From JML to BCSL (Draft Version)

Lilian Burdy, Mariela Pavlova

July 18, 2005

Contents

1 Introduction **3**

2 General Conditions **3**
2.1 Restrictions on the Java compiler 3

3 Class annotation **3**
3.1 Model variables 3
3.2 Model(pure) methods 4
3.3 Class invariant 5
3.4 History Constraints 5

4 Method annotation **6**
4.1 Method specification 6
4.2 Set 7
4.3 Assert 8
4.4 Loop specification 8
4.5 Block specification 9

5 Formula and Expression compiler function **10**

6 Formulas **10**
6.1 Translation of formulas 10
6.2 Predicate constants 11
6.3 Logical connectors 11
6.4 Quantifiers 11
6.5 Bound Variables 11
6.6 Quantificator symbols 11
6.7 Predicate symbols 12

7 Expressions **12**
7.1 Arithmetic Expressions 13
7.2 JML expressions 13
7.3 Calls to Pure Methods 14
7.4 Array access 14
7.5 Cast expression 15
7.6 References 15
7.6.1 Variable Names 15

7.7 Java keywords 16
7.8 Fully qualified names 16
7.9 Specific keywords for the language 17
7.10 Codes 17

8 Example **17**

A Codes **20**

1 Introduction

This document defines a compiler for a subset of the Java Modeling Language(JML) [1] language. The compilation process needs the source file containing JML specification and the corresponding class file. The compilation process will compile the source specification in Java class attributes and will add them in the class file. These attributes are in compliance of the Java Virtual Machine specification (JVMS) [2] and their presence in a class file does not violate the Java virtual machine. The attributes are instantiation of a data structure predefined by the JVMS for "vendor specific use" [2] (4.7.1). The next section describes the requirements towards the Java compilers that may generate the class file and also the bytecode verifiers used on verification time.

The subset of JML that we support is enough to express a lot of properties. On class level we support the possibility of specifying properties concerning a class - class invariants, history constraints, declaration of JML fields - special variables not seen by the compiler. The set of JML specification concerning methods and supported by the compiler is : method preconditions and normal and exceptional postconditions, frame conditions and assertions (e.g. loop invariants) inside the bytecode.

2 General Conditions

2.1 Restrictions on the Java compiler

When compiling the specification of method m the compiler uses the `Local_Variable_table` and the `Line_Number_table` attribute of the `Code` attribute of the `Method_Info` for method m. Thus the class file may be generated by any standard Java compiler that generates also the tables `Local_Variable_table` and the `Line_Number_table` for every method appearing in the class file format. By standard compiler here is meant that the compiler is not performing any special optimizations except for taking away conditional branches that will be never taken.

3 Class annotation

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

3.1 Model variables

Model_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 "Model_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.

3.2 Model(pure) methods

```
Pure_Method_attribute {

    u2 attribute_name_index;
    u4 attribute_length;
    u2 methods_count;
    {  u2 access_flags;
       u2 name_index;
       u2 descriptor_index;
       u2 attributes_count;
       attribute_info attributes[attributes_count];
    } method_info_structure[methods_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 "Model_Method".

attribute_length
the length of the attribute in bytes.

method_info_structure
a structure where the name_index, descriptor_index are indexes in the constant pool.

3.3 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 (6) for formula grammar

3.4 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 $\mathtt{P} state, old(state)$ that establishes relation between the prestate and the postate of a method execution. see (6) for formula grammar

4 Method annotation

4.1 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.

4.2 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

4.3 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 (6)

4.4 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 (6)

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

4.5 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 (6)
modifies_count
The number of modified variable.

modifies[]
The array of modified formula.

postcondition
the predicate that is the postcondition of the block ,see (6)

5 Formula and Expression compiler function

The compiler function is denoted with $\ulcorner\urcorner_{\texttt{context}}$. The `context` is important for compiling field reference 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`.

6 Formulas

6.1 Translation of formulas

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

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 $\forall\ A\ a; P(a) \Rightarrow Q(a)$

6.2 Predicate constants

Predicate	*code*
True	0x00
False	0x01

Codes for the predicate constants True, False

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

6.3 Logical connectors

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

Codes for the Connector symbols

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

6.4 Quantifiers

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

6.5 Bound Variables

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

6.6 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.

6.7 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`)

7 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`

```
Expression ::= Arithmetic_Expression |
               Identifier |
               [ Expression Arithmetic_Expression |
               ( Expression Expression* |
               .  Expression Expression |
               cast Expression Expression |
               null |
               super |
               this |
               JML_Expression
               VM_Expression
```

```
VM_Expression ::= Stack(Counter_Stack)
```

```
Counter_Stack ::= Counter |
                  Counter + int constant |
                  Counter - int constant
```

```
JML_Expression ::= typeof(Expression) |
                   elemtype(Expression) |
                   \old(Expression) |
                   \ result |
                   [ Expression * |
```

```
                    [ Expression Expression Expression |
                    \type(Expression) |
                    \TYPE

Arithmetic_Expression ::= + Expression Expression |
                          − Expression Expression |
                          ∗ Expression Expression |
                          / Expression Expression |
                          % Expression Expression |
                          − Expression |
                          int constant |
```

7.1 Arithmetic Expressions

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

Codes for Arithmetic operations

binary operations :

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

$\quad code(\texttt{op})$

$\quad \ulcorner \texttt{Expression}_1 \urcorner_{\texttt{context}}$

$\quad \ulcorner \texttt{Expression}_2 \urcorner_{\texttt{context}}$

unary operations :

$\ulcorner \texttt{op Expression} \urcorner_{\texttt{context}} =$

$\quad code(\texttt{op})$

$\quad \ulcorner \texttt{Expression} \urcorner_{\texttt{context}}$

7.2 JML expressions

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

Codes of JML constant

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

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

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

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

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

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

$\ulcorner$`\TYPE`$\urcorner_{\texttt{context}}$ = *code*(`TYPE`)
see 7.6, etc. see **??**, 7.7

7.3 Calls to Pure Methods

symbol	*code*
`(`	0x60

Code for method call symbol

$\ulcorner$`(`$\urcorner_{\texttt{context}}$ = *code*(`(`)
$\ulcorner$`( Expression Expression`$_1$......`Expression`$_n$ $\urcorner_{\texttt{context}}$ =
$\ulcorner$`(`$\urcorner_{\texttt{context}}$
$\ulcorner$`Expression`$\urcorner_{\texttt{context}}$
n
$\ulcorner$`Expression`$_1$ $\urcorner_{\texttt{type(this)}}$
...
$\ulcorner$`Expression`$_n$ $\urcorner_{\texttt{type(this)}}$

7.4 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`$\urcorner_{\texttt{context}}$

7.5 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`$\urcorner_{\texttt{context}}$

7.6 References

7.6.1 Variable Names

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

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

7.7 Java keywords

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

Codes for Java keywords

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

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.

7.8 Fully qualified names

symbol	*code*
.	0x63

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

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

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

7.9 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

7.10 Codes

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

8 Example

$\ulcorner$ nodes.equals(\ `old`(nodes.insert(n))) $\urcorner_{\texttt{this}}$

$=$

$$\ulcorner\ .\ \urcorner_{\texttt{this}}$$
$$\quad\quad \ulcorner\ \text{equals}(\backslash\texttt{old}\ (\text{nodes.insert(n)})\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad \ulcorner\text{nodes}\ \urcorner_{\texttt{this}}$$

$=$

$$\ulcorner\ .\ \urcorner_{\texttt{this}}$$
$$\quad\quad \ulcorner\texttt{(}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad \ulcorner\ \text{equals}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad 1$$
$$\quad\quad\quad\quad \ulcorner\ \backslash\ \texttt{old}(\text{nodes.insert(n)})\ \urcorner_{\texttt{type(this)}}$$
$$\quad\quad \ulcorner\ \text{nodes}\urcorner_{\texttt{this}}$$

$=$

$$\ulcorner\ .\urcorner_{\texttt{this}}$$
$$\quad\quad \ulcorner\texttt{(}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad \ulcorner\ \text{equals}\urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad 1$$
$$\quad\quad\quad\quad \ulcorner\ \text{nodes.insert(n)}\ \urcorner^{old}_{\texttt{this}}$$
$$\quad\quad \ulcorner\ \text{nodes}\urcorner_{\texttt{this}}$$

$=$

$$\ulcorner\ .\ \urcorner_{\texttt{this}}$$
$$\quad\quad \ulcorner\texttt{(}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad \ulcorner\ \text{equals}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad 1$$
$$\quad\quad\quad\quad \ulcorner\ .\ \urcorner^{old}_{\texttt{this}}$$
$$\quad\quad\quad\quad\quad\quad \ulcorner\ \text{insert(n)}\ \urcorner^{old}_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad\quad\quad \ulcorner\ \text{nodes}\ \urcorner^{old}_{\texttt{this}}$$
$$\quad\quad \ulcorner\ \text{nodes}\urcorner_{\texttt{this}}$$

$=$

$$\ulcorner\ .\urcorner_{\texttt{this}}$$
$$\quad\quad \ulcorner\texttt{(}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad \ulcorner\ \text{equals}\ \urcorner_{\texttt{type(nodes(this))}}$$
$$\quad\quad\quad\quad 1$$
$$\quad\quad\quad\quad \ulcorner\ .\ \urcorner^{old}_{\texttt{this}}$$
$$\quad\quad\quad\quad\quad\quad \ulcorner\ \texttt{(}\ \urcorner^{old}_{\texttt{type(nodes(this))}}$$

$$\ulcorner \text{insert} \urcorner^{old}_{\texttt{type(nodes(this))}}$$

1

$$\ulcorner \text{n} \urcorner^{old}_{\texttt{type(this)}}$$

$$\ulcorner \text{nodes} \urcorner^{old}_{\texttt{type(this)}}$$

$$\ulcorner \text{nodes} \urcorner_{\texttt{this}}$$

=

code(.)

code(( )

code(type(nodes(this)),equals)

1

code(.)

code(()

old(type(nodes(this)), insert)

1

old(type(this), n)

old(type(this) , nodes)

code(type(this), nodes)

A 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 literal`	i
0x41	`char literal`	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 model field	i
0xB0	Methodref	i
0xC0	Type	i
0xE0	BoundVar	i
0xF0	Stack	
0xF1	Counter	

References

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

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