list_expr \end{array} \\
\ulcorner . \urcorner_{\texttt{context}} \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{elem_type(expr}_1\texttt{)}}\ \ulcorner \texttt{Expression}_1 \urcorner_{\texttt{context}} & \begin{array}{l} if\ \texttt{Expression}_1 = \\ \quad [\texttt{expr}_1\ \texttt{expr}_2 \end{array} \\
\ulcorner . \urcorner_{\texttt{context}} \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{type(Expression}_1\texttt{)}}\ \mathrm{code}(\mathrm{context}, \texttt{Expression}_1) & \begin{array}{l} if\ \texttt{Expression}_1\ \texttt{is} \\ a\ field\ name \end{array} \\
\ulcorner . \urcorner_{\texttt{context}} \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{type(Expression}_1\texttt{)}}\ \ulcorner \texttt{Expression}_1 \urcorner_{\texttt{context}} & else
\end{cases}$$

A.5.9 Specific keywords for the language

We introduce the keyword EXCEPTION that may appear only in exceptional postconditions. It stands for the thrown exception object

EXCEPTION	0xB5

A.5.10 Codes

$\ulcorner \mathrm{EXCEPTION} \urcorner_{\texttt{context}} = \mathrm{code}(\mathrm{EXCEPTION})$

A.6 Codes

Code	Symbol	Grammar
0x00	`True`	
0x01	`False`	
0x02	$\wedge$	Formula Formula
0x03	$\vee$	Formula Formula
0x04	$\Rightarrow$	Formula Formula
0x05	`!`	Formula
0x06	$\forall$	n (Type $)_n$ Formula
0x07	$\exists$	n (Type $)_n$ Formula
0x10	==	Expression Expression
0x11	>	Expression Expression
0x12	<	Expression Expression
0x13	<=	Expression Expression
0x14	>=	Expression Expression
0x15	`instanceof`	Expression Type
0x16	<:	Type Type
0x20	+	Expression Expression
0x21	−	Expression Expression
0x22	∗	Expression Expression
0x23	/	Expression Expression
0x24	%	Expression Expression
0x25	−	Expression
0x30	*and*	Expression Expression
0x31	*or*	Expression Expression
0x32	*xor*	Expression Expression
0x33	<<	Expression Expression
0x34	>>	Expression Expression
0x35	>>>	Expression Expression
0x40	`int constant`	i
0x41	`char constant`	i
0x50	\ `typeof`	Expression
0x51	\ `elemtype`	Type
0x52	\ `result`	
0x53	∗	Expression
0x54	\ `type`	Expression
0x55	\ `Type`	
0x56	\ `old`	
0x60	`(`	Expression n (Expression $)_n$
0x61	`[`	Expression Expression
0x62	`cast`	Type Expression
0x63	`.`	Expression Expression
0x64	`? :`	Formula Formula Formula
0x70	`this`	
0x72	`null`	
0x80	Fieldref	i
0x90	Local variable	i
0xA0	JML ghost field	i
0xB0	Methodref	i
0xC0	Type	i
0xE0	BoundVar	i
0xF0	Stack	
0xF1	Counter	

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] 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.

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

[5] 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.

[6] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1997. Also in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 258–267. Available by anonymous ftp from ftp.cs.iastate.edu.

[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.

ClassSpec ::=	$\mid$ *ClassInv* F	
	$\mid$ *ClassHistoryConstr* F	
	$\mid$ *declare ghost JavaType Name*	
InterMethodSpec ::=	$\mid$ *loopInvariant* F	
	$\mid$ *loopModifies list loc*	
	$\mid$ *loopDecreases*E	
	$\mid$ *assert* F	
	$\mid$ set E E	(sets to a new value a ghost variable)
MethodSpec ::=	$\mid$ SpecCase	
	$\mid$ SpecCase *also* MethodSpec	(specifies several specification cases)
SpecCase ::=	$\mid$ *requires* F	
	$\mid$ *modifies list loc*	(specifies the locations modified by a method)
	$\mid$ *ensures* F	
	$\mid$ *exsures* (*Exception exc*) F	(specifies what is the postcondition in case the method terminates with exception exc)
op ::=	$+ \mid - \mid * \mid$ *div* $\mid$ *rem* $\mid$ *bitwise*	
E ::=	$\mid$ Values	
	$\mid reg_i$	
	$\mid f$(E)	(field access)
	$\mid$ E[E]	(array access)
	$\mid$ *length*(E)	(returns the length of the array E)
	$\mid$ *null*	
	$\mid$ E *op* E	
	$\mid$ *cntr*	(stands for the counter of the operand stack of a method)
	$\mid$ *st*(E)	
	$\mid$ specExpr	
specExpr ::=	$\mid$ *typeof(*E*)*	(returns the dynamic type of E)
	$\mid$ *type(ClassName)*	
	$\mid$ *elemtype(*E*)*	(returns the type of the elements of the array E)
	$\mid$ *old(*E*)*	(used in method postcondition and stands for the value of E in the prestate of the method)
	$\mid$ *result*	(stands for the value returned by the method in case the method is not void)
R ::=	$\mid == \mid \neq \mid \leq \mid \leq \mid \geq \mid > \mid$ *subtype*	
F ::=	$\mid$ E_1 *rel* E_2, *rel* $\in$ R	
	$\mid$ *true*	
	$\mid$ *false*	
	$\mid$ *not* F	
	$\mid$ F $\wedge$ F	
	$\mid$ F $\vee$ F	
	$\mid$ F $\Rightarrow$ F	
	$\mid \forall\ x : Values.(\mathrm{F}(x))$	
	$\mid \exists\ x : Values.(\mathrm{F}(x))$	
Values ::=	$\mid i, i \in$ *int literal*	
	$\mid R, R \in \mathtt{REF}$	

Figure 2: grammar of BCSL

$$\backslash result = 1$$

$$\Longleftrightarrow$$

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

Figure 3: THE COMPILATION OF THE POSTCONDITION IN FIG. 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 4: STRUCTURE OF THE LOOP ATTRIBUTE