Bytecode Verification and its Applications

July 11, 2006

Contents

1 Introduction 5

2 Java bytecode language and its operational semantics 7

2.1 Related Work 9
2.2 Notation 9
2.3 Classes, Fields and Methods 10
2.4 Program types and values 11
2.5 State configuration 12
2.5.1 Modeling the Object Heap 14
2.5.2 Registers 17
2.5.3 The operand stack 17
2.5.4 Program counter 17
2.6 Throwing and handling exceptions 18
2.7 Bytecode Language and its Operational Semantics 19

3 Specification language for Java bytecode programs 29

3.1 Introduction 29
3.2 A quick overview of JML 30
3.3 BML 31
3.3.1 Notation convention 32
3.3.2 BML Grammar 32
3.3.3 Informal semantics of BML 34
3.4 Well formed BML specification 41
3.5 Compiling JML into BML 42

4 Verification condition generator for Java bytecode 47

4.1 Related work 48
4.2 The expression language 49
4.2.1 Substitution 51
4.2.2 Interpretation 52
4.3 Representing bytecode programs as control flow graphs 54
4.4 Extending method declarations with specification 56
4.5 Weakest precondition calculus 59
4.5.1 Intermediate predicates 59

4.5.2 Weakest precondition in the presence of exceptions . . . 60
4.5.3 Rules for single instruction 61
4.6 Example 69

5 Correctness of the verification condition generator 71
5.1 Substitution properties 72
5.2 Proof of Correctness 74

6 Equivalence between Java source and bytecode proof Obligations 83

7 A compact verification condition generator 85

8 Applications 87

9 Conclusion 89

Appendices 89

Chapter 1

Introduction

Chapter 2

Java bytecode language and its operational semantics

The purpose of this section is to introduce the fundamental concepts of the present thesis. In particular, we present a bytecode language and its operational semantics. Those concepts will be used later in Chapter 4 for the definition of the verification procedure as well as for establishing its correctnes w.r.t. the operational semantics given in this section. As our verification procedure is tailored to Java bytecode the bytecode language introduced hereafter is close to the Java Virtual Machine language [20](JVM for short). However, it abstracts from some of the JVM language features while supporting others. Thus, we can concentrate on the part of the JVM which we consider the most typical. We now look closer at what are the characteristic of our bytecode language.

The features supported by our bytecode language are

- arithmetic operations like multiplication, division, addition and substruction
- stack manipulation. Similarly to the JVM our abstract machine is stack based, i.e. instructions get their arguments from the operand stack and push their result on the operand stack
- method invokation. Our bytecode language is modular and thus, methods are the basic execution units. In our formalization methods always return a value
- object manipulation and creation. We support field access and update as well as object creation
- exception throwing and handling. Our bytecode language supports exceptions which are thrown if the program execution does not respect the language semantics like for example, dereferencing a null object reference

- classes and class inheritance. Like in the JVM language, our bytecode language supports a tree class hierarchy in which every class has a super class except the class `Object`

- basic types. The unique basic type that we support is the integer type. This is not so unrealistic as the JVM supports only few instructions for dealing with the other integral types, like byte, short and long. On the other hand, supporting floating point numbers is not in the scope of the current thesis

Our bytecode language omits some of the features of Java, in order to concentrate on the features listed above.

The features not supported by our bytecode language are

- void methods, still this is not a major restriction for our bytecode language as it can be extended easily to support this feature

- static fields and methods. This kind of data is shared between all the instances of the class where the data is declared. This restriction can be overcome easily by

- static initialization.

- subroutines. The basic reason that our bytecode language does not support subroutines is that in the implementation of our bytecode verification condition generator we inline them and thus, there is no need of supporting them in the language.

- interface types

- floating point arithmetic

In what follows, we give a big step operational semantics of the bytecode language whose major difference with most of the formalizations of the JVM is that it abstracts from the method frame stack. JVM is stack based and when a new method is called a new method frame is pushed on the frame stack and the execution continues on this new frame. A method frame contains the method operand stack, the array of registers and the constant pool of the class the method belongs to. When a method terminates its execution normally, the result, if any, is popped from the method operand stack, the method frame is popped from the frame stack and the method result (if any) is pushed on the operand stack of its caller. If the method terminates with an exception, it does not return any result and the exception object is propagated back to its callers. This is different from most of the existing formalization of the JVM (or JVM like languages), and is due to its big step nature. However, this semantics is sufficient for our purposes which are to prove the correctness of our verification calculus.

The rest of this chapter is organized as follows: subsection 2.1 is an overview of existing formalisations of the JVM semantics, subsection 2.2 gives some particular notations that will be used from now on along the thesis, subsection 2.3 introduces the structures classes, fields and methods used in the virtual machine, subsection 2.4 gives the type system which is supported by the bytecode language, subsection 2.5 introduces the notion of state configuration, subsection 2.5.1 gives the modelisation of the memory heap, subsection 2.7 gives the operational semantics of our language.

2.1 Related Work

A considerable effort has been done on the formalization of the semantics of the JVM. Most of the existing formalizations cover a representative subset of the language. Among them is the work [13] by N.Freund and J.Mitchell and [23] by Qian, which give a formalization in terms of a small step operational semantics of a large subset of the Java bytecode language including method calls, object creation and manipulation, exception throwing and handling as well subroutines, which is used for the formal specification of the language and the bytecode verifier.

Based on the work of Qian, in [22] C.Pusch gives a formalization of the JVM and the Java Bytecode Verifier in Isabelle/HOL and proves in it the soundness of the specification of the verifier. In [17], Klein and Nipkow give a formal small step and big step operational semantics of a Java-like language called Jinja, an operational semantics of a Jinja VM and its type system and a bytecode verifier as well as a compiler from Jinja to the language of the JVM. They prove the equivalence between the small and big step semantics of Jinja, the type safety for the Jinja VM, the correctness of the bytecode verifier w.r.t. the type system and finally that the compiler preseves semantics and well-typedness.

The small size and complexity of the JavaCard platform simplifies the formalization of the system and thus, has attracted particularly the scientific interest. CertiCartes [5, 4] is an in-depth formalization of JavaCard. It has a formal executable specification written in Coq of a defensive and an offensive JCVM and an abstract JCVM together with the specification of the Java Bytecode Verifier. Siveroni proposes a formalization of the JCVM in [27] in terms of a small step operational semantics.

2.2 Notation

Here we give the semantics of several notations used in the rest of this chapter. If we have a function f with domain type A and range type B we note it with $f : A \rightarrow B$. If the function receives n arguments of type $A_1 \ \dots \ A_n$ respectively and maps them to elements of type B we note the function signature with $f : A_1 * ... * A_n \rightarrow B$. Function updates of function f with n arguments is denoted with $f[\oplus x_1 \dots x_n \rightarrow y]$ and the definition of such function is :

$$f[\oplus x_1 \dots x_n \rightarrow y](z_1 \dots z_n) = \begin{cases} y & if\ x_1 = z_1 \wedge ... \wedge x_n = z_n \\ f(z_1 \dots z_n) & else \end{cases}$$

The function *inList* takes as arguments any list and an object and returns *true* if the object is in the list and *false* otherwise:

$$\mathit{inList} : \mathit{list}\ A * A \rightarrow\ \mathit{bool}$$

The empty list is denoted with [ ]. For any type A, the function cons takes as argument any list $l : \mathit{list}\ A$ and an object $o : A$ and returns a list $l1$ such that $l1.\mathit{head} = o \wedge l1.\mathit{tail} = l$:

$$\mathrm{cons} : \mathit{list}\ A * A \rightarrow \mathit{list}\ A$$

The function *inDom* (f, e) determines if the element e is in the domain of the function f. The function *inRan* (f, e) determines if the element e is in the range of the function f

2.3 Classes, Fields and Methods

Java programs are a set of classes. As the JVM says *A class declaration specifies a new reference type and provides its implementation. . . . The body of a class declares members (fields and methods), static initializers, and constructors.* In our formalisation, the set of classes is denoted with **Class**, the set of fields with **Field**, the set of methods **Method**. We define a domain for class names **ClassName**, for field names **FieldName** and for method names **MethodName** respectively.

An object of type **Class** is a tuple with the following components: list of field objects (fields), which are declared in this class, list of the methods declared in the class (methods), the name of the class (className) and the super class of the class (superClass). All classes, except the special class `Object` , have a unique direct super class. Formally, a class of our bytecode language has the following structure:

$$\mathbf{Class} = \left\{ \begin{array}{ll} \mathsf{fields} & : \mathit{list}\ \mathbf{Field} \\ \mathsf{methods} & : \mathit{list}\ \mathbf{Method} \\ \mathsf{className} & : \mathbf{ClassName} \\ \mathsf{superClass} & : \mathbf{Class} \cup \{\bot\} \end{array} \right\}$$

A field object is a tuple that contains the unique field id and a field type and the class where it is declared :

$$\mathbf{Field} = \left\{ \begin{array}{ll} \mathsf{Name} & : \mathbf{FieldName}; \\ \mathsf{Type} & : \mathit{JType}; \\ \mathsf{declaredIn} & : \mathbf{Class} \cup \{\bot\} \end{array} \right\}$$

We introduce a special field which stands for the number of components of any reference pointing to an array object and which does not belong to any class (the name of the object and its field Name have the same name):

$$\text{arrLength} = \left\{ \begin{array}{ll} \mathsf{Name} & = \text{arrLength}; \\ \mathsf{Type} & = \mathit{int}; \\ \mathsf{declaredIn} & = \bot \end{array} \right\}$$

A method has a unique method id (Name), a return type (retType), a list containing the formal parameter names and their types(args), the number of its formal parameters (nArgs), list of bytecode instructions representing its body (body), the exception handler table (excHndlS) and the list of exceptions (exceptions) that the method may throw

$$\mathbf{Method} = \left\{ \begin{array}{ll} \mathsf{Name} & : \mathbf{MethodName} \\ \mathsf{retType} & : \mathit{JType} \\ \mathsf{args} & : (\mathit{name} * \mathit{JType})[] \\ \mathsf{nArgs} & : \mathit{nat} \\ \mathsf{body} & : \ \mathrm{I}[] \\ \mathsf{excHndlS} & : \mathbf{ExcHandler}[] \\ \mathsf{exceptions} & : \mathbf{Class}_{exc}[] \end{array} \right\}$$

We assume that for every method `m`the entrypoint is the first instruction in the array of instructions of which its body consists, i.e. $\mathtt{m}.\mathsf{entryPnt} = \mathtt{m}.\mathsf{body}[0]$.

An object of type **ExcHandler** contains information about the region in the method body that it protects, i.e. the start position (startPc) of the region and the end position (endPc), about the exception it protects from (exc), as well as what position in the method body the exception handler starts (handlerPc) at.

$$\mathbf{ExcHandler} = \left\{ \begin{array}{ll} \mathsf{startPc} & : \mathit{nat} \\ \mathsf{endPc} & : \mathit{nat} \\ \mathsf{handlerPc} & : \mathit{nat} \\ \mathsf{exc} & : \mathbf{Class}_{exc} \end{array} \right\}$$

We impose the following constraints about startPc, endPc and handlerPc:

$$\begin{array}{l} \forall \mathtt{m} : \mathbf{Method}, \\ \forall i : \mathit{nat}, 0 \leq i < \mathtt{m}.\mathsf{excHndlS}.\mathit{length}, \\ \quad 0 \leq \mathtt{m}.\mathsf{excHndlS}[i].\mathsf{endPc} < \mathtt{m}.\mathsf{body}.\mathit{length} \wedge \\ \quad 0 \leq \mathtt{m}.\mathsf{excHndlS}[i].\mathsf{startPc} < \mathtt{m}.\mathsf{body}.\mathit{length} \wedge \\ \quad 0 \leq \mathtt{m}.\mathsf{excHndlS}[i].\mathsf{handlerPc} < \mathtt{m}.\mathsf{body}.\mathit{length} \end{array}$$

2.4 Program types and values

The types supported by our language are a simplified version of the types supported by the JVM. Thus, we have a unique simple type : the integer data type **int**. The reference type (*RefType*) stands for the simple reference types

(*RefClType*) and array reference types (*RefArrType*). As we said in the beginning of this chapter, the language does not support interface types.

$$
\begin{array}{ll}
\mathit{JType} & ::= \mathbf{int} \mid \mathit{RefType} \\
\mathit{RefType} & ::= \mathit{RefClType} \ \mid \mathit{RefArrType} \\
\mathit{RefClType} & ::= \mathbf{Class} \\
\mathit{RefArrType} & ::= \mathit{JType}[]
\end{array}
$$

Our language supports two kinds of values : values of the basic type **int** and reference values *RefVal*. *RefVal*may be either references to class objects or references to array objects. The set of references of class objects is denoted with ref and the set of references to array objects is represented with $refArr$. The following definition gives the formal grammar of values:

$$
\begin{array}{ll}
\mathit{Values} & ::= i, i \in \ \mathbf{int}\ \text{literal} \ \mid \mathit{RefVal} \\
\mathit{RefVal} & ::= ref \mid \mathit{RefValArr} \mid \mathbf{null} \\
\mathit{RefValArr} & ::= refArr
\end{array}
$$

Every type has an associated default value which can be accessed via the function *defVal*. The function is defined as follows:

$$\mathsf{defVal} : \mathit{RefType} \rightarrow \mathit{Values}$$

$$\mathsf{defVal}(\mathtt{T}) = \begin{cases} \mathbf{null} & \mathtt{T} \in \mathit{RefType} \\ 0 & \mathtt{T} = \mathbf{int} \end{cases}$$

We define also a subtyping relation as follows:

$$\frac{}{\mathsf{subtype}\ (C,C)} \qquad \frac{C2{=}C1.\mathsf{superClass}}{\mathsf{subtype}\ (C1,C2)}$$

$$\frac{C3{=}C1.\mathsf{superClass}\ \mathsf{subtype}\ (C3,C2)}{\mathsf{subtype}\ (C1,C2)} \qquad \frac{}{\mathsf{subtype}\ (C1,\mathtt{Object})}$$

$$\frac{}{\mathsf{subtype}\ (C[],\mathtt{Object})} \qquad \frac{\mathsf{subtype}\ (C1,C2)}{\mathsf{subtype}\ (C1[],C2[])}$$

2.5 State configuration

In this section, we introduce the notion of state configuration. A state configuration Kmodels the program state in particular execution program point by specifying what is the state of the memory heap, the stack and the stack counter, the values of the local variables of the currently executed method and what is the instruction which is executed next. Note that, as we stated before our semantics ignores the method call stack and so, state configurations also omit the call frames stack.

We define two kinds of state configurations:

$$K = K^{interm} \cup K^{final}$$

The set K^{interm} consists of method intermediate state configurations, which stand for an *intermediate state* in which the execution of the current method is not finished i.e. there is still another instruction of the method body to be executed. The configuration $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \in K^{interm}$ has the following elements:

- the function H: HeapType which stands for the heap in the state configuration
- Cntr is a variable that contains a natural number which stands for the number of elements in the operand stack.
- Stis a partial function from natural numbers to values which stands for the operand stack.
- Reg is a partial function from natural numbers to values which stands for the array of local variables of a method. Thus, for an index `i` it returns the value $\mathbf{reg}(i)$ which is stored at that index of the array of local variables
- Pc stands for the program counter and contains the index of the instruction to be executed in the current state

The elements of the set K^{final} are the final states, states in which the current method execution is terminated and consists of normal termination states (K^{norm}) and exceptional termination states (K^{exc}):

$$K^{final} = K^{norm} \cup K^{exc}$$

A method may terminate either normally (by reaching a return instruction) or exceptionally (by throwing an exception).

- $< \mathrm{H}, \mathrm{Reg}, \mathrm{Res} >^{norm} \in K^{norm}$ which describes a *normal final state*, i.e. the method execution terminates normally. The normal termination configuration has the following components :
  - the function H: HeapType which reflects what is the heap state after the method terminated
  - Reg is the array of local variables of a method
  - Res stands for the return value of the method
- $< \mathrm{H}, \mathrm{Reg}, \mathrm{Exc} >^{exc} \in K^{exc}$ which stands for an *exceptional final state* of a method, i.e. the method terminates by throwing an exception. The exceptional configuration has the following components:
  - the heap H
  - Reg is the array of local variables of a method
  - Exc is a reference to the uncaught exception that caused the method termination

When an element of a state configuration $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ is updated we use the notation:

$$K[E \leftarrow V],\ E \in \{\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}\}$$

We will denote with $< \mathrm{H}, \mathrm{Reg}, \mathit{Final} >^{final}$ for any configuration which belongs to the set K^{final}. Later on in this chapter, we define in terms of state configuration transition relation the operational semantics of our bytecode programming language. In the following, we focus in more detail on the heap modelization and the operand stack.

2.5.1 Modeling the Object Heap

An important issue for the modelization of an object oriented programming language and its operational semantics is the garbage collected memory heap. As the JVM specification states, the heap is the runtime data area from which memory for all class instances and arrays is allocated. Whenever a new instance is allocated, the JVM returns a reference value that points to the newly created object. We introduce a record type HeapType which models the memory heap. We do not take into account garbage collection and thus, we assume that heap objects has an infinite space memory.

In our modelization, a heap consists of the following components:

- a component named Fld which is a partial function that maps field structures (of type **Field** introduced in subsection 2.3) into partial functions from references (*RefType*) into values (*Values*).
- a component Arr which maps the components of arrays into their values
- a component Loc which stands for the list of references that the heap has allocated
- a component TypeOf is a partial function which maps references to their dynamic type

Formally, the data type HeapType has the following structure:

$$\forall \mathrm{H} : \mathsf{HeapType},$$
$$\mathrm{H} = \left\{ \begin{array}{ll} \mathsf{Fld} & : \mathbf{Field} \rightharpoonup (\mathit{RefVal} \rightharpoonup \mathit{Values}) \\ \mathsf{Arr} & : \mathit{RefValArr} * \mathit{nat} \rightharpoonup \mathit{Values} \\ \mathsf{Loc} & : \mathit{list}\ \mathit{RefVal} \\ \mathsf{TypeOf} & : \mathit{RefVal} \rightharpoonup \mathit{RefType} \end{array} \right\}$$

Another possibility is to model the heap as partial function from locations to objects where objects contain a function from fields to values. Both formalizations are equivalent, still we have chosen this model as it follows closely our implementation of the verification condition generator.

In the following, we are interested only in heap objects H which guarantee that the value of the components H.Fld and H.Arr are functions which are defined only for references from the proper type and which are in the list of references of the heap H.Loc:

as well as that if a field function returns a reference value then this reference value is once again in H.Loc:

$$\begin{array}{ll}
\forall f : \mathbf{Field}, \forall \mathtt{ref} \in \mathit{RefVal}, & \mathit{inDom}\ (\mathrm{H}.\mathsf{Fld}(f), \mathtt{ref}) \Rightarrow \\
& \mathit{inList}(\mathrm{H}.\mathsf{Loc}, \mathtt{ref})\ \wedge \\
& \mathsf{subtype}\ (\mathrm{H}.\backslash\mathbf{typeof(}\mathtt{ref}\mathbf{)}, f.\mathsf{declaredIn}) \\
\wedge & \\
\forall \mathtt{ref} \in \mathit{RefValArr}, & \mathit{inDom}\ (\mathrm{H}.\mathsf{Arr}, (\mathtt{ref}, i)) \Rightarrow \\
& \mathit{inList}(\mathrm{H}.\mathsf{Loc}, \mathtt{ref})\ \wedge \\
& 0 \leq i < \mathrm{H}.\mathsf{Fld}(\mathrm{arrLength})(\mathtt{ref})
\end{array}$$

Also, we assume that the heap must contain well formed values. By this, we mean that the heap maps any field object f : **Field** which has a reference type (i.e. the component f .Type contains a reference type) into a function which may only return references which are already defined in the heap :

$$\begin{array}{ll}
\forall f : \mathbf{Field}, & \forall \mathtt{ref} \in \mathit{RefVal}, \\
& f.\mathsf{Type} \in \mathit{RefType} \wedge \\
& \mathit{inRan}\ (\mathrm{H}.\mathsf{Fld}(f), \mathtt{ref}) \Rightarrow \\
& \quad \mathit{inList}(\mathrm{H}.\mathsf{Loc}, \mathtt{ref})\ \vee \mathtt{ref} = \mathbf{null}
\end{array}$$

We define an operation allocator which add a new reference to the list of references in a heap. The only change that the operation will cause to the heap H is to add a new reference `ref` to the list of references of the heap H.Loc:

$$\mathrm{allocator}\ : \mathsf{HeapType} * \mathit{RefType} \rightarrow \mathsf{HeapType}$$

Formally, the operation is defined as follows:

$$\begin{array}{l}
\mathrm{allocator}(\mathrm{H}, \mathtt{ref})\ = \mathrm{H}' \iff^{def} \\
\\
\mathrm{H}.\mathsf{Loc} = l \wedge \\
\mathit{inList}(l, \mathtt{ref})\ = \mathit{false} \wedge \\
\mathrm{H}'.\mathsf{Loc} = \mathrm{cons}(\mathtt{ref}, l) \wedge \\
\mathrm{H}.\mathsf{Fld} = \mathrm{H}'.\mathsf{Fld} \wedge \\
\mathrm{H}.\mathsf{Arr} = \mathrm{H}'.\mathsf{Arr} \wedge
\end{array}$$

In the above definition, we use the function *instFlds*, which for a given field f and C returns true if f is an instance field of C:

$$\mathit{instFlds} : \mathbf{Field} \rightarrow \mathbf{Class} \rightarrow \mathit{bool}$$

$$\mathit{instFlds}(f, C) = \\
\begin{cases}
\mathit{true} & f.\mathsf{declaredIn} = C \\
\mathit{false} & C = \mathtt{Object} \wedge f.\mathsf{declaredIn} \neq \mathtt{Object} \\
\mathit{instFlds}(f, C.\mathsf{superClass}) & \mathit{else}
\end{cases}$$

If a new object of class C is created in the memory, a fresh reference `ref` which points to the newly created object is added in the heap H and all the values of the field functions that correspond to the fields in class C are updated for the new reference with the default values for their corresponding types. The function which for a heap Hand a class type Creturns the same heap but with a fresh reference of type C has the following name and signature:

$$\text{newRef} : \text{H} \rightarrow \textit{RefClType} \ \rightarrow \text{H} * ref$$

The formalization of the resulting heap and the new reference is the following:

$$\begin{array}{l}
\text{newRef}(\text{H}, C) = (\text{H}', \texttt{ref}) \iff^{def} \\
\\
\text{allocator}(\text{H}, \texttt{ref}) \ = \text{H}' \wedge \\
\texttt{ref} \neq \textbf{null} \wedge \\
\text{H}'.\mathsf{TypeOf} \ := \text{H}.\mathsf{TypeOf} \ [\oplus \texttt{ref} \rightarrow C] \wedge \\
\ \forall f : \textbf{Field}, \quad \textit{instFlds}(f, C) \Rightarrow \\
\qquad\qquad\qquad \text{H}'.\mathsf{Fld} := \text{H}'.\mathsf{Fld}[\oplus f \rightarrow f[\oplus \texttt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]] \wedge
\end{array}$$

Identically, when allocating a new object of array type whose elements are of type `T` and length l, we obtain a new heap object newArrRef(H, `T`[ ], l) which is defined similarly to the previous case:

$$\text{newArrRef} : \text{H} \rightarrow \textit{RefArrType} \rightarrow \text{H} * refArr$$

$$\begin{array}{l}
\text{newArrRef}(\text{H}, \texttt{T}[\ ], l) \ = (\text{H}', \texttt{ref}) \iff^{def} \\
\\
\text{allocator}(\text{H}, \texttt{ref}) \ = \text{H}' \wedge \\
\texttt{ref} \neq \textbf{null} \wedge \\
\text{H}'.\mathsf{TypeOf} \ := \text{H}.\mathsf{TypeOf} \ [\oplus \texttt{ref} \rightarrow \texttt{T}[\ ]] \wedge \\
\text{H}'.\mathsf{Fld} := \text{H}'.\mathsf{Fld}[\oplus \text{arrLength} \rightarrow \text{arrLength}[\oplus \texttt{ref} \rightarrow l]] \wedge \\
\forall i, 0 \leq i < l \Rightarrow \text{H}'.\mathsf{Arr} := \text{H}'.\mathsf{Arr}[\oplus(\texttt{ref}, i) \rightarrow \mathsf{defVal}(\texttt{T})]
\end{array}$$

In the following, we adopt few more naming conventions which do not create any ambiguity. Getting the function corresponding to a field f in a heap H : H.$\mathsf{Fld}(f)$ is replaced with H(f) for the sake of simplicity.

The same abreviation is done for access of an element in an array object referenced by the reference `ref`at index i in the heap H. Thus, the usual denotation: H.$\mathsf{Arr}(\texttt{ref}, i)$ becomes H($\texttt{ref}, i$).

Whenever the field f for the object pointed by reference `ref` is updated with the value *val*, the component H.Fld is updated:

$$\text{H}.\mathsf{Fld} := \text{H}.\mathsf{Fld}[\oplus f \rightarrow \text{H}.\mathsf{Fld}(f)[\oplus \texttt{ref} \rightarrow val]]$$

In the following for the sake of clarity, we will use another lighter notation for a field update which do not imply any ambiguities:

$$\text{H}[\oplus f \rightarrow f[\oplus \texttt{ref} \rightarrow val]]$$

If in the heap H the i^{th} component in the array referenced by `ref` is updated with the new value *val*, this results in assigning a new value of the component H.Arr:

$$\mathsf{H.Arr} := \mathrm{H}.\mathsf{Arr}[\oplus(\mathtt{ref}, i) \rightarrow val]$$

In the following, for the sake of clarity, we will use another lighter notation for an update of an array component which do not imply any ambiguities:

$$\mathrm{H}[\oplus(\mathtt{ref}, i) \rightarrow val]$$

2.5.2 Registers

State configurations have an array of registers which is denoted with Reg. Registers are addressed by indexing and the index of the first local variable is zero. Thus, Reg(0) stands for the first register in the state configuration. An integer is be considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array. Registers are used to pass parameters on method invocation. On class method invocation any parameters are passed in consecutive local variables starting from register Reg(0). Reg(0) is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1.

2.5.3 The operand stack

Like the JVM language, our bytecode language is stack based. This means that every method is supplied with a Last In First Out stack which is used for the method execution to store intermediate results. The method stack is modeled by the partial function St and the variable Cntr keeps track of the number of the elements in the operand stack. St is defined for any integer `ind` smaller than the operand stack counter Cntr and returns the value St(`ind`) stored in the operand stack at `ind` positions of the bottom of the stack. When a method starts execution its operand stack is empty and we denote the empty stack with [ ]. Like in the JVM our language supports instructions to load values stored in registers or object fields and viceversa. There are also instructions that take their arguments from the operand stack St, operate on them and push the result on the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

2.5.4 Program counter

The last component of an intermediate state configuration is the program counter Pc. It contains the number of the instruction in the array of instructions of the current method which must be executed in the state.

2.6 Throwing and handling exceptions

As the JVM specification states *exception are thrown if a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception. An example of such a violation is an attempt to index outside the bounds of an array. The Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a nonlocal transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred. A method invocation that completes because an exception causes transfer of control to a point outside the method is said to complete abruptly. Programs can also throw exceptions explicitly, using throw statements ...*

Our language supports an exception handling mechanism similar to the JVM one. More particularly, it supports Runtime exceptions:

- `NullPntrExc` thrown if a null pointer is dereferenced
- `NegArrSizeExc` thrown if an array is accessed out of its bounds
- `ArrIndBndExc` thrown if an array is accessed out of its bounds
- `ArithExc` thrown if a division by zero is done
- `CastExc` thrown if an object reference is cast to to an incompatible type
- `ArrStoreExc` thrown if an object is tried to be stored in an array and the object is of incompatible type with type of the array elements

The language also supports programming exceptions. Those exceptions are forced by the programmer, by a special intruction called athrow . The modelization of the exception handling mechanism involves several functions. The function *getStateOnExc* deals with bytecode instructions that may throw exceptions. The function returns the state configuration after the current instruction during the execution of `m` throws an exception of type `E`. If the method `m` has an exception handler which can handle exceptions of type `E` thrown at the index of the current instruction, the execution is not stuck and thus, the state configuration is an intermediate state configuration. If the method `m` does not have an exception handler for dealing with exceptions of type `E` at the current index, the execution of `m` terminates exceptionally and the current instruction causes the method exceptional termination:

$$\mathit{getStateOnExc} \ : K^{interm} * \mathit{ExcType} * \textbf{ExcHandler}[] \to K^{interm} \cup K^{exc}$$

$$\mathit{getStateOnExc}\ (<\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}>, \mathtt{E}, \mathrm{Pc}, \mathsf{excH}[]) =$$
$$\begin{cases} <\mathrm{H}', 0, \mathrm{St}[\oplus 0 \to \mathtt{ref}], \mathrm{Reg}, \mathsf{handlerPc}> & \begin{array}{l}\text{if } \mathit{findExcHandler}(\mathtt{E}, \mathrm{Pc}, \mathsf{excH}[]) \\ = \mathsf{handlerPc}\end{array} \\ \\ <\mathrm{H}', \mathrm{Reg}, \mathtt{ref}>^{exc} & \begin{array}{l}\text{if } \mathit{findExcHandler}(\mathtt{E}, \mathrm{Pc}, \mathsf{excH}[]) \\ = \perp\end{array} \end{cases}$$

where
$(\mathrm{H}', \mathtt{ref}) = \mathrm{newRef}(\mathrm{H}, \mathtt{E})$

If an exception $\mathtt{E}$ is thrown by instruction at position i while executing the method $\mathtt{m}$, the exception handler table $\mathtt{m}.\mathsf{excHndlS}$ will be searched for the first exception handler that can handle the exception. The search is done by the function *findExcHandler* . If there is found such a handler the function returns the index of the instruction at which the exception handler starts, otherwise it returns $\perp$:

$$\mathit{findExcHandler} \ : \mathit{ExcType} * \mathit{nat} * \textbf{ExcHandler}[] \to \mathit{nat}$$

$$\mathit{findExcHandler}(\ \mathtt{E}, \mathrm{Pc}, \mathsf{excH}[]) =$$
$$\begin{cases} \mathsf{excH}[m].\mathsf{handlerPc} & hExc \neq emptySet \Rightarrow min(hExc) = m \\ \\ \perp & hExc = emptySet \end{cases}$$

where
$$hExc = \{k \mid \begin{array}{l} \mathsf{excH}[k] = (\mathsf{startPc}, \mathsf{endPc}, \mathsf{handlerPc}, \mathtt{E}') \wedge \\ \mathsf{startPc} \leq \mathrm{Pc} < \mathsf{endPc} \wedge \\ \mathsf{subtype}\ (\mathtt{E}, \mathtt{E}') \end{array} \}$$

2.7 Bytecode Language and its Operational Semantics

The bytecode language that we introduce here corresponds to a representative subset of the Java bytecode language. In particular, it supports object manipulation and creation, method invokation, as well as exception throwing and handling. In fig. 2.1, we give the list of instructions that constitute our bytecode language.

Note that the instruction arith_op stands for any arithmetic instruction in the list add , sub , mult , and , or , xor , ishr , ishl , div , rem).

We define the operational semantics of a single Java instruction in terms of relation between the instruction and the state configurations before and after its execution.

$$
\begin{array}{rl}
\mathrm{I} ::= & \mathrm{if\_cond} \\
 & \mid \ \mathrm{goto} \\
 & \mid \ \mathrm{return} \\
 & \mid \ \mathrm{arith\_op} \\
 & \mid \ \mathrm{load} \\
 & \mid \ \mathrm{store} \\
 & \mid \ \mathrm{push} \\
 & \mid \ \mathrm{pop} \\
 & \mid \ \mathrm{dup} \\
 & \mid \ \mathrm{iinc} \\
 & \mid \ \mathrm{new} \\
 & \mid \ \mathrm{newarray} \\
 & \mid \ \mathrm{putfield} \\
 & \mid \ \mathrm{getfield} \\
 & \mid \ \mathrm{type\_astore} \\
 & \mid \ \mathrm{type\_aload} \\
 & \mid \ \mathrm{arraylength} \\
 & \mid \ \mathrm{instanceof} \\
 & \mid \ \mathrm{checkcast} \\
 & \mid \ \mathrm{athrow} \\
 & \mid \ \mathrm{invoke}
\end{array}
$$

Figure 2.1: Bytecode Language instructions

Definition 2.7.1 (State Transition) *If an instruction* I *in the body of method* `m` *starts execution in a state with configuration* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and terminates execution in state with configuration* $< \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$ *we denote this by*

$$\mathtt{m} \vdash \ \mathrm{I} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$$

We also define how the execution of a list of instructions change the state configuration in which their execution starts.

Definition 2.7.2 (Transitive closure of a method state transition relation) *If the method* `m` *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *with* `m.body`*[0] and there exists a transitive state transition to the state* $< \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$ *we denote this with:*

$$< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow^{*} < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$$

Definition 2.7.3 (Termination of method execution) *If the method* `m` *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *with* `m.body`*[0] and there is a transitive state transition to* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, k >$ *such that the instruction* `m.body`*[k] is either a* return *instruction or an instruction which terminates* *execution with an uncaught exception and the configuration after its execution is* $<\mathrm{H}', \mathrm{Reg}', \mathit{Final}>^{final}$ *then we denote this with:*

$$\mathtt{m} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \Rightarrow < \mathrm{H}', \mathrm{Reg}', \mathit{Final} >^{final}$$

We first give the operational semantics of a method execution. The execution of method `m`is the execution of its body upto reaching a final state configuration:

$$\frac{\mathtt{m} \vdash \mathtt{m.body}[0] :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow^{*} < \mathrm{H}', \mathrm{Reg}', \mathit{Final} >^{final}}{\mathtt{m} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \Rightarrow < \mathrm{H}', \mathrm{Reg}', \mathit{Final} >^{final}}$$

Next, we define the operational semantics of every instruction. The operational semantics of an instruction states how the execution of an instruction affects the program state configuration in terms of state configuration transitions defined in the previous subsection 2.5. Note that we do not model the method frame stack of the JVM which is not needed for our purposes.

- Control transfer instructions

1. Conditional jumps : if_cond

$$\frac{\mathtt{cond}(\mathrm{St}(\mathrm{Cntr}), \mathrm{St}(\mathrm{Cntr} - 1))}{\mathrm{m} \ \vdash \ \mathrm{if\_cond} \ \ n :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr} - 2, \mathrm{St}, \mathrm{Reg}, n >}$$

$$\frac{\mathit{not}(\ \mathtt{cond}(\mathrm{St}(\mathrm{Cntr}), \mathrm{St}(\mathrm{Cntr} - 1)))}{\mathtt{m} \vdash \ \mathrm{if\_cond} \ \ n :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr} - 2, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} + 1 >}$$

   The condition `cond` $= \{=, \neq, \leq, <, >, \geq\}$ is applied to the stack top St(Cntr) and the element below the stack top St(Cntr -1)which must be of type **int**. If the condition is true then the control is transfered to the instruction at index `n`, otherwise the control continues at the instruction following the current instruction. The top two elements St(Cntr) and St(Cntr - 1) of the stack top are popped from the operand stack.

2. Unconditional jumps: goto

$$\frac{}{\mathtt{m} \vdash \ \mathrm{goto} \ \ n :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, n >}$$

   Transfers control to the instruction at position *n*.

3. return

$$\frac{}{\mathtt{m} \vdash \ \mathrm{return} \ \ :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Reg}, \mathrm{St}(\mathrm{Cntr}) >^{norm}}$$

The instruction causes the normal termination of the execution of the current method `m`. The instruction does not affect changes on the heap Hand the return result is contained in the stack top element St(Cntr)

- Arithmetic operations

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} - 1\\ \mathrm{St}' = \mathrm{St}[\oplus\mathrm{Cntr} - 1 \rightarrow \mathrm{St}(\mathrm{Cntr})\ \mathrm{op}\ \mathrm{St}(\mathrm{Cntr} - 1)]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m}\ \vdash\ op :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

Pops the values which are on the stack top St(Cntr) and St(Cntr - 1) at the position below and applies the arithmetic operation `op` on them. The stack counter is decremented and the resulting value on the stack top St(Cntr - 1) op St(Cntr) is pushed on the stack top St(Cntr - 1).

- Load Store instructions

1. load

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus\mathrm{Cntr} + 1 \rightarrow \mathrm{Reg}(i)]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m}\ \vdash\ \mathrm{load}\ \ \mathrm{i} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

The instruction increments the stack counter Cntr and pushes the content of the local variable **reg**(i) on the stack top St(Cntr + 1)

2. store

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} - 1\\ \mathrm{Reg}' = \mathrm{Reg}[\oplus\ \mathrm{i} \rightarrow \mathrm{St}(\mathrm{Cntr})]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m}\ \vdash\ \mathrm{store}\ \ \mathrm{i} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}', \mathrm{St}, \mathrm{Reg}', \mathrm{Pc}' >}$$

Pops the stack top element St(Cntr) and stores it into local variable **reg**(i) and decrements the stack counter Cntr

3. iinc

$$\frac{\begin{array}{l}\mathrm{Reg}' = \mathrm{Reg}[\oplus\ \mathrm{i} \rightarrow \mathbf{reg}(\ \mathrm{i}) + 1]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m}\ \vdash\ \mathrm{iinc}\ \ \ \mathrm{i} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}', \mathrm{Pc}' >}$$

Increments the value of the local variable **reg**(i)

4. push

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus\mathrm{Cntr} + 1 \rightarrow\ \mathrm{i}]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m}\ \vdash\ \mathrm{push}\ \ \mathrm{i} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr} + 1, \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

5. pop

$$\frac{}{\mathrm{m}\ \vdash\ \mathrm{pop} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr} + 1, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} + 1 >}$$

- Object creation and manipulation

1. new `Cl`

$$\frac{\begin{array}{l} (\mathrm{H}', \mathtt{ref}) = \mathrm{newRef}(\mathrm{H}, C) \\ \mathrm{Cntr}' = \mathrm{Cntr} + 1 \\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \mathtt{ref}] \\ \mathrm{Pc}' = \mathrm{Pc} + 1 \end{array}}{\mathrm{m}\ \vdash\ \mathrm{new}\ C :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

A new fresh location `ref` is added in the memory heap H of type C, the stack counter Cntr is incremented. The reference `ref` is put on the stack top St(Cntr + 1).

2. putfield

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr} - 1) \neq \mathbf{null} \\ \mathrm{H}' = \mathrm{H}[\oplus f \rightarrow f[\oplus \mathrm{St}(\mathrm{Cntr} - 1) \rightarrow \mathrm{St}(\mathrm{Cntr})]] \\ \mathrm{Cntr}' = \mathrm{Cntr} - 2 \\ \mathrm{Pc}' = \mathrm{Pc} + 1 \end{array}}{\mathtt{m} \vdash\ \mathrm{putfield}\ f :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr} - 1) = \mathbf{null} \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >,\ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = K \end{array}}{\mathtt{m} \vdash\ \mathrm{putfield}\ f :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

The top value contained on the stack top St(Cntr) and the reference contained in St(Cntr - 1) are popped from the operand stack. If St(Cntr - 1) is not **null**[1] , the value of its field `f` for the object is updated with the valueSt(Cntr) and the counter Cntr is decremented. If the reference in St(Cntr - 1) is **null**then a `NullPntrExc` is thrown

3. getfield

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}) \neq \mathbf{null} \\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} \rightarrow \mathrm{H}(f)(\mathrm{St}(\mathrm{Cntr}))] \\ \mathrm{Pc}' = \mathrm{Pc} + 1 \end{array}}{\mathtt{m} \vdash\ \mathrm{getfield}\ f :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}) = \mathbf{null} \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >,\ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = K \end{array}}{\mathtt{m} \vdash\ \mathrm{getfield}\ f :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

The top stack element St(Cntr) is popped from the stack. If St(Cntr) is not **null**the value of the field `f` in the object referenced by the

[1] here we assume that the code has passed successfully the bytecode verification procedure and thus, for instance, St(Cntr - 1) contains certainly a reference of type `C`

reference contained in St(Cntr), is fetched and pushed onto the operand stack St(Cntr). If St(Cntr) is **null**then a `NullPointerExc` is thrown, i.e. the stack counter is set to 0, a new object of type `NullPointerExc` is created in the memory heap store Hand a reference to it $\texttt{ref}_{NullPointerExc}$ is pushed onto the operand stack

4. newarray `T`

$$\frac{\begin{array}{l} \mathrm{St(Cntr)} \geq 0 \\ (\mathrm{H}', \texttt{ref}) = \mathrm{newArrRef(H}, type, \mathrm{St(Cntr))} \\ \mathrm{Cntr}' = \mathrm{Cntr} + 1 \\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \texttt{ref}] \\ \mathrm{Pc}' = \mathrm{Pc} + 1 \end{array}}{\texttt{m} \vdash \text{ newarray } \texttt{T} :< \mathrm{H, Cntr, St, Reg, Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l} \mathrm{St(Cntr)} < 0 \\ getStateOnExc\ (< \mathrm{H, Cntr, St, Reg, Pc} >,\ \texttt{NegArrSizeExc}, \texttt{m}.\mathsf{excHndlS}) = K \end{array}}{\texttt{m} \vdash \text{ newarray } \texttt{T} :< \mathrm{H, Cntr, St, Reg, Pc} > \hookrightarrow K}$$

A new array whose components are of type `T` and whose length is the stack top value is allocated on the heap. The array elements are initialised to the default value of `T` and a reference to it is put on the stack top. In case the stack top is less than 0, then `NegArrSizeExc` is thrown

5. type_astore

$$\frac{\begin{array}{l} \mathrm{St(Cntr} - 2) \neq \mathbf{null} \\ 0 \leq \mathrm{St(Cntr} - 1) < \mathrm{arrLength(St(Cntr} - 2)) \\ \mathrm{H}' = \mathrm{H}[\oplus(\mathrm{St(Cntr} - 2), \mathrm{St(Cntr} - 1)) \rightarrow \mathrm{St(Cntr)}] \\ \mathrm{Cntr}' = \mathrm{Cntr} - 3 \\ \mathrm{Pc}' = \mathrm{Pc} + 1 \end{array}}{\texttt{m} \vdash \text{ type\_astore } :< \mathrm{H, Cntr, St, Reg, Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l} \mathrm{St(Cntr} - 2) = \mathbf{null} \\ getStateOnExc\ (< \mathrm{H, Cntr, St, Reg, Pc} >,\ \texttt{NullPntrExc}, \texttt{m}.\mathsf{excHndlS}) = K \end{array}}{\texttt{m} \vdash \text{ type\_astore } :< \mathrm{H, Cntr, St, Reg, Pc} > \hookrightarrow K}$$

$$\frac{\begin{array}{l} \mathrm{St(Cntr} - 2) \neq \mathbf{null} \\ (\mathrm{St(Cntr} - 1) < 0 \vee \\ \mathrm{St(Cntr} - 1) \geq \mathrm{arrLength(St(Cntr} - 2))) \Rightarrow \\ getStateOnExc\ (< \mathrm{H, Cntr, St, Reg, Pc} >,\ \texttt{ArrIndBndExc}, \texttt{m}.\mathsf{excHndlS}) = K \end{array}}{\texttt{m} \vdash \text{ type\_astore } :< \mathrm{H, Cntr, St, Reg, Pc} > \hookrightarrow K}$$

The three top stack elements St(Cntr), St(Cntr - 1) and St(Cntr - 2) are popped from the operand stack. The type value contained in St(Cntr) must be assignment compatible with the type of the elements of the array reference contained in St(Cntr - 2), St(Cntr - 1) must be of type int.

The value St(Cntr) is stored in the component at index St(Cntr - 1) of the array in St(Cntr - 2). If St(Cntr - 2) is **null**a `NullPntrExc`is thrown. If St(Cntr - 1) is not in the bounds of the array in St(Cntr - 2) an `ArrIndBndExc` exception is thrown. If St(Cntr) is not assignment compatible with the type of the components of the array, then `ArrStoreExc` is thrown

6. type_aload

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}-1) \neq \mathbf{null} \\ \mathrm{St}(\mathrm{Cntr}) \geq 0 \\ \mathrm{St}(\mathrm{Cntr}) < \mathrm{arrLength}(\mathrm{St}(\mathrm{Cntr}-1)) \\ \mathrm{Cntr}' = \mathrm{Cntr}-1 \\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr}-1 \rightarrow \mathrm{H}(\mathrm{St}(\mathrm{Cntr}-1)\mathrm{St}(\mathrm{Cntr}))] \\ \mathrm{Pc}' = \mathrm{Pc}+1 \end{array}}{\mathtt{m} \vdash \ \mathrm{type\_aload} \ :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}-1) = \mathbf{null} \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = K \end{array}}{\mathtt{m} \vdash \ \mathrm{type\_aload} \ :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}-1) \neq \mathbf{null} \\ (\mathrm{St}(\mathrm{Cntr}) < 0 \vee \\ \mathrm{St}(\mathrm{Cntr}) \geq \mathrm{arrLength}(\mathrm{St}(\mathrm{Cntr}-1))) \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \ \mathtt{ArrIndBndExc}, \mathtt{m}.\mathsf{excHndlS}) = K \end{array}}{\mathtt{m} \vdash \ \mathrm{type\_aload} \ :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

Loads a value from an array. The top stack element St(Cntr) and the element below it St(Cntr -1) are popped from the operand stack. St(Cntr) must be of type **int**. The value in St(Cntr -1) must be of type *RefClType* whose components are of type type. The value in the component of the array `arrRef` at index `ind` is retrieved and pushed onto the operand stack. If St(Cntr -1) contains the value **null**a `NullPntrExc`is thrown. If St(Cntr) is not in the bounds of the array object referenced by St(Cntr -1) a `ArrIndBndExc` is thrown

7. arraylength

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}) \neq \mathbf{null} \\ \mathrm{H}' = \mathrm{H} \\ \mathrm{Cntr}' = \mathrm{Cntr} \\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} \rightarrow \mathrm{H}(\mathrm{arrLength})(\mathrm{St}(\mathrm{Cntr}))] \\ \mathrm{Pc}' = \mathrm{Pc}+1 \end{array}}{\mathtt{m} \vdash \ \mathrm{arraylength} \ :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l} \mathrm{St}(\mathrm{Cntr}) = \mathbf{null} \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = K \end{array}}{\mathtt{m} \vdash \ \mathrm{arraylength} \ :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

The stack top element is popped from the stack. It must be a reference that points to an array. If the stack top element St(Cntr) is not **null**the length of the array arrLengthSt(Cntr) is fetched and pushed on the stack. If the stack top element St(Cntr) is **null**then a `NullPntrExc`is thrown.

8. instanceof

$$\frac{\begin{array}{l}\mathsf{subtype}\ (\mathrm{H}.\mathsf{TypeOf}\ (\mathrm{St}(\mathrm{Cntr})), C)\\ \mathrm{St}' = \mathrm{St}[\oplus\mathrm{Cntr} \rightarrow 1]\\ \mathrm{Pc}' = \mathrm{Pc}+1\end{array}}{\mathrm{instanceof}\ C :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l}not(\mathsf{subtype}\ (\mathrm{H}.\mathsf{TypeOf}\ (\mathrm{St}(\mathrm{Cntr})), C)) \vee \mathrm{St}(\mathrm{Cntr}) = \mathbf{null}\\ \mathrm{St}' = \mathrm{St}[\oplus\mathrm{Cntr} \rightarrow 0]\\ \mathrm{Pc}' = \mathrm{Pc}+1\end{array}}{\mathtt{m} \vdash\ \mathrm{instanceof}\ \mathsf{C} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

The stack top is popped from the stack. If it is of subtype Cor is **null**, then the `1` is pushed on the stack, otherwise `0`.

9. checkcast

$$\frac{\begin{array}{l}\mathsf{subtype}\ (\mathrm{H}.\mathsf{TypeOf}\ (\mathrm{St}(\mathrm{Cntr})), C) \vee \mathrm{St}(\mathrm{Cntr}) = \mathbf{null}\\ \mathrm{Pc}' = \mathrm{Pc}+1\end{array}}{\mathtt{m} \vdash\ \mathrm{checkcast}\ C :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l}not(\mathsf{subtype}\ (\mathrm{H}.\mathsf{TypeOf}\ (\mathrm{St}(\mathrm{Cntr})), C))\\ getStateOnExc\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \mathtt{CastExc}, \mathtt{m}.\mathsf{excHndlS}) = K\end{array}}{\mathtt{m} \vdash\ \mathrm{checkcast}\ C :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

The stack top is popped from the stack. If it is not of subtype C an exception of type `CastExc`is thrown.

- Throw exception instruction. athrow

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr}) \neq \mathbf{null}\\ getStateOnExc\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, typeof(\mathrm{St}(\mathrm{Cntr})), \mathtt{m}.\mathsf{excHndlS}) = K\end{array}}{\mathtt{m} \vdash\ \mathrm{athrow} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr}) = \mathbf{null}\\ getStateOnExc\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >,\ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = K\end{array}}{\mathtt{m} \vdash\ \mathrm{athrow} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow K}$$

The stack top element must be a reference of an object of type `Throwable`. If there is a handler that protects this bytecode instruction from the exception thrown, the control is transfered to the instruction at which the exception handler starts[2]. If the object on the stack top is **null**, a `NullPntrExc` is thrown.

- Method Invokation. invoke [3]

[2] for every method the ExceptionHandler table describes the corresponding exception handler by the limits of the region it protects, the Exception that it catches, and the instruction at which it starts

[3] only the case when the invoked method returns a value

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr} - \mathit{meth}.\mathsf{nArgs}) \neq \mathbf{null} \\ \mathit{meth} :< \mathrm{H}, 0, [\ ], [\mathrm{St}(\mathrm{Cntr} - \mathit{meth}.\mathsf{nArgs}), \ldots, \mathrm{St}(\mathrm{Cntr})], 0 > \Rightarrow < \mathrm{H}', \mathrm{Reg}', \mathrm{Res} >^{norm} \\ \\ \mathrm{Cntr}' = \mathrm{Cntr} - \mathtt{m}.\mathsf{nArgs} + 1 \\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr}' \rightarrow \mathrm{Res}] \\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathtt{m} \vdash \ \text{invoke } \mathit{meth} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}, \mathrm{Pc}' >}$$

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr} - \mathit{meth}.\mathsf{nArgs}) \neq \mathbf{null} \\ \mathit{meth} :< \mathrm{H}, 0, [\ ], [\mathrm{St}(\mathrm{Cntr} - \mathit{meth}.\mathsf{nArgs}), \ldots, \mathrm{St}(\mathrm{Cntr})], 0 > \Rightarrow < \mathrm{H}', \mathrm{Reg}', \mathrm{Exc} >^{exc} \\ \Rightarrow \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \mathit{typeof}(\mathrm{Exc}), \mathtt{m}.\mathsf{excHndlS}) = K\end{array}}{\mathtt{m} \vdash \ \text{invoke } \mathit{meth} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow \ K}$$

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr} - \mathit{meth}.\mathsf{nArgs}) = \mathbf{null} \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = K\end{array}}{\mathtt{m} \vdash \ \text{invoke } \mathit{meth} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow \ K}$$

The first top $\mathit{meth}.\mathsf{nArgs}$ elements in the operand stack St are popped from the operand stack. If St(Cntr - meth.nArgs) is not **null**, the invoked method is executed on the object St(Cntr - meth.nArgs) and where the first nArgs+ 1 elements of the list of its of local variables is initialised with St(Cntr - meth.nArgs) . . . St(Cntr). In case that the execution of method *meth* terminates normally, the return value Res of its execution is stored on the operand stack of the invoker. If the execution of of method *meth* terminates because of an exception Exc, then the exception handler of the invoker is searched for a handler that can handle the exception. In case the object St(Cntr - meth.nArgs) on which the method *meth* must be called is **null**, a `NullPntrExc`is thrown.

Chapter 3

Specification language for Java bytecode programs

3.1 Introduction

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

Before going further, we discuss what advocates the need of a low level specification language. Traditionally, specification languages were tailored for high level languages. Source specification allows to express complex functional or security properties about programs. Thus, they are / 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 3.2, we give an overview of the main features of JML. A detailed overview of BML is given in section 3.3. As we stated before, we support also a compiler from the high level specification language JML into BML. The compilation process from JML to BML is discussed in section 3.5. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

3.2 A quick overview of JML

JML [14] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications. JML follows the design-by-contract approach (see [7]), 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 [9]). 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.* [8]). 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 [10]. 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 [18] or dynamic checking by tools such as the assertion checker jmlrac [11]. An overview of the JML tools can be found in [9].

Figure 3.1 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 3.1: CLASS ListArray WITH JML ANNOTATIONS

3.3 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. 3.3.2 gives the formal grammar of BML. The formal grammar of BML is given in the next definition.

3.3.1 Notation convention

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

3.3.2 BML Grammar

$$
\begin{array}{ll}
\mathit{constants}_{bml} & ::= \mathit{intLiteral} \mid \mathit{signedIntLiteral} \mid \mathbf{null} \mid \mathit{ident} \\
\mathit{signedIntLiteral} & ::= +\mathit{nonZerodigit}[\mathit{digits}] \mid -\mathit{nonZerodigit}[\mathit{digits}] \\
\mathit{intLiteral} & ::= \mathit{digit} \mid \mathit{nonZerodigit}[\mathit{digits}] \\
\mathit{digits} & ::= \mathit{digit}[\mathit{digits}] \\
\mathit{digit} & ::= \mathbf{0} \mid \mathit{nonZerodigit} \\
\mathit{nonZerodigit} & ::= \mathbf{1} \mid \ldots \mid \mathbf{9} \\
\mathit{ident} & ::= \#\ \mathit{intLiteral} \\
\mathit{boundVar} & ::= \ \mathbf{bv\_}\mathit{intLiteral}
\end{array}
$$

$$
\begin{array}{ll}
E_{bml} & ::= \mathit{constants}_{bml} \\
 & \mid \mathbf{reg}(\mathit{digits}) \\
 & \mid E_{bml}\mathbf{.}\mathit{ident} \\
 & \mid \mathit{ident} \\
 & \mid \mathbf{arrayAccess}(E_{bml}, E_{bml}) \\
 & \mid E_{bml}\ \mathit{op}\ E_{bml} \\
 & \mid \mathbf{cntr} \\
 & \mid \mathbf{st}(E_{bml}) \\
 & \mid \backslash\mathbf{old(}E_{bml}\mathbf{)} \\
 & \mid \backslash\mathbf{EXC} \\
 & \mid \backslash\mathbf{result} \\
 & \mid \mathit{boundVar}
\end{array}
$$

$$\begin{array}{ll}
T_{bml} & ::= \backslash\mathbf{typeof(}E_{bml}\mathbf{)} \\
 & \mid \backslash\mathbf{type(}\mathit{ident}\mathbf{)} \\
 & \mid \backslash\mathbf{elemtype(}E_{bml}\mathbf{)} \\
 & \mid \backslash\mathbf{TYPE}
\end{array}$$

$$\begin{array}{ll}
\mathit{SpecExp}_{bml} & ::= E_{bml} \\
 & \mid T_{bml}
\end{array}$$

$$\begin{array}{ll}
\mathit{op} & ::= \mathbf{+} \mid \mathbf{-} \mid \mathbf{mult} \mid \mathbf{div} \mid \mathbf{rem}
\end{array}$$

$$\begin{array}{ll}
\mathcal{R} & ::= = \mid \neq \mid \leq \mid \leq \mid \geq \mid > \mid <:
\end{array}$$

$$\begin{array}{ll}
\mathcal{P}_{bml} & ::= E_{bml}\ \mathcal{R}\ E_{bml} \\
 & \mid \mathbf{true} \\
 & \mid \mathbf{false} \\
 & \mid \mathit{not}\ \mathcal{P}_{bml} \\
 & \mid \mathcal{P}_{bml} \wedge \mathcal{P}_{bml} \\
 & \mid \mathcal{P}_{bml} \vee \mathcal{P}_{bml} \\
 & \mid \mathcal{P}_{bml} \Rightarrow \mathcal{P}_{bml} \\
 & \mid \mathcal{P}_{bml} \iff \mathcal{P}_{bml} \\
 & \mid \forall\ \mathit{boundVar}, \mathcal{P}_{bml} \\
 & \mid \exists\ \mathit{boundVar}, \mathcal{P}_{bml}
\end{array}$$

$$\begin{array}{ll}
\mathit{classSpec} & ::= \mathbf{ClassInv}\ \mathcal{P}_{bml} \\
 & \mid \mathbf{ClassHistoryConstr}\ \mathcal{P}_{bml} \\
 & \mid \mathbf{declare\ ghost}\ \mathit{ident}\ \mathit{ident}
\end{array}$$

$$\begin{array}{lll}
\mathit{intraMethodSpec} & ::= & \begin{array}{l} \mathbf{atIndex}\ \mathit{nat}; \\ \mathit{assertion}; \end{array}
\end{array}$$

$$\begin{array}{ll}
\mathit{assertion} & ::= \mathit{loopSpec} \\
 & \mid \mathbf{assert}\ \mathcal{P}_{bml} \\
 & \mid \mathbf{set}\ E_{\mathrm{bml}}\ E_{\mathrm{bml}}
\end{array}$$

$$\begin{array}{lll}
\mathit{loopSpec} & ::= & \begin{array}{l} \mathbf{loopInv}\ \mathcal{P}_{bml}; \\ \mathbf{loopModif}\ \mathit{list}; \\ \mathbf{loopDecreases}\ E_{bml}; \end{array}
\end{array}$$

$$\begin{array}{ll}
\mathit{methodSpec} & ::= \mathit{specCase} \\
 & \mid \mathit{specCase}\ \mathbf{also}\ \mathit{methodSpec}
\end{array}$$

$$\begin{array}{lll}
\mathit{specCase} & ::= & \begin{array}{l} \mathbf{requires}\ \mathcal{P}_{bml}; \\ \mathbf{modifies}\ \mathit{list}\ \mathit{locations}; \\ \mathbf{ensures}\ \mathcal{P}_{bml}; \\ \mathit{exsuresList} \end{array}
\end{array}$$

$$\begin{array}{ll}
\mathit{exsuresList} & ::= [] \mid \mathbf{exsures}\ (\mathit{ident})\ \mathcal{P}_{bml}; \mathit{exsuresList}
\end{array}$$

$$\begin{array}{ll}
\mathit{locations} & ::= E_{bml}\mathbf{.}\mathit{ident} \\
 & \mid \mathbf{reg}(i) \\
 & \mid \mathit{arrayModAt}(E_{bml}, \mathit{specIndex}) \\
 & \mid \mathbf{everything} \\
 & \mid \mathbf{nothing}
\end{array}$$

$$\begin{array}{ll}
\mathit{specIndex} & ::= \mathrm{all} \mid i_1..i_2 \mid i
\end{array}$$

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

3.3.3 Informal semantics of BML

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. As we will see hereafter, BML allows to express field access, array access, method parameters and local variables, stack expressions etc. The rule that produces the BML expression corresponds to the nonterminal $\mathit{SpecExp}_{bml}$. As we can see from the rule, we divide the expressions into two categories : E_{bml} and T_{bml}.

Note that few of the expressions that are generated by the nonterminal E_{bml} do not have analogs in JML. We first focus on the expressions produced by E_{bml} and particularly those that have a translation in JML :

- $\mathit{constants}_{bml}$ represents the constants in BML. A constant is either a signed or unsigned integer, or an identificator. Integers are defined in a standard way. Identificators correspond to indexes in the constant pool of a Java class and are always prefixed by the symbol #.

- $\textbf{reg}(i)$ 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.

- E_{bml}**.***ident* stands for accessing the field which is at index *ident* in the class constant pool. for the reference denoted by the expression E_{bml}.

- **arrayAccess**(E^1_{bml}, E^2_{bml}) stands for an access to the element at index E^2_{bml} in the array denoted by the expression E^1_{bml}. This corresponds to the Java notation $E^1_{bml}[E^2_{bml}]$

- E_{bml} *op* E_{bml} stands for the usual arithmetic operations. *op* ranges over the standard arithmetic operations $+, -, *, div, rem$

- **\old(**E_{bml}**)** denotes the value of E_{bml} 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 are produced by the nonterminal E_{bml} and which 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.

- **st**(E_{bml}) stands for the element in the operand stack at position E_{bml} 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 **st**$(\mathbf{cntr} - 1)$

Finally, the expressions generated by the nonterminal T_{bml} are:

- **\typeof(**E_{bml}**)** denotes the dynamic type of the expression E_{bml}

- **\type(***ident***)** denotes the class described at index *ident* in the constant pool of the corresponding class file

- **\elemtype(**E_{bml}**)** denotes the type of the elements of the array E_{bml}

- **\TYPE**

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 $\mathcal{P}_{bml}$. From the formal syntax, we can notice that BML supports the standard logical connectors $\wedge, \vee, \Rightarrow$, existential $\exists$ and universal quantification $\forall$ as well as standard relation between the expressions of our language like $\neq, =, \leq, \leq \ldots$

```
class C {
  int a ;

  /@*
    * invariant a > 0;
    * historyConstraint old(a) >= a;
    *@/
  public void decrease(int b) {
     ...
  }
}
```

Figure 3.2: An example for class specification

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:

- **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.3.2 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.

Method specification case

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

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

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

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

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

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 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. 3.3 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 3.3.2, BML allows to specify a property that must hold at particular program point inside a method body. The nonterminal which describes the grammar of assertions is *intraMethodSpec*. Let us see in detail what kind of specifications can be supported in BML:

- **atIndex** *nat* specifies the index of the instruction which identifies the instruction to which the specification refers. 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. 3.1 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

```
class C {
   int a ;

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

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

Figure 3.3: An example for an intra method specification

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** $\mathcal{P}_{bml}$ where $\mathcal{P}_{bml}$ is the property that must hold whenever the corresponding loop entry instruction is reached during execution
    - **loopModif** *list loc* is the list of locations modified in the loop. This means that at the borders of every iteration (beginning and end), all the expressions not mentioned in the loop frame condition must have the same value.
    - **loopDecreases** E_{bml} specifies the expression E_{bml} which guarantees loop termination. The values of E_{bml} must be from a well founded set (usually from **int** type) and the values of E_{bml} should decrease at every iteration

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

Frame conditions

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

- E_{bml}**.**$ident$ states that the method / loop modifies the value of the field at index *ident* in the constant pool for the reference denoted by E_{bml}

- **reg**(i) states that the local variable may modified by a loop. Note that this kind of modified expression makes sense only for expressions modified in a loop. However a modification of a local variable does not make sense for a method frame condition, as methods in Java are called by value, and thus, a method can not cause a modification of a local variable that is observable by the rest of the program.

- $arrayModAt(E_{bml}, specIndex)$ states that the components at the indexes specified by *specIndex* in the array denoted by E_{bml} may be modified. The indexes of the array components that may be modified *specIndex* have the following syntax:

  - i is the index of the component at index i. For instance, $arrayModAt(E_{bml}, i)$ means that the array component at index i might be modified. Of course, in order that such a specification make sense the following must hold: $0 \leq i < \text{arrLength}(E_{bml})$
  - all specifies that all the components of the array may be modified, i.e. the expression $arrayModAt(E_{bml}, \text{all})$ is a syntactic sugar for

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

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

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

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

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

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

3.4 Well formed BML specification

In the previous Section 3.3, we gave the formal grammar of BML. However, we are interested in a strict subset of the specifications that can be generated from this grammar. In particular, we want that a BML specification is well typed and respects few structural constraints.

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

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

Example for structural constraint are :

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

Actually, an extension of the bytecode verifier may perform the checks if a BML specification respects this kind of structural and type constraints. However, we are not going farther in this subject as it is out of the scope of the present thesis. For the curious reader, it will be certainly of interest to turn to the Java Virtual Machine specification [20] which contains the official specification of the Java bytecode verifier or to the existing literature on bytecode verification (see the overview article [19])

3.5 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([20], 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 [20], 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 [14]). 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 [25] For instance, a Java method which has two normal behaviours is given in Fig. 3.4. Its desugared form corresponds to the method given in Fig. 3.3

3. Linking with source data structures
   When the JML specification is desugared, we are ready for the linking and resolving phases. In this stage, the JML specification gets into an intermediate format in which the identifiers are resolved to data structures standing for the data that it represents. For instance, consider once again the example in Fig. 3.4 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

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

```
public class C {
   int a ;

   invariant a > 0;
   historyConstraint  old(a) >= 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 3.4: An example for a method with two normal behaviours specified in JML

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

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

Figure 3.5: The compilation of the postcondition in Fig. 3.1

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.3.1 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. 3.5. 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.3.1 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. 3.6. 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

[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

```
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 3.6: Structure of the Loop Attribute

modified in a loop iteration, the invariant associated to this loop and the decreasing expression in case of total correctness,

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

Chapter 4

Verification condition generator for Java bytecode

This section describes a Hoare style verification condition generator for bytecode based on a weakest precondition predicate transformer function.

A natural question is to ask what are the motivations behind building a bytecode verification condition generator (vcGen for short) while a considerable list of tools for source code verification exists. We consider that today's software industry requires more and more guarantees about software security especially when mobile computing becomes a reality. Thus in mobile code scenarios, performing verification on source code of untrusted executable unit requires a trust in the compiler but which is not always reasonable. On the other hand, type based verification used for example, in the Java bytecode verifier could not deal with complex functional or security properties which is the case for a verification condition generator. The vcGen is tailored to the bytecode language introduced in Section 2.7 and thus, it deals with stack manipulation, object creation and manipulation, field access and update, as well as exception throwing and handling.

Bytecode verification has become lately quite fashionable, thus several works exist on bytecode verification. Section 4.1 is an overview of the existing work in the domain.

Performing Hoare style logic verification over an unstructured program like bytecode programs has few particularities which verification of structured programs lacks. For example loops on source level correspond to a syntactic structure in the source language and thus, identifying a loop in a source program is not difficult. However, this is not the case for unstructured programs. As we saw in the previous section 3.1, our approach consists in compiling source specification into bytecode specification. When compiling a loop invariant, we need to know where exactly in the bytecode the invariant must hold. Section 4.3 introduces the notion of a loop in an unstructured program.

As we stated earlier, our verification condition generator is based on a weakest precondition (wp) calculus. As we shall see in Section 4.5 a wp function for bytecode is similar to a wp function for source code. However, a logic tailored to stack based bytecode should take into account particular bytecode features as for example the operand stack.

4.1 Related work

In the following, we review briefly the existing work related to program verification and more particularly program verification tailored to Java and Java bytecode programs.

Floyd is among the first to work on program verification using logic methods for unstructured program languages (see [26]). Following the Floyd's approach, T.Hoare gives a formal logic for program verification in [15] known today under the name Hoare logic. Dijkstra [12] proposes then an efficient way for applying Hoare logic in program verification, i.e. he comes up with a weakest precondition (wp) and strongest postcondition (sp) calculi.

Concerning bytecode validation, there exists several approaches depending on the kind of properties that one want to check for.

Bytecode verification is concerned with establishing that a bytecode is well typed (every instruction is applied to operands of the correct type) and well formed (e.g. no jumps to an un-existing bytecode index), differently from the goals of the present work where program correctness is defined in terms of functional correctness. The JVM, for example, is provided with a bytecode verifier. There is a lot of research work done in the domain and for a detailed overview of the state of the art one can look at [19].

As Java has been gaining popularity in industry since the nineties of the twentieth century, it also attracted the research interest. Thus the nineties upto nowadays give rise to several verification tools tailored to Java based on Hoare logic. Among the ones that gained most popularity are esc/java developed at Compaq [18], the Loop tool [16], Krakatoa, Jack [10] etc.

Few works have been dedicated to the definition of a bytecode logic. May be the earliest work in the field of bytecode verification is the thesis of C.Quigley [24] in which Hoare logic rules are given for a bytecode like language. This work is limited to a subset of the Java virtual machine instructions and does not treat for example method calls, neither exceptional termination. The logic is defined by searching a structure in the bytecode control flow graph, which gives an issue to complex and weak rules.

The work by Nick Benton [6] gives a typed logic for a bytecode language with stacks and jumps. The technique that he proposes checks at the same time types and specifications. The language is simple and supports basically stack and arithmetic operations. Finally, a proof of correctness w.r.t. an operational semantics is given.

Following the work of Nick Benton, Bannwart and Muller [2] give a Hoare logic rules for a bytecode language with objects and exceptions. A compiler from source proofs into bytecode proofs is also defined. As in our work, they assume that the bytecode has passed the bytecode verification certification. The bytecode logic aims to express functional properties. Invariants are inferred by fixpoint calculation. However, inferring invariants is not a decidable problem.

In [28], M. Wildmoser and T. Nipkow describe a framework for verifying Jinja (a Java subset) bytecode against arithmetic overflow. The annotation is written manually, which is not comfortable, especially on bytecode. Here we propose a way to compile a specification written in a high level language, allowing specification to be written at source level, which we consider as more convenient.

The Spec# ([3]) programming system developed at Microsoft proposes a static verification framework where the method and class contracts (pre, post conditions, exceptional postconditions, class invariants) are inserted in the intermediate code . Spec# is a superset of the C# programming language, with a built-in specification language, which proposes a verification framework (there is a choice to perform the checks either at runtime or statically). The static verification procedure involves translation of the contract specification into metadata which is attached to the intermediate code. The verification procedure [21] that is performed includes several stages of processing the bytecode program: elimination of irreducible loops, transformation into an acyclic control flow graph, translation of the bytecode into a guarded passive command language program. Despite that here in our implementation we also do a transformation in the graph into an acyclic program, we consider that in a mobile code scenario one should limit the number of program transformations for several reasons. First, we need a verification procedure as simple as possible, and second every transformation must be proven correct which is not always trivial.

4.2 The expression language

In the following, we will introduce a deep encoding of the expressions and predicates over which the *wp* calculus will be defined. Most of the expressions are directly taken from the specification language BML introduced in Chapter 3.1. However, there are several construct which does not belong to the BML grammar. We keep the same set of predicates as in BML. The next definition gives the set of expressions and formulas.

Definition 4.2.1 (Language of expressions)

$$
\begin{array}{ll}
\mathit{constants} & ::= \mathit{Values} \mid \mathbf{Class} \\
 & \\
\mathit{Values} & ::= i, i \in \ \mathbf{int}\ \text{literal} \ \mid \mathit{RefVal} \\
 & \\
\mathit{RefVal} & ::= ref \mid \mathit{RefValArr} \mid \mathbf{null} \\
 & \\
\mathit{RefValArr} & ::= refArr \\
 & \\
E & ::= \mathit{constants} \\
 & \mid \mathbf{reg}(nat) \\
 & \mid E.f, f : \mathbf{Field} \\
 & \mid f[\oplus E \rightarrow E](E), f : \mathbf{Field} \\
 & \mid \mathbf{arrayAccess}(E, E) \\
 & \mid \text{arrAccess}[\oplus(E, E) \rightarrow E](E, E) \\
 & \mid E \ \ op \ \ E \\
 & \mid \mathbf{cntr} \\
 & \mid \mathbf{st}(E) \\
 & \mid \backslash\mathbf{EXC} \\
 & \mid \backslash\mathbf{result} \\
 & \mid \backslash\mathbf{old}(E) \\
 & \\
T & ::= \backslash\mathbf{typeof}(E) \\
 & \mid \backslash\mathbf{type}(E) \\
 & \mid \backslash\mathbf{elemtype}(E) \\
 & \mid \backslash\mathbf{TYPE} \\
 & \\
\mathit{Expr} & ::= E \mid T \\
 & \\
op & ::= \mathbf{+} \mid \mathbf{-} \mid \mathbf{mult} \mid \mathbf{div} \mid \mathbf{rem} \\
 & \\
\mathcal{R} & ::= \ = \mid \neq \mid \leq \mid \leq \mid \geq \mid > \mid <: \\
 & \\
P & ::= \mathit{Expr} \ \mathcal{R} \ \mathit{Expr} \\
 & \mid \mathbf{instances}(\mathit{RefVal}) \\
 & \mid \mathbf{true} \\
 & \mid \mathbf{false} \\
 & \mid \neg \ P \\
 & \mid P \wedge P \\
 & \mid P \vee P \\
 & \mid P \Rightarrow P \\
 & \mid P \iff P \\
 & \mid \forall \ x, P \\
 & \mid \exists \ x, P
\end{array}
$$

From the above definition, that the constants in the language are the values as defined in Chapter 2.7, section 2.4 as well the set of classes **Class**. Note that the expression language also supports update expressions $f[\oplus E \rightarrow E](E)$ and $\text{arrAccess}[\oplus(E,E) \rightarrow E](E,E)$ for field and array access. These expressions appear in the intermediate states of the *wp* calculus. In the grammar for formulas which corresponds to the nonterminal P, we can see that we introduce the predicate **instances** over reference values. Informally, **instances**(r) means that r corresponds to an object which is allocated in the heap in the initial state of the method execution. In the following, we will proceed with subsection 4.2.1 which discusses how substitution is done. In section 4.2.2, we give a meaning of formulas from our assertion language in a state.

4.2.1 Substitution

Expression substitution is defined inductively in a standard way over the expression structure. Still, we allow also substitution over objects that are not from our language, i.e. we apply substitution over field objects which results in an update version of the field. Such a substitution has the form :

$$\mathit{Expr}[f \leftarrow f[\oplus \mathit{Expr} \rightarrow \mathit{Expr}]]$$

This substitution does not affect any of the ground expressions,, i.e. it does not affect local variables ($\mathbf{reg}(i)$), the constants of our language (*constants*), the stack counter (**cntr**), the result expression (**result**), the thrown exception instance variable (**EXC**). For instance, the following substitution does not change $\mathbf{reg}(1)$:

$$\mathbf{reg}(1)[f \leftarrow f[\oplus \mathit{Expr} \rightarrow \mathit{Expr}]] = \mathbf{reg}(1)$$

Field substitution affects only field objects as we see in the following:

$$f^1[f^2 \leftarrow f^2[\oplus \mathit{Expr}^1 \longrightarrow \mathit{Expr}^2]] =$$

$$\begin{cases} if\ f^1 \neq f^2\ then\ f^1 \\ \\ else\ if\ f^1 = f^2\ then\ f^2[\oplus \mathit{Expr}^1 \longrightarrow \mathit{Expr}^2] \end{cases}$$

$$f^1[\oplus \mathit{Expr}^1 \rightarrow \mathit{Expr}^2][f^2 \leftarrow f^2[\oplus \mathit{Expr}^3 \rightarrow \mathit{Expr}^4]] =$$

$$\begin{cases} if\ f^1 \neq f^2\ then \\ f^1[\oplus \mathit{Expr}^1[f^2 \leftarrow f^2[\oplus \mathit{Expr}^3 \rightarrow \mathit{Expr}^4]] \rightarrow \mathit{Expr}^2[f^2 \leftarrow f^2[\oplus \mathit{Expr}^3 \rightarrow \mathit{Expr}^4]]] \\ \\ else\ f^1 = f^2\ then \\ f^1 \begin{array}{l} [\oplus \mathit{Expr}^1[f^2 \leftarrow f^2[\oplus \mathit{Expr}^3 \rightarrow \mathit{Expr}^4]] \longrightarrow \mathit{Expr}^2[f^2 \leftarrow f^2[\oplus \mathit{Expr}^3 \rightarrow \mathit{Expr}^4]]] \\ [\oplus \mathit{Expr}^3 \longrightarrow \mathit{Expr}^4] \end{array} \end{cases}$$

For example, consider the following substitution expression:

$$f(\mathbf{reg}(1))[f \leftarrow f[\oplus \mathbf{reg}(2) \rightarrow 3]]$$

This results in the new expression :

$$f[\oplus \mathbf{reg}(2) \rightarrow 3](\mathbf{reg}(1))$$

The same kind of substitution is allowed for array access expressions, where the array object arrAccess can be updated.

4.2.2 Interpretation

We discuss the evaluation of expressions and interpretation of predicates in a particular program state configuration. Thus, we first define a function for expression evaluation, as well as a function which for a given state and predicate returns the interpretation of the given predicate in the given state. The function *eval* which evaluates expressions in a state has the following signature:

$$eval : Expr \rightarrow K \rightarrow K \rightarrow Values \cup JType \cup \bot$$

Note that the evaluation function takes as arguments an expression ($Expr$) of the assertion language presented in the previuos Section 4.2, the current state as well as the initial state of the current method and returns a value as defined in Section 2.4.

Definition 4.2.2.1 (Evaluation of expressions) *The evaluation in a state* $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *or* $s =< \mathrm{H}, \mathrm{Reg}, Final >^{final}$ *of an expression Expr w.r.t. an initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *is denoted with* $eval(Expr, s, s_0)$ *and is defined inductively over the grammar of expressions Expr as follows:*

$$eval(v, s, s_0) = v$$
$$where\ v \in \ \mathbf{int}\ \vee\ v \in RefVal$$

$$eval(f(E), s, s_0) =$$
$$= \mathrm{H}(f)(\ eval(E, s, s_0))$$

$$eval(f[\oplus E_1 \rightarrow E_2](E_3), s, s_0) =$$
$$= \mathrm{H}[\oplus f \rightarrow f[\oplus\ eval(E_1, s, s_0) \rightarrow\ eval(E_2, s, s_0)]](f)(\ eval(E_3, s, s_0))$$

$$eval(\mathbf{arrayAccess}(E_1, E_2), s, s_0) =$$
$$= \mathrm{H}(\ eval(E_1, s, s_0),\ eval(E_2, s, s_0))$$

$$eval(\mathrm{arrAccess}[\oplus(E_1, E_2) \rightarrow E_3](E_4, E_5), s, s_0) =$$
$$= \mathrm{H}[\oplus(\ eval(E_1, s, s_0),\ eval(E_2, s, s_0)) \rightarrow\ eval(E_3, s, s_0)]$$
$$(\ eval(E_4, s, s_0),\ eval(E_5, s, s_0))$$

$$eval(\mathbf{reg}(i), s, s_0) = \mathrm{Reg}(i)$$

$$eval(\backslash\mathbf{old(}E\mathbf{)}, s, s_0) = \ eval(E, s_0, s_0)$$

$$eval(E_1 \ op \ E_2, s, s_0) = \ eval(E_1, s, s_0) op \ eval(E_2, s, s_0)$$

$$eval(\backslash\mathbf{typeof(}E\mathbf{)}, s, s_0) = \\ \begin{cases} \textbf{int} & eval(E, s, s_0) \in \textbf{int} \\ \mathsf{H.TypeOf} \ (\ eval(E, s, s_0)) & else \end{cases}$$

$$eval(\backslash\mathbf{elemtype(}E\mathbf{)}, s, s_0) = \\ \{ \ \mathtt{T} \quad if \ \mathsf{H.TypeOf} \ (\ eval(E, s, s_0)) = \ \mathtt{T[\ ]}$$

$$eval(\backslash\mathbf{TYPE}, s, s_0) = \ \mathtt{java.lang.Class}$$

The evaluation of stack expressions can be done only in intermediate state configurations $s =<\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}>$ *:*

$$eval(\mathbf{cntr}, s, s_0) = \mathrm{Cntr}$$

$$eval(\mathbf{st}(E), s, s_0) = \mathrm{St}(\ eval(E, s, s_0))$$

The evaluation of the following expressions can be done only in a final state $s =<\mathrm{H}, \mathrm{Reg}, Final >^{final}$*:*

$$\begin{array}{ll} eval(\backslash\mathbf{result}, s, s_0) = \mathrm{Res} & where \ s =<\mathrm{H}, \mathrm{Reg}, \mathrm{Res}>^{norm} \\ eval(\backslash\mathbf{EXC}, s, s_0) = \mathrm{Exc} & where \ s =<\mathrm{H}, \mathrm{Reg}, \mathrm{Exc}>^{exc} \end{array}$$

The relation $\models$ that we define next, gives a meaning to the formulas from our assertion language P.

Definition 4.2.2.2 (Interpretation of predicates) *The interpretation* $s \models P$ *of a predicate* P *in a state configuration* $s =<\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}>$ *w.r.t.*

an initial state $s_0 =< \mathrm{H}_0, 0, [\ ], \mathrm{Reg}, 0 >$ *is defined inductively as follows:*

$s, s_0 \vDash$ **true** *is true in any state s*

$s, s_0 \vDash$ **false** *is false in any state s*

$s, s_0 \vDash \neg\ P\ iff$ *not* $s, s_0 \vDash P$

$s, s_0 \vDash P_1 \wedge P_2\ iff\ s, s_0 \vDash P_1$ *and* $s, s_0 \vDash P_2$

$s, s_0 \vDash P_1 \vee P_2\ iff\ s, s_0 \vDash P_1$ *or* $s, s_0 \vDash P_2$

$s, s_0 \vDash P_1 \Rightarrow P_2\ iff\ if\ s, s_0 \vDash P_1$ *then* $s, s_0 \vDash P_2$

$s, s_0 \vDash P_1 \iff P_2\ iff\ s, s_0 \vDash P_1\ if$ *and only* $if\ s, s_0 \vDash P_2$

$s, s_0 \vDash \forall x : T.P(x)\ iff\ forall$ *value* $\mathbf{v}\ of$ *type* $T\ s, s_0 \vDash P(\ \mathbf{v})$

$s, s_0 \vDash \exists x : T.P(x)\ iff$ *a value* $\mathbf{v}\ of$ *type* T *exists such that* $s, s_0 \vDash P(\ \mathbf{v})$

$$s, s_0 \vDash E_1\ \mathcal{R}\ E_2\ iff \quad \begin{array}{l} eval(E_1, s, s_0) \neq \bot \wedge \\ eval(E_2, s, s_0) \neq \bot \wedge \\ eval(E_1, s, s_0)\ rel(\mathcal{R})\ eval(E_2, s, s_0)\ is\ true \end{array}$$

$s, s_0 \vDash$ **instances**$(\texttt{ref})$, *where* $\texttt{ref} \in RefVal\ iff\ inList(\texttt{ref}, \mathrm{getLoc}(\mathrm{H}_0)\ )$

4.3 Representing bytecode programs as control flow graphs

This section will introduce a formalization of an unstructured program in terms of a control flow graph. The notion of a loop in a bytecode program will be also defined. Performing analysis on programs written in structured languages, is usually easier than performing the same analysis on unstructured programs. In particular, source loops in a method body correspond to a syntactic construction which is not the case for loops in methods on bytecode level. In order to discover a loop in a bytecode program we first need to define what is a bytecode program. Note that in the following, by a bytecode program we mean a method body.

Every method `m` has an array of bytecode instructions `m.body` which we already introduced in Section 2.3. The $k - th$ instruction in the bytecode array `m.body` is denoted with `m.body`$[k]$. We assume that the method body has exactly one entry point (an entry point instruction is the instruction at which an execution of a method starts) which is the first element in the method body `m.body`$[0]$. The array of bytecode instructions of a method `m` determine an oriented graph $G(V, \rightarrow)$ in which the vertices are the instructions of the method body, i.e.

$$V = \{ins \mid \exists k, 0 \leq k < \texttt{m.body}.length \wedge ins = \texttt{m.body}[k]\}$$

The following definition defines the set of edges in the control flow graph.

Definition 4.3.1 (Edge in control flow graph) *The set of edges → is a relation between the vertices elements*

$$\rightarrow : V * V$$

and is defined as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{(\mathtt{m}.\mathsf{body}[j], \mathtt{m}.\mathsf{body}[k]) \in \rightarrow} \\
\multicolumn{2}{l}{\Longleftrightarrow} \\
\mathtt{m}.\mathsf{body}[j] \neq & \text{return } \wedge (\\
\mathtt{m}.\mathsf{body}[j] = & \text{if\_cond } \ k \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{goto } \ k \ \vee \\
\mathtt{m}.\mathsf{body}[j] \neq & \text{goto } \ \wedge k = j + 1 \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{putfield} \wedge \textit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{getfield} \wedge \textit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{type\_astore} \wedge \textit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{type\_astore} \wedge \textit{findExcHandler}(\ \mathtt{ArrIndBndExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{type\_aload} \wedge \textit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{type\_aload} \wedge \textit{findExcHandler}(\ \mathtt{ArrIndBndExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{invoke } \mathtt{n} \wedge \textit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{invoke } \mathtt{n} \wedge \forall \mathtt{Exc}, \exists s, \mathtt{n}.\mathsf{exceptions}[s] = \mathtt{Exc} \wedge \\
& \qquad\quad \textit{findExcHandler}(\mathtt{Exc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
\mathtt{m}.\mathsf{body}[j] = & \text{athrow} \wedge \forall \mathtt{Exc}, \textit{findExcHandler}(\mathtt{Exc}, j, \mathtt{m}.\mathsf{excHndlS}) = k \ \vee \\
) &
\end{array}$$

From the Def. 4.3.1 follows that there is an edge between two vertices $\mathtt{m}.\mathsf{body}[j]$ and $\mathtt{m}.\mathsf{body}[k]$ if they may execute immediately one after another. We say that $\mathtt{m}.\mathsf{body}[j]$ is a predecessor of $\mathtt{m}.\mathsf{body}[k]$ and that $\mathtt{m}.\mathsf{body}[k]$ is a successor of $\mathtt{m}.\mathsf{body}[j]$. The definition states the return instruction does not have successors. If $\mathtt{m}.\mathsf{body}[j]$ is the jump instruction if_cond k then its successors are the instruction at index k in the method body $\mathtt{m}.\mathsf{body}[k]$ and the instruction and the instruction $\mathtt{m}.\mathsf{body}[j+1]$. From the definition, we also get that every instruction which potentially may throw an exception of type `Exc` has as successor the first instruction of the exception handler that may handle the exception type `Exc`. For instance, a successor of the instruction putfield is the exception handler entry point which can handle the `NullPntrExc` exception. The possible successors of the instruction athrow are the entry point of any exception handler in the method `m`. In the following, we will rather use the infix notation $\mathtt{m}.\mathsf{body}[j] \rightarrow \mathtt{m}.\mathsf{body}[k]$.

We assume that the control flow graph of every method is reducible, i.e. every loop has exactly one entry point. This actually is admissible as it is rarely the case that a compiler produce a bytecode with a non reducible control flow graph and the practice shows that even hand written code is usually reducible. However, there exist algorithms to transform a non reducible control flow graph into a reducible one. For more information on program control flow graphs, the curious reader may refer to [1]. The next definition identifies backedges in the reducible control flow graph (intuitively, the edge that goes from an instruction in a given loop in the control flow graph to the loop entry) with the special execution relation $\rightarrow^l$ as follows:

Definition 4.3.2 (Backedge Definition) *Assume we have the method* m *with body* m.body *which determine the control flow graph* $G(V, \rightarrow)$. *We assume also that the entry point of* G *is the vertice* m.body$[0]$. *In such a graph* G, *we say that* $loopEntry : \mathtt{instr}$ *is a loop entry instruction and* $f : \mathtt{instr}$ *is a loop end instruction of the same loop if the following conditions hold:*

- *for every execution path* P *from* m.body$[0]$ *to* $f : \mathtt{instr}$*:* $P = \mathtt{m.body}[0] \rightarrow^+ f : \mathtt{instr}$ *there exists a subpath which is a prefix of* P $subP = \mathtt{m.body}[0] \rightarrow^* loopEntry : \mathtt{instr}$ *such that* $f : \mathtt{instr} \notin \ subP$

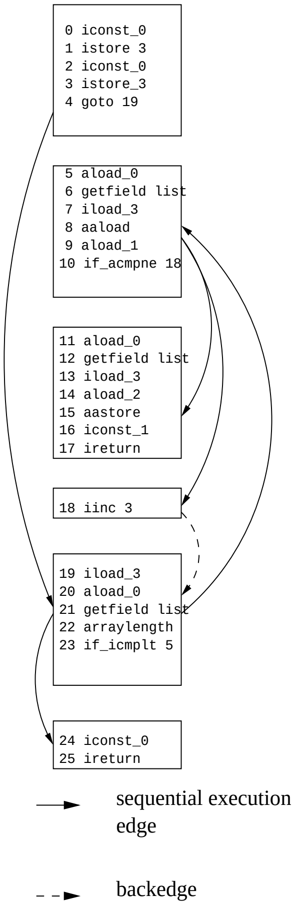
- *there is a path in which* $loopEntry : \mathtt{instr}$ *is executed immediately after the execution of* $f : \mathtt{instr}$ *(* $f : \mathtt{instr} \rightarrow loopEntry : \mathtt{instr}$*)*

We denote the execution relation between $f : \mathtt{instr}$ *and* $loopEntry : \mathtt{instr}$ *with* $f : \mathtt{instr} \rightarrow^l loopEntry : \mathtt{instr}$ *and we say that* $\rightarrow^l$ *is a backedge.*

We illustrate the upper definition with the control flow graph of the example from Fig. 3.1 in Fig. 4.1. In the figure, we rather show the execution relation between basic blocks which is a standard notion denoting a sequence of instructions which execute sequentially and where only the last one may be a jump and the first may be a target of a jump. The black edges represent a sequential execution relation, while dashed edges represent a backedge, i.e. the edge which stands for the execution relation between a final instruction (instruction at index 18) in the bytecode cycle and the entry instruction of the cycle (instruction at index 19).

4.4 Extending method declarations with specification

In the following, we propose an extension of the method formalization given in Section 2.3. The extension takes into account the method specification. The extended method structure is given below:



```
0 iconst_0
1 istore 3
2 iconst_0
3 istore_3
4 goto 19

5 aload_0
6 getfield list
7 iload_3
8 aaload
9 aload_1
10 if_acmpne 18

11 aload_0
12 getfield list
13 iload_3
14 aload_2
15 aastore
16 iconst_1
17 ireturn

18 iinc 3

19 iload_3
20 aload_0
21 getfield list
22 arraylength
23 if_icmplt 5

24 iconst_0
25 ireturn
```

Figure 4.1: The control flow graph of the source program from Fig.3.1

$$
\mathbf{Method} = \left\{ \begin{array}{ll}
\textsf{Name} & : \mathbf{MethodName} \\
\textsf{retType} & : \mathit{JType} \\
\textsf{args} & : (\mathit{name} * \mathit{JType})[] \\
\textsf{nArgs} & : \mathit{nat} \\
\textsf{body} & : \ \mathrm{I}[] \\
\textsf{excHndlS} & : \mathbf{ExcHandler}[] \\
\textsf{exceptions} & : \mathbf{Class}_{exc}[] \\
\textsf{pre} & : P \\
\textsf{modif} & : \mathit{Expr}[\ ] \\
\textsf{excPostSpec} & : \mathit{ExcType} \rightharpoonup P \\
\textsf{normalPost} & : P \\
\textsf{loopSpecS} & : \mathbf{LoopSpec}[\ ]
\end{array} \right\}
$$

Let's see the meaning of the new elements in the method data structure.

- $\mathtt{m}.\mathsf{pre}$ gives the precondition of the method, i.e. the predicate that must hold whenever $\mathtt{m}$ is called

- $\mathtt{m}.\mathsf{normalPost}$ is the postcondition of the method in case $\mathtt{m}$terminates normally

- $\mathtt{m}.\mathsf{modif}$ is also called the method frame condition. It is a list of expressions that the method may modify during its execution

- $\mathtt{m}.\mathsf{excPostSpec}$ is a total function from exception types to formulas which returns the predicate $\mathtt{m}.\mathsf{excPostSpec}(\mathtt{Exc})$ that must hold in the method's poststate if the method $\mathtt{m}$ terminates on an exception of type $\mathtt{Exc}$. Note that this function is constructed from the **exsures** clause of a method introduced in Chapter 3.1, section 3.3. For instance, if method $\mathtt{m}$ has an exsures clause:

  $$\textbf{exsures}\ (\ \mathtt{Exc})\ \textbf{reg}(1) = \textbf{null}$$

  then for every exception type $\mathtt{SExc}$ such that $\mathsf{subtype}\ (\mathtt{SExc}\ ,\mathtt{Exc})$ the function the result of the function $\mathtt{m}.\mathsf{excPostSpec}$ for $\mathtt{SExc}$ is $\mathtt{m}.\mathsf{excPostSpec}(\mathtt{SExc}) = \textbf{reg}(1) = \textbf{null}$. If for an exception $\mathtt{Exc}$ there is not specified exsures clause then the function $\mathsf{excPostSpec}$ returns the default exceptional postcondition predicate *false* , i.e. $\mathtt{m}.\mathsf{excPostSpec}(\mathtt{Exc}) = \mathit{false}$

- $\mathtt{m}.\mathsf{loopSpecS}$ is an array of **LoopSpec** data structures which give the specifcication information for a particular loop in the bytecode

The contents of a **LoopSpec** data structure is given hereafter:

$$\textbf{LoopSpec} = \left\{ \begin{array}{ll} \mathsf{pos} & : nat \\ \mathsf{invariant} & : P \\ \mathsf{modif} & : Expr[\,] \end{array} \right\}$$

For any method $\mathtt{m}$ for any k such that $0 \leq k < \mathtt{m}.\mathsf{loopSpecS}.length$

- the field $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}$ is a valid index in the body of $\mathtt{m}$: $0 \leq \mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos} < \mathtt{m}.\mathsf{body}.length$ and is a loop entry instruction in the sense of Def.4.3.2

- $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{invariant}$ is the predicate that must hold whenever the instruction $\mathtt{m}.\mathsf{body}[\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}]$ is reached in the execution of the method $\mathtt{m}$

- $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{modif}$ are the locations such that for any two states $state_1$, $state_2$ in which the instruction $\mathtt{m}.\mathsf{body}[\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}]$ executes agree on local variables and the heap modulo the locations that are in the list modif. We denote the equality between $state_1$, $state_2$ modulo the modifies locations like this $state_1 =^{\mathsf{modif}} state_2$

4.5 Weakest precondition calculus

In what follows, we assume that the bytecode has passed the bytecode verifier, thus it is well typed and well structured. Actually, our calculus is concerned only with functional properties of programs leaving the problem of code well structuredness and welltypedness to the bytecode verification techniques

The weakest precondition predicate transformer function which for any instruction of the Java sequential fragment determines the predicate that must hold in the prestate of the instruction has the following signature:

$$wp : (nat,\ \mathrm{I}) \longrightarrow \textbf{Method} \longrightarrow P$$

The function *wp* takes two arguments : the second argument is the method `m` to which the instruction belongs and the first argument is the instruction (for instance putfield) along with its position in `m`.

The function *wp* returns a predicate $wp(pos\ ins, \mathtt{m})$ such that if it holds in the prestate of the method `m` and if the `m` terminates normally then the normal postcondition `m`.normalPost holds when `m` terminates execution, otherwise if `m` terminates on an exception `Exc` the exceptional postcondition `m`.excPost(`Exc`) holds. Thus, the *wp* function takes into account both normal and exceptional program termination. Note however, that *wp* deals only with partial correctness, i.e. it does not guarantee program termination.

In order to define the *wp* function, we will need two other notions. The first one is a function which will determine the predicate between two instructions that are in execution relation as defined in Def. 4.3.1. Note that this is not necessary for structured programs. However, for unstructured programs with loops annotated with invariants and frame conditions, this is a necessary step. The definition of the intermediate predicate is given in the next subsection 4.5.1. We will also see how the weakest precondition is defined in presence of exceptions. This is done in subsection **??**.

4.5.1 Intermediate predicates

In this subsection, we define a function *inter* which for two instructions that may execute one after another in a control graph of a method `m` determines the predicate $inter(j, k, \mathtt{m})$ which must hold in between them. The function has the signature:

$$inter : nat \longrightarrow nat \longrightarrow \textbf{Method} \longrightarrow P$$

The predicate $inter(j, k, \mathtt{m})$ will be used for determining the weakest predicate that must hold in the prestate of the instruction j : `instr` if the execution path after passes through the instruction k : `instr`.

This predicate depends on the execution relation between the two instructions j : `instr` and k : `instr` as the next definition shows.

Definition 4.5.1 (Intermediate predicate between two instructions) *Assume that* $j : \texttt{instr} \rightarrow k : \texttt{instr}$*. The predicate* $inter(j, k, \texttt{m})$ *must hold after the execution of* $j : \texttt{instr}$ *and before the execution of* $k : \texttt{instr}$ *and is defined as follows:*

- *if* $k : \texttt{instr}$ *is a loop entry instruction,* $j : \texttt{instr} \rightarrow^l k : \texttt{instr}$ *and* $\texttt{m}.\mathsf{loopSpecS}[s].\mathsf{pos} = k$ *then the corresponding loop invariant must hold:*

$$inter(j, k, \texttt{m}) \equiv \texttt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant}$$

- *else if* $k : \texttt{instr}$ *is a loop entry and* $\texttt{m}.\mathsf{loopSpecS}[s].\mathsf{pos} = k$ *then the corresponding loop invariant* $\texttt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant}$ *must hold before* $k : \texttt{instr}$ *is executed, i.e. after the execution of* $j : \texttt{instr}$*. We also require that* $\texttt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant}$ *implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations* $\texttt{m}.\mathsf{loopSpecS}[s].\mathsf{modif}$ *that may be modified in the loop body:*

$$\begin{array}{l} inter(j, k, \texttt{m}) \equiv \\ \texttt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant} \wedge \\ \forall i, i = 1..\texttt{m}.\mathsf{loopSpecS}[s].\mathsf{modif}.length, \\ \forall \texttt{m}.\mathsf{loopSpecS}[s].\mathsf{modif}[i], (\\ \qquad\qquad \texttt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant} \Rightarrow \\ \qquad\qquad\qquad\qquad wp(k\ \ , \texttt{m})) \end{array}$$

- *else*

$$inter(j, k, \texttt{m}) \equiv wp(k\ \ , \texttt{m})$$

4.5.2 Weakest precondition in the presence of exceptions

Our weakest precondition calculus deals with exceptional termination and thus, we need some mechanism for providing the exceptional postcondition predicate of an instruction when it throws an exception. For this, we define the function getExcPostIns with signature :

$$\mathsf{getExcPostIns} : int \longrightarrow ExcType \longrightarrow P$$

The function $\texttt{m}.\mathsf{getExcPostIns}$ takes as arguments an index i in the array of instructions of method $\texttt{m}$ and an exception type $\texttt{Exc}$ and returns the predicate $\texttt{m}.\mathsf{getExcPostIns}(i,\ \texttt{Exc}\ )$ that must hold after the instruction at index i throws an exception. We give a formal definition hereafter.

Definition 4.5.2.1 (Postcondition in case of a thrown exception)

$$\begin{array}{l} \texttt{m}.\mathsf{getExcPostIns}(i,\ \texttt{Exc}) = \\ \left\{ \begin{array}{ll} inter(i, \mathsf{handlerPc}, \texttt{m}) & if\ findExcHandler(\ \texttt{Exc}, i, \texttt{m}.\mathsf{excHndlS}) = \mathsf{handlerPc} \\ \texttt{m}.\mathsf{excPostSpec}(\ \texttt{Exc}) & findExcHandler(\ \texttt{Exc}, i, \texttt{m}.\mathsf{excHndlS}) = \bot \end{array} \right. \end{array}$$

Next, we introduce an auxiliary function which will be used in the definition of the *wp* function for instructions that may throw runtime exceptions. Thus, for every method m we define the auxiliary function m.excPost with signature:

$$\mathtt{m}.\mathsf{excPost} : int \longrightarrow ExcType \longrightarrow P$$

$\mathtt{m}.\mathsf{excPost}(\ i,\ \mathtt{Exc})$ returns the predicate that must hold in the preststate of the instruction at index i which may throw a runtime exception of type Exc. Note that the function m.excPost does not deal with programmatic exceptions thrown by the instruction athrow , neither exception caused by a method invokation (execution of instruction invoke) as those instructions may potentially throw exceptions which are not runtime exceptions. Those two cases are handled in a different way as we shall see later in the definition of the *wp* function in Section 4.5.

The function application $\mathtt{m}.\mathsf{excPost}(\ i,\ \mathtt{Exc})$ is defined as follows:

Definition 4.5.2.2 (Auxuliary function for instructions throwing runtime exceptions)

$$
\begin{array}{l}
i:\mathtt{instr} \neq \ \text{athrow} \wedge i:\mathtt{instr} \neq \ \text{invoke} \Rightarrow \\
\mathtt{m}.\mathsf{excPost}(\ i,\ \mathtt{Exc}) = \\
\forall \mathtt{ref}, \\
\quad \neg\ \mathbf{instances}(\mathtt{ref}) \wedge \\
\quad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\qquad \mathtt{m}.\mathsf{getExcPostIns}(\ i,\ \mathtt{Exc}) \\
\qquad\ [\mathbf{cntr} \leftarrow 0] \\
\qquad\ [\mathbf{st}(0) \leftarrow \mathtt{ref}] \\
\qquad\ [f \leftarrow f[\oplus \mathtt{ref} \rightarrow defVal(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\ \mathsf{subtype}\ (f.\mathsf{declaredIn},\ \mathtt{Exc})} \\
\qquad\ [\backslash\mathbf{typeof(}\mathtt{ref}\mathbf{)} \leftarrow\ \mathtt{Exc}]
\end{array}
$$

The function m.excPost will return a predicate which states that for every newly created exception reference the predicate returned by the function getExcPostIns must hold.

4.5.3 Rules for single instruction

In the following, we give the definition of the weakest precondition function for every instruction.

- Control transfer instructions

  1. unconditional jumps

  $$wp(i\ \ \text{goto}\ \ n, \mathtt{m}) =\ inter(i, n, \mathtt{m})$$

  The rule says that an unconditional jump does not modify the program state and thus, the postcondition and the precondition of this instruction are the same

2. conditional jumps

$$\begin{array}{l} wp(i \;\; \text{if\_cond} \;\; n, \mathtt{m}) = \\ \quad \mathtt{cond}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr} - 1)) \Rightarrow \\ \qquad \mathit{inter}(i, n, \mathtt{m})[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ \wedge \\ \mathit{not}(\;\; \mathtt{cond})(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr} - 1))) \Rightarrow \\ \qquad \mathit{inter}(i, i+1, \mathtt{m})[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \end{array}$$

In case of a conditional jump, the weakest precondition depends on if the condition of the jump is satisfied by the two stack top elements. If the condition of the instruction evaluates to true then the predicate between the current instruction and the instruction at index n must hold where the stack counter is decremented with 2 $\mathit{inter}(i, n, \mathtt{m})[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2]$ If the condition evaluates to false then the predicate between the current instruction and its next instruction holds where once again the stack counter is decremented with two $\mathit{inter}(i, i+1, \mathtt{m})[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2]$.

3. return

$$wp(\mathtt{m} \;\; \text{return} \;, i) = \mathtt{m}.\mathsf{normalPost}[\backslash\mathbf{result} \leftarrow \mathbf{st}(\mathbf{cntr})]$$

As the instruction return marks the end of the execution path, we require that its postcondition is the normal method postcondition normalPost. Thus, the weakest precondition of the instruction is normalPost where the specification variable **result** is substituted with the stack top element.

- load and store instructions

1. load a local variable on the operand stack

$$\begin{array}{l} wp(i \;\; \text{load} \;\; \mathtt{j}, \mathtt{m}) = \\ \mathit{inter}(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(j)] \end{array} \end{array}$$

The weakest precondition of the instruction then is the predicate that must hold between the current instruction and its successor, but where the stack counter is incremented and the stack top is substituted with $\mathbf{reg}(j)$. For instance, if we have that the predicate $\mathit{inter}(i, i+1, \mathtt{m})$ is equal to $\mathbf{st}(\mathit{counter}) == 3$ then we get that the precondition of instruction is $\mathbf{reg}(j) == 3$:

$$\begin{array}{l} \{\mathbf{reg}(j) == 3\} \\ i : \text{ load } \mathrm{j} \\ \{\mathbf{st}(\mathbf{cntr}) == 3\} \\ i + 1 : \ldots \end{array}$$

2. store the stack top element in a local variable

$$\begin{array}{l} wp(i \ \ \text{store} \ \ \mathtt{j}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 1] \\ [\mathbf{reg}(j) \leftarrow \mathbf{st}(\mathbf{cntr})] \end{array} \end{array}$$

   Contrary to the previous instruction, the instruction store j will take the stack top element and will store its contents in the local variable $\mathbf{reg}(j)$.

3. push an integer constant on the operand stack

$$\begin{array}{l} wp(i \ \ \text{push} \ \ \mathtt{j}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \ \mathtt{j} \ ] \end{array} \end{array}$$

   The predicate that holds after the instruction holds in the prestate of the instruction but where the stack counter **cntr** is incremented and the constant j is stored in the stack top element

4. incrementing a local variable

$$\begin{array}{l} wp(\mathtt{m} \ \ \text{iinc} \ \ \mathtt{j}, i) = \\ inter(i, i+1, \mathtt{m})[\mathbf{reg}(j) \leftarrow \mathbf{reg}(j) + 1] \end{array}$$

- arithmetic instructions

1. instructions that cannot cause exception throwing ($\mathtt{arithOp} =$ add , sub , mult , and , or , xor , ishr , ishl ,)

$$\begin{array}{l} wp(i \ \ \text{arith\_op}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 1] \\ [\mathbf{st}(\mathbf{cntr} - 1) \leftarrow \mathbf{st}(\mathbf{cntr})\mathtt{op}\ \mathbf{st}(\mathbf{cntr} - 1)] \end{array} \end{array}$$

   We illustrate this rule with an example. Let us have the arithmetic instruction add at index i such that the predicate $inter(i, i+1, \mathtt{m}) \equiv \mathbf{st}(\mathbf{cntr}) \geq 0$. In this case, applying the rule we get that the weakest precondition is $\mathbf{st}(\mathbf{cntr} - 1) + \mathbf{st}(\mathbf{cntr}) \geq 0$:

$$\begin{array}{l} \{\mathbf{st}(\mathbf{cntr} - 1) + \mathbf{st}(\mathbf{cntr}) \geq 0\} \\ i : \ \text{add} \\ \{\mathbf{st}(\mathbf{cntr}) \geq 0\} \end{array}$$

2. instructions that may throw exceptions ($\mathtt{arithOp} =$ rem , div)

$$
\begin{array}{l}
wp(i \;\; \text{arithOp} \;, \mathtt{m}) = \\
\;\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad inter(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 1] \\ [\mathbf{st}(\mathbf{cntr} - 1) \leftarrow \mathbf{st}(\mathbf{cntr}) \; \mathtt{op} \; \mathbf{st}(\mathbf{cntr} - 1)] \end{array} \\
\\
\;\wedge \\
\\
\;\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPost}(i, \; \mathtt{NullPntrExc})
\end{array}
$$

- object creation and manipulation

  1. create a new object

$$
\begin{array}{l}
wp(i \;\; \text{new} \;\; C, \mathtt{m}) = \\
\forall \mathtt{ref}, \\
\quad not \; \mathbf{instances}(\mathtt{ref}) \wedge \\
\quad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\qquad\quad inter(i, i+1, \mathtt{m}) \\
\qquad\quad [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\
\qquad\quad [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathtt{ref}] \\
\qquad\quad [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}\;(f.\mathsf{declaredIn}, C)} \\
\qquad\quad [\backslash \mathbf{typeof}(\mathtt{ref}) \leftarrow C]
\end{array}
$$

  The postcondition of the instruction new is the intermediate predicate $inter(i, i+1, \mathtt{m})$. The weakest precondition of the instruction says that for any reference `ref` if `ref` was not instantiated in the initial state of the execution of `m` then the precondition is the same predicate but in which the stack counter is incremented and `ref` is pushed on the stack top where the fields for the `ref` are initialized with their default values

  2. array creation

$$
\begin{array}{l}
wp(i \;\; \text{newarray} \; \mathtt{T}, \mathtt{m}) = \\
\forall \mathtt{ref}, \\
\quad not \; \mathbf{instances}(\mathtt{ref}) \wedge \\
\quad \mathtt{ref} \neq \mathbf{null} \wedge \\
\quad \mathbf{st}(\mathbf{cntr}) \geq 0 \Rightarrow \\
\qquad\quad inter(i, i+1, \mathtt{m}) \\
\qquad\quad [\mathbf{st}(\mathbf{cntr}) \leftarrow \mathtt{ref}] \\
\qquad\quad [\text{arrAccess} \leftarrow \text{arrAccess}[\oplus(\mathtt{ref}, j) \rightarrow \mathsf{defVal}(\mathtt{T})]]_{\forall j, 0 \leq j < \mathbf{st}(\mathbf{cntr})} \\
\qquad\quad [\text{arrLength} \leftarrow \text{arrLength}[\oplus \mathtt{ref} \rightarrow \mathbf{st}(\mathbf{cntr})]] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) < 0 \Rightarrow \mathtt{m}.\mathsf{excPost}(i, \; \mathtt{NegArrSizeExc})
\end{array}
$$

  Here, the rule for array creation is similar to the rule for object creation. However, creation of an array might terminate exceptionally in case the length of the array stored in the stack top element $\mathbf{st}(\mathbf{cntr}$ $)$ is smaller than 0. In this case, function $\mathtt{m}.\mathsf{excPost}$ will search for the corresponding postcondition of the instruction at position i and the exception `NegArrSizeExc`

3. field access

$$\begin{array}{l} wp(i \ \text{ getfield } f, \mathtt{m}) = \\ \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\ \quad\quad inter(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr}) \leftarrow f(\mathbf{st}(\mathbf{cntr}))] \\ \wedge \\ \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPost}(i, \ \mathtt{NullPntrExc}) \end{array}$$

The instruction for accessing a field value takes as postcondition the predicate that must hold between it and its next instruction $inter(i, i+1, \mathtt{m})$. This instruction may terminate normally or on an exception. In case the stack top element is not **null**, the precondition of getfield is its postcondition where the stack top element is substituted by the field access expression $f(\mathbf{st}(\mathbf{cntr})$. If the stack top element is **null**, then the instruction will terminate on a `NullPntrExc` exception. In this case the precondition of the instruction is the predicate returned by the function $\mathtt{m}.\mathsf{excPost}$ for position i in the bytecode and exception `NullPntrExc`

4. field update

$$\begin{array}{l} wp(i \ \text{ putfield } f, \mathtt{m}) = \\ \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\ \quad\quad inter(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]] \end{array} \\ \wedge \\ \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPost}(i, \ \mathtt{NullPntrExc}) \end{array}$$

This instruction also may terminate normally or exceptionally. The termination depends on the value of the stack top element in the prestate of the instruction. If the top stack element is not **null** then in the precondition of the instruction $inter(i, i+1, \mathtt{m})$ must hold where the stack counter is decremented with two elements and the f object is substituted with an updated version $f[\oplus \mathbf{st}(\mathbf{cntr} - 2) \rightarrow \mathbf{st}(\mathbf{cntr} - 1)]$

For example, let us have the instruction putfield f in method `m`. Its normal postcondition is $inter(i, i+1, \mathtt{m}) \equiv f(\mathbf{reg}(1)) \neq \mathbf{null}$. Assume that `m` does not have exception handler for `NullPntrExc` exception for the region in which the putfield instruction. Let the exceptional postcondition of `m` for `NullPntrExc` be *false* , i.e. $\mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{NullPntrExc}) = \ false$ If all these conditions hold, the function *wp* will return for the putfield instruction the following formula :

$$\begin{array}{l}
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad (f(\mathbf{reg}(1)) \neq \mathbf{null}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]] \end{array} \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathit{false}
\end{array}$$

After applying the substitution following the rules described in Section 4.2.1, we obtain that the precondition is

$$\begin{array}{l}
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})](\mathbf{reg}(1)) \neq \mathbf{null} \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathit{false}
\end{array}$$

Finally, we give the instruction putfield its postcondition and the respective weakest precondition:

$$\begin{array}{l}
\{ \begin{array}{l} \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\ f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})](\mathbf{reg}(1)) \neq \mathbf{null}\} \\ \wedge \\ \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathit{false} \end{array} \} \\
i : \ \text{putfield } f \\
\{f(\mathbf{reg}(1)) \neq \mathbf{null}\} \\
i + 1 : \ldots
\end{array}$$

5. access the length of an array

$$\begin{array}{l}
\mathit{wp}(i \ \ \text{arraylength}, \mathtt{m}) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad \mathit{inter}(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr}) \leftarrow \text{arrLength}(\mathbf{st}(\mathbf{cntr}))] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPost}(i, \ \mathtt{NullPntrExc})
\end{array}$$

The semantics of arraylength is that it takes the stack top element which must be an array reference and puts on the operand stack the length of the array referenced by this reference. This instruction may terminate either normally or exceptionally. The termination depends on if the stack top element is **null** or not. In case $\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null}$ the predicate $\mathit{inter}(i, i+1, \mathtt{m})$ must hold where the stack top element is substituted with its length. The case when a `NullPntrExc`is thrown is similar to the previous cases with exceptional termination

6. checkcast

$$\begin{array}{l}
\mathit{wp}(i \ \ \text{checkcast } C, \mathtt{m}) = \\
\backslash\mathbf{typeof(st}(\mathbf{cntr})) <: C \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\qquad \mathit{inter}(i, i+1, \mathtt{m}) \\
\wedge \\
\mathit{not}(\backslash\mathbf{typeof(st}(\mathbf{cntr})) <: C) \Rightarrow \mathtt{m}.\mathsf{excPost}(i, \mathtt{CastExc})
\end{array}$$

The instruction checks if the stack top element can be cast to the class C. Two termination of the instruction are possible. If the stack top element $\textbf{st}(\textbf{cntr}\ )$ is of type which is a subtype of class C or is **null** then the predicate $inter(i, i+1, \mathtt{m})$ holds in the prestate. Otherwise, if $\textbf{st}(\textbf{cntr}\ )$ is not of type which is a subtype of class C, the instruction terminates on `CastExc` and the predicate returned by `m`.excPost for the position i and exception `CastExc` must hold

7. instanceof

$$\begin{array}{l} wp(i \;\; \text{instanceof } C, \mathtt{m}) = \\ \backslash\textbf{typeof(st(cntr))} <: C \Rightarrow \\ \qquad inter(i, i+1, \mathtt{m})[\textbf{st}(\textbf{cntr}) \leftarrow 1] \\ \wedge \\ not(\backslash\textbf{typeof(st(cntr))} <: C) \vee \textbf{st}(\textbf{cntr}) = \textbf{null} \Rightarrow \\ \qquad inter(i, i+1, \mathtt{m})[\textbf{st}(\textbf{cntr}) \leftarrow 0] \end{array}$$

This instruction, depending on if the stack top element can be cast to the class type C pushes on the stack top either 0 or 1. Thus, the rule is almost the same as the previous instruction checkcast .

- method invocation (only the case for non void instance method is given).

$$\begin{array}{l} wp(i \;\; \text{invoke } \mathtt{n}\ , \mathtt{m}) = \\ \mathtt{n}\ .\mathsf{pre}[\textbf{reg}(s) \leftarrow \textbf{st}(\textbf{cntr} + s - \mathtt{m}.\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \\ \\ \wedge \\ \\ \forall mod, (mod \in \mathtt{n}\ .\mathsf{modif}), \forall freshVar(\\ \qquad \mathtt{n}\ .\mathsf{normalPost} \begin{array}{l} [\backslash\textbf{result} \leftarrow freshVar] \\ [\textbf{reg}(s) \leftarrow \textbf{st}(\textbf{cntr} + s - \mathtt{n}\ ).\mathsf{nArgs}]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \end{array} \Rightarrow \\ \\ \qquad\qquad inter(i, i+1, \mathtt{m}) \begin{array}{l} [\textbf{cntr} \leftarrow \textbf{cntr} - \mathtt{n}\ .\mathsf{nArgs}] \\ [\textbf{st}(\textbf{cntr} - \mathtt{n}\ .\mathsf{nArgs}) \leftarrow freshVar] \end{array}) \\ \\ \wedge_{j=0}^{\mathtt{n}\ .\mathsf{exceptions}.length-1} \\ \\ \forall mod, (mod \in \mathtt{n}\ .\mathsf{modif}), \\ \quad (findExcHandler(\mathtt{n}\ .\mathsf{exceptions}[j], i, \mathtt{m}.\mathsf{excHndlS}) = \bot \Rightarrow \\ \quad \forall \textbf{bv}_i(\\ \qquad \mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j])[\backslash\textbf{EXC} \leftarrow \textbf{bv}_i] \Rightarrow \\ \qquad\quad \mathtt{m}.\mathsf{getExcPostIns}(i, \mathtt{m}.\mathsf{exceptions}[j])[\backslash\textbf{EXC} \leftarrow \textbf{bv}_i])) \\ \quad \wedge \\ \quad (findExcHandler(\mathtt{m}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j]), i, \mathtt{m}.\mathsf{excHndlS}) = k \Rightarrow \\ \quad \forall \textbf{bv}_i(\\ \qquad \mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j])[\backslash\textbf{EXC} \leftarrow \textbf{bv}_i] \Rightarrow \\ \qquad\quad \mathtt{m}.\mathsf{getExcPostIns} \begin{array}{l} [\textbf{cntr} \leftarrow 0] \\ [\textbf{st}(0) \leftarrow \textbf{bv}_i] \end{array})) \end{array}$$

Let us look in detail what is the meaning of the weakest precondition for method invokation. Because we are following a contract based approach the caller, i.e. the current method m must establish several facts. First, we require that the precondition n.pre of the invoked method n holds where the formal parameters are correctly initialized with the first n.nArgs elements from the operand stack.

Second, we get a logical statement which guarantees the correctness of the method invokation in case of normal termination. On the other hand, its postcondition n.normalPost is assumed to hold and thus, we want to establish that under the assumption that m.normalPost holds with **result** substituted with a fresh bound variable $\mathbf{bv}_i$ and correctly initialized formal parameters is true we want to establish that the predicate *inter*(i, i+1 , m) holds . This implication is quantified over the locations n.modif that a method may modify and the variable $\mathbf{bv}_i$ which stands for the result that the invoked method n returns.

The third part of the rule deals with the exceptional termination of the method invokation. In this case, if the invoked method n terminates on any exception which belongs to the array of exceptions n.exceptions that n may throw. Two cases are considered - either the thrown exception can be handled by m or not. If the thrown exception Exc can not be handled by the method m (i.e. $\mathit{findExcHandler}(\mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j]), i, \mathtt{m}.\mathsf{excHndlS}) = \bot$) then if the exceptional postcondition predicate n .excPostSpec(Exc) of n holds then m.excPostSpec(Exc) for any value of the thrown exception object. In case the thrown exceptionExc is handled by m, i.e. $\mathit{findExcHandler}(\mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j]), i, \mathtt{m}.\mathsf{excHndlS}) = k$ then if the exceptional postcondition n .excPostSpec(Exc) of n holds then the intermediate predicate $\mathit{inter}(i, k, \mathtt{m})$ that must hold after i : instr and before k : instr must hold once again for any value of thrown exception.

- throw exception instruction

$$
\begin{array}{l}
wp(i\ \ \text{athrow}\ , \mathtt{m}) = \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{getExcPostIns}(i,\ \mathtt{NullPntrExc}) \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\ \forall \mathtt{Exc}, \\
\ \backslash\mathbf{typeof(st(cntr))} <: \mathtt{Exc} \Rightarrow \\
\qquad \mathtt{m}.\mathsf{getExcPostIns}(i, \mathtt{Exc})[\backslash\mathbf{EXC} \leftarrow \mathbf{st}(\mathbf{cntr})]
\end{array}
$$

The thrown object is on the top of the stack **st**(**cntr**). If the stack top object **st**(**cntr**) is **null**, then the instruction athrow will terminate on an exception NullPntrExc where the predicate returned by the function m.excPost must hold. The case when the thrown object is not **null** should consider all the possible exceptions that might be thrown by the current instruction. This is because we do not know the type of the thrown object

```
//@ ensures \result == i*i;
public int square( int i ) {
  int sqr  = 0;
  if ( i < 0) {
    i = -i;
  }
  //@ loop_modifies s, sqr;
  //@ loop_invariant (0 <= s) && (s <= i) && sqr == s*s ;
  for (int s = 0 ; s < i; s++ ) {
    sqr = sqr + 2*s + 1;
  }
  return sqr;
}
```

Figure 4.2: Java method which calculates the square of its input

Figure 4.3: bytecode of method square

which is on the stack top. The part of the *wp* when the thrown object on the stack top **st**(**cntr**) is not **null** considers all the possible types of the exception thrown. In any of

Supposing the execution of a method always terminates, the verification condition for a method m with a precondition m.pre is defined in the following way:

$$\mathtt{m}.\mathsf{pre} \Rightarrow wp(0\ \ \mathrm{m}\ .\mathsf{body}[0], \mathtt{m})$$

4.6 Example

In the following, we will consider an example of the application of the verification procedure with *wp*. Consider Fig. 4.2, which gives an example of a Java method which calculates the square of its input which is stated in its postcondition. The calculation of the square of the parameter i is done with an iteration which sums all the impair numbers 2∗s + 1, 0<=s <=i in the local variable sqr. The invariant states that whenever the loop entry is reached the variable sqr will contain the square of the local variable s and that 0 <=s <=i. In Fig.4.3, we show the bytecode of method square.

We next show the weakest preconditions of the basic block containing the return instruction as well the block containing the loop end instruction.

Chapter 5

Correctness of the verification condition generator

In the previous chapter 4, we defined a verification condition generator for a Java bytecode like language. We used a weakest precondition to build the verification conditions. In this section, we will argue formally that the proposed verification condition generator is correct, or in other words that it is sufficient to prove the verification conditions generated over a bytecode program and its specification for establishing that the program respects the specification.

In particular, we will prove the correctness of our methodology w.r.t. the operational semantics of our bytecode language given in chapter 2.7. The way in which the proof is done is standard. Note that the formalization of the operational semantics in terms of relation on states serves us to give a model for our assertion language.

We now proceed with the proof of the partial correctness of the weakest precondition calculus, i.e. we assume that programs always terminate. Note also that in the following we do not consider recursive methods. The first section 5.1 introduces several properties concerning expression evaluation and interpretation of predicates in a particular state. Those properties will play role in the correctness proof of the verification condition generator in section 5.2. Section 5.2 starts with a formal definition for method correctness. Then, we establish the correctness of a single instruction (lemma 5.2.1). The next step of the proof is to establish that if all the steps in an execution path establish the intermediate predicates then the execution can either proceed by establishing the next weakest precondition predicate or will terminate in a state which respects the adequate postcondition.

5.1 Substitution properties

The following lemmas estasblish that substitution over state configurations or expressions / formulas result in the same evaluation

Lemma 5.1.1 (Update a local variable) *For any expressions $Expr_1, Expr_2$ if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}[\oplus i \rightarrow \ eval(Expr_2, s_1, s_0)], \mathrm{Pc} >$ then the following holds:*

1. $eval(Expr_1[\mathbf{reg}(i) \leftarrow Expr_2], s_1, s_0) = \ eval(Expr_1, s_2, s_0)$
2. $s_1, s_0 \vDash \psi[\mathbf{reg}(i) \leftarrow Expr_2] \iff s_2, s_0 \vDash \psi$

Proof : by structural induction on the structure of $Expr_1$

1. we look at the first part of the lemma concerning expression evaluation

   - $Expr_1 = \mathbf{reg}(i)$

     $(left)$ $\mathbf{reg}(i)[\mathbf{reg}(i) \leftarrow Expr_2] = Expr_2$
     $\Rightarrow$
     (1) $eval(\mathbf{reg}(i)[\mathbf{reg}(i) \leftarrow Expr_2], s_1, s_0) = \ eval(Expr_2, s_1, s_0)$

     $(right)$ $eval(\mathbf{reg}(i), s_2, s_0) =$
     { *by Def.4.2.2.1 of the evaluation for local variables* }
     (2) $= \ eval(Expr_2, s_1, s_0)$
     { *from (1) and (2) we get that the lemma holds in this case* }

   - $Expr_1 = Expr_3.f$

     $Expr_3.f[\mathbf{reg}(i) \leftarrow Expr_2] =$
     { *by definition of the substitution* }
     $= Expr_3[\mathbf{reg}(i) \leftarrow Expr_2].f$
     { *by induction hypothesis* }
     (1) $eval(Expr_3[\mathbf{reg}(i) \leftarrow Expr_2], s_1, s_0) = \ eval(Expr_3, s_2, s_0)$

     { *by Def.4.2.2.1 of the evaluation for field access expressions* }
     $(left)$ $eval(Expr_3.f[\mathbf{reg}(i) \leftarrow Expr_2], s_1, s_0) =$
     $= \mathrm{H}(f)(\ eval(Expr_3[\mathbf{reg}(i) \leftarrow Expr_2], s_1, s_0))$

     $(right)$ $eval(Expr_3.f, s_2, s_0) =$
     $= \mathrm{H}(f)(\ eval(Expr_3, s_2, s_0))$

     { *from (1),(left) and (right)*
     we get that the lemma holds in this case }

   - the rest of the cases proceed in a similar way by appluing the induction hypothesis

2. second case of the lemma

   - $\psi = E' \mathcal{R}\ E'$

     $\{$ *from the first part of the lemma we get* $\}$
     (1) $\mathit{eval}(\mathit{Expr}'[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2], s_1, s_0) = \mathit{eval}(\mathit{Expr}', s_2, s_0)$
     (2) $\mathit{eval}(\mathit{Expr}''[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2], s_1, s_0) = \mathit{eval}(\mathit{Expr}'', s_2, s_0)$

     $s_1, s_0 \vDash \psi[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2]$
     $\{$ *definition of substitution* $\}$
     (3) $\equiv$
     $s_1, s_0 \vDash \mathit{Expr}'[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2]\ \mathcal{R}\ \mathit{Expr}''[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2]$
     $\{$ *by Def.4.2.2.2 we get* $\}$
     $\Longleftrightarrow$
     $\mathit{eval}(\mathit{Expr}'[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2], s_1, s_0)\ \mathit{rel}(\mathcal{R})\ \mathit{eval}(\mathit{Expr}''[\mathbf{reg}(i) \leftarrow \mathit{Expr}_2], s_1, s_0)$ *is true*
     $\{$ *from (1) , (2) and (3)* $\}$
     $\Longleftrightarrow$
     $\mathit{eval}(\mathit{Expr}', s_2, s_0)\ \mathit{rel}(\mathcal{R})\ \mathit{eval}(\mathit{Expr}'', s_2, s_0)$
     $\equiv$
     $s_2, s_0 \vDash \psi$

   - the rest of the cases are by structural induction

Lemma 5.1.2 (Update of the heap) *For any expressions* $\mathit{Expr}_1, \mathit{Expr}_2, \mathit{Expr}_3$ *and any field* f *if we have that the states* s_1 *and* s_2 *are such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and*
$s_2 =< \mathrm{H}[\oplus f \rightarrow f[\oplus\ \mathit{eval}(\mathit{Expr}_2, s_1, s_0) \rightarrow\ \mathit{eval}(\mathit{Expr}_3, s_1, s_0)]], \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$
the following holds

1. $\mathit{eval}(\mathit{Expr}_1[f \leftarrow f[\oplus \mathit{Expr}_2 \rightarrow \mathit{Expr}_3]], s_1, s_0) = \mathit{eval}(\mathit{Expr}_1, s_2, s_0)$
2. $s_1, s_0 \vDash \psi[f \leftarrow f[\oplus \mathit{Expr}_2 \rightarrow \mathit{Expr}_3]] \iff s_2, s_0 \vDash \psi$

Lemma 5.1.3 (Update of the heap with a newly allocated object) *For any expressions* Expr_1 *if we have that the states* s_1 *and* s_2 *are such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and* $s_2 =< \mathrm{H}', \mathrm{Cntr}, \mathrm{St}[\oplus \mathrm{Cntr} \rightarrow\ \mathit{eval}(\mathtt{ref}, s_1, s_0)], \mathrm{Reg}, \mathrm{Pc} >$ *where* $\mathrm{newRef}(\mathrm{H}, C) = (\mathrm{H}', \mathtt{ref})$ *the following holds*

1.

$$\mathit{eval}(\mathit{Expr}_1 \begin{array}{l} [\mathbf{st}(\mathbf{cntr}) \leftarrow \mathtt{ref}] \\ [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathit{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}, \mathsf{subtype}\ (f.\mathsf{declaredIn}, C)} \end{array}, s_1, s_0)$$
$$=$$
$$\mathit{eval}(\mathit{Expr}_1, s_2, s_0)$$

2.

$$s_1, s_0 \vDash \psi \begin{array}{l} [\mathbf{st}(\mathbf{cntr}) \leftarrow \mathtt{ref}] \\ [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathit{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}, \mathsf{subtype}\ (f.\mathsf{declaredIn}, C)} \end{array}$$
$$\Longleftrightarrow$$
$$s_2, s_0 \vDash \psi$$

Lemma 5.1.4 (Update the stack) *For any expressions* $Expr_1, Expr_2, Expr_3$ *if we have that the states* s_1 *and* s_2 *are such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and* $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}[\oplus\ eval(Expr_2, s_1, s_0) \rightarrow\ eval(Expr_3, s_1, s_0)], \mathrm{Reg}, \mathrm{Pc} >$ *then the following holds:*

1. $eval(Expr_1[\mathbf{st}(Expr_2) \leftarrow Expr_3], s_1, s_0) =\ eval(Expr_1, s_2, s_0)$

2. $s_1, s_0 \models \psi[\mathbf{st}(Expr_2) \leftarrow Expr_3] \iff s_2, s_0 \models \psi$

Lemma 5.1.5 (Update the stack counter) *For any expressions* $Expr_1, Expr_2$ *if we have that the states* s_1 *and* s_2 *are such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and* $s_2 =< \mathrm{H},\ eval(Expr_2, s_1, s_0), \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *then the following holds:*

1. $eval(Expr_1[\mathbf{cntr} \leftarrow Expr_2], s_1, s_0) =\ eval(Expr_1, s_2, s_0)$

2. $s_1, s_0 \models \psi[\mathbf{cntr} \leftarrow Expr_2] \iff s_2, s_0 \models \psi$

Lemma 5.1.6 (Return value property) *For any expression* $Expr_1$ *and* $Expr_2$*, for any two states* s_1 *and* s_2 *such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and* $s_2 =< \mathrm{H}, \mathrm{Reg},\ eval(Expr_2, s_1, s_0) >^{norm}$ *then the following holds:*

1. $eval(Expr_1[\backslash\mathbf{result} \leftarrow Expr_2], s_1, s_0) =\ eval(Expr_1, s_2, s_0)$

2. $s_1, s_0 \models \psi[\backslash\mathbf{result} \leftarrow Expr_2] \iff s_2, s_0 \models \psi$

The next definition defines a particular set of assertion formulas which we call valid formulas.

Definition 5.1.1 (Valid formulas) *If an assertion formula* $f \in P$ *holds in any current state and any initial state, i.e.* $\forall state, state_{init}, state, s_0 \models f$ *we say that this is a valid formula and we note it with :* $\models f$

5.2 Proof of Correctness

The correctness of our verification condition generator is established w.r.t. to the operational semantics described in Section 2.7. We look only at partial correctness, i.e. we assume that programs always terminate and we assume that there are no recursive methods.

We first give a definition that a "method is correct w.r.t its specification"

Definition 5.2.1 (A method is correct w.r.t. its specification) *For every method* m *with precondition* m.pre*, normal postcondition* m.normalPost*and exceptional postcondition function* m.excPostSpec*, we say that* m *respects its specification if for every two states* s_0 *and* s_1 *such that :*

- $\mathtt{m} : s_0 \Rightarrow s_1$

- $s_0, s_0 \models \mathtt{m.pre}$

Then if $\texttt{m}$ *terminates normally then the normal postcondition holds in the final state* s_1*:* $s_1, s_0 \models \texttt{m}.\mathsf{normalPost}$. *Otherwise, if* $\texttt{m}$ *terminates on an exception* $\texttt{Exc}$ *the exceptional postcondition holds in the poststate* s_1 $s_1, s_0 \models \texttt{m}.\mathsf{excPostSpec}(\ \texttt{Exc})$

The next issue that is important for understanding our approach is that we follow the design by contract paradigm [7]. This means that when verifying a method body, we assume that the rest of the methods respect their specification in the sense of the previuos definition 5.2.1.

First, we establish the correctness of the weakest precondition function for a single instruction: if the *wp* (short for weakest precondition) of an instruction holds in the prestate then in the poststate of the instruction the postcondition upon which the wp is caclulated holds.

Lemma 5.2.1 (Single execution step correctness) *For every instruction* s : $\texttt{instr}$*, for every state* $s_0 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, s >$ *and initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *of the execution of method* $\texttt{m}$ *if the following conditions hold:*

- $\texttt{m}.\mathsf{body}[0] : s_0 \hookrightarrow^* s_n$
- $\texttt{m}.\mathsf{body}[s] : s_n \hookrightarrow s_{n+1}$
- $s_n, s_0 \models wp(\ \mathrm{Pc}_n : \texttt{instr}, \texttt{m})$
- $\forall\ \texttt{n} : \textbf{Method}.\ \texttt{n} \neq \texttt{m}$ *n is correct w.r.t. its specification*

then :

- *if* $\mathrm{Pc}_n\ :\ \texttt{instr} \neq$ return *and the instruction does not terminate on exception,* $s_{n+1} =< \mathrm{H}_{n+1}, \mathrm{Cntr}_{n+1}, \mathrm{St}_{n+1}, \mathrm{Reg}_{n+1}, \mathrm{Pc}_{n+1} >$ *then* $s_{n+1}, s_0 \models$ *inter*$(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \texttt{m})$ *holds*
- *if* $\mathrm{Pc}_n : \texttt{instr} =$ return *then* $s_{n+1}, s_0 \models \texttt{m}.\mathsf{normalPost}$ *holds*
- *else if* $\mathrm{Pc}_n\ :\ \texttt{instr} \neq$ return *and the instruction terminates on a not handled exception* $\texttt{Exc}$ *, then* $s_{n+1}, s_0 \models \texttt{m}.\mathsf{excPostSpec}(\ \texttt{Exc}\ )$

Proof : The proof is by case analysis on the type of instruction that will be next executed. We are going to see only the proofs for the instructions return , load and invoke , the other cases being the same

1. $\mathrm{Pc}_n : \texttt{instr} =$ return

$$
\begin{array}{l}
\{\ \textit{by initial hypothesis}\ \} \\
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models wp(\texttt{m}\ \ \mathrm{return}\ , \mathrm{Pc}_n) \\
\{\ \textit{by definition the weakest precondition for}\ \ \mathrm{return}\ \ \} \\
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \texttt{m}.\mathsf{normalPost}[\backslash\textbf{result} \leftarrow \textbf{st}(\mathrm{cntr})] \\
\{\ \textit{by the substitution property 5.1.6}\ \} \\
\Longleftrightarrow \\
< \mathrm{H}_n, \mathrm{Reg}_n,\ \ eval(\textbf{st}(\mathrm{cntr}), < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0) >^{norm}, s_0 \models \mathsf{normalPost} \\
\{\ \textit{by definition of the evaluation function}\ \ eval\ \} \\
\Longleftrightarrow \\
< \mathrm{H}_n, \mathrm{Reg}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}, s_0 \models \mathsf{normalPost}
\end{array}
$$

2. $\mathrm{Pc}_n : \texttt{instr} =$ load i

{ *by initial hypothesis* }

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models wp(\mathrm{Pc}_n \ \ \text{load } i, \texttt{m})$

{ *definition of the wp function* }

$\equiv$

$$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \ \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \texttt{m}) \begin{array}{l} [\mathrm{cntr} \leftarrow \mathrm{cntr} + 1] \\ [\mathbf{st}(\mathrm{cntr} + 1) \leftarrow \mathbf{reg}(i)] \end{array}$$

{ *applying the substitution properties 5.1.5 and 5.1.4* }

$\Longleftrightarrow$

$< \mathrm{H}_n, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus \mathrm{Cntr}_n + 1 \rightarrow \mathrm{Reg}_n(i)], \mathrm{Reg}_n, \mathrm{Pc}_{n+1} >, s_0 \models$
$\mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \texttt{m})$

{ *from the operational semantics of the* load *instruction in section 2.7*}

$s_{n+1}, s_0 \models \ \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \texttt{m})$

{ *and the lemma holds in this case* }

3. new C

{ *by initial hypothesis* }

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models wp(\mathrm{Pc} \ \ \text{new } C, \texttt{m})$

{ *definition of the wp function* }

$\equiv$

$$\begin{array}{l} < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \\ \quad \forall \texttt{ref}, \mathit{not}\ \mathbf{instances}(\texttt{ref}) \wedge \\ \quad\quad \texttt{ref} \neq \mathbf{null} \Rightarrow \\ \qquad\qquad \mathit{inter}(i, i+1, \texttt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \texttt{ref}] \\ [f \leftarrow f[\oplus \texttt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)} \\ [\backslash \mathbf{typeof(}\texttt{ref}\mathbf{)} \leftarrow C] \end{array} \end{array} \quad \mathit{(1)}$$

{ *from the operational semantics of* new *in section 2.7* }

(2)$s_{n+1} =< \mathrm{H}_{n+1}, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus \mathrm{Cntr}_n + 1 \rightarrow \texttt{ref}], \mathrm{Reg}_n, \mathrm{Pc}_n + 1 >$

(3)$\mathrm{newRef}(\mathrm{H}, C) = (\mathrm{H}_{n+1}, \texttt{ref}')$

$\{$ *following Def. 4.2.2.2 instantiate (1) with* $\texttt{ref}'$ $\}$

$$
\begin{array}{ll}
 & < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \\
 & \quad \textit{not } \mathbf{instances}(\texttt{ref}') \wedge \\
 & \quad \texttt{ref}' \neq \mathbf{null} \Rightarrow \\
\textit{(4)} & \qquad \textit{inter}(i, i+1, \texttt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \texttt{ref}'] \\ [f \leftarrow f[\oplus \texttt{ref}' \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)} \\ [\backslash \mathbf{typeof}(\texttt{ref}) \leftarrow C] \end{array}
\end{array}
$$

$\{$ *from (3)* $\}$

$$
\textit{(5)} < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \begin{array}{l} \textit{not } \mathbf{instances}(\texttt{ref}') \wedge \\ \texttt{ref}' \neq \mathbf{null} \end{array}
$$

$\{$ *from (4) and (5) and Def. 4.2.2.2* $\}$

$$
\begin{array}{l}
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \\
\quad \textit{inter}(i, i+1, \texttt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \texttt{ref}'] \\ [f \leftarrow f[\oplus \texttt{ref}' \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)} \\ [\backslash \mathbf{typeof}(\texttt{ref}) \leftarrow C] \end{array}
\end{array}
$$

$\{$*from lemmas 5.1.5, 5.1.4 and 5.1.2, 5.1.3*
and the operational semantics of the instruction new $\}$

$s_{n+1}, s_0 \models \ \textit{inter}(i, i+1, \texttt{m})$

4. $\mathrm{Pc} : \texttt{instr} =$ putfield f

$\{$ *by initial hypothesis* $\}$

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \textit{wp}(\ \mathrm{Pc}_n \ \ \text{putfield } f, \texttt{m})$

$\{$ *definition of the wp function* $\}$

$\equiv$

$$
\begin{array}{ll}
 & < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \models \\
 & \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\textit{(1)} & \qquad \textit{inter}(i, i+1, \texttt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]] \end{array} \\
 & \wedge \\
 & \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m}.\mathsf{excPost}(i,\ \texttt{NullPntrExc})
\end{array}
$$

$\{$ *we get three cases* $\}$

(a) the dereferenced reference on the stack top is **null** and an exception handler starting at instruction k exists for `NullPntrExc` and Pc_n :

`instr` is in its scope

$\{$ *thus, we get the hypothesis* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathbf{st}(\mathbf{cntr}) = \mathbf{null}$
$\{$ *from the above conclusion and (1) we get* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathtt{m}.\mathsf{excPost}(\mathrm{Pc}_n,\ \mathtt{NullPntrExc})$
$\{$ *from Def. 4.5.2.2 of the function* $\mathtt{m}.\mathsf{excPost}$
??? *and the assumption that the exception is handled we get* $\}$

$$
\begin{array}{l}
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \\
\quad \forall \mathtt{ref}, \\
\qquad \neg\ \mathbf{instances}(\mathtt{ref}) \wedge \\
\qquad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\qquad\qquad \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow 0] \\ [\mathbf{st}(0) \leftarrow \mathtt{ref}] \\ [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathit{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\ \mathsf{subtype}\ (f.\mathsf{declaredIn},} \end{array}
\end{array}
$$

$\{$ *from lemmas 5.1.5, 5.1.2, 5.1.4 and 5.1.3*
and the operational semantics of putfield $\}$
$s_{n+1}, s_0 \vDash\ \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$

(b) the reference on the stack top is **null** and and the exception thrown is not handled. In this case, we obtain following the same way of reasoning as the previous case :

$$
\begin{array}{l}
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \\
\quad \forall \mathtt{ref}, \\
\qquad \neg\ \mathbf{instances}(\mathtt{ref}) \wedge \\
\qquad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\quad \mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{NullPntrExc}) \\
\qquad [\backslash \mathbf{EXC} \leftarrow \mathtt{ref}] \\
\qquad [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathit{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}, \mathsf{subtype}\ (f.\mathsf{declaredIn},\ \mathtt{Exc})}
\end{array}
$$

$\{$ *from lemmas 5.1.4, 5.1.2, 5.1.3 and*
the operational semantics of putfield $\}$
$s_{n+1}, s_0 \vDash \mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{NullPntrExc})$

(c) the reference on the stack top is not **null**

$\{$ *thus, we get the hypothesis* $\}$
$< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, s_0 \vDash \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null}$
$\{$ *from the above conclusion and (1) we get* $\}$

$$
\begin{array}{l}
< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, s_0 \vDash \\
\quad \mathit{inter}(i, i+1, \mathtt{m}) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]] \end{array}
\end{array}
$$

$\{$ *applying lemmas 5.1.5 and 5.1.2 and*
of the operational semantics of putfield $\}$
$s_{n+1}, s_0 \vDash\ \mathit{inter}(i, i+1, \mathtt{m})$

We now establish a property of the correctness of the wp function for an execution path. The following lemma states that if the calculated preconditions of all the instructions in an execution path holds then either the execution terminates normally (executing a return) or exceptionally, or another step can be made and the *wp* of the next instruction holds.

Lemma 5.2.2 (Progress) *Assume we have a method* $\mathtt{m}$ *with normal postcondition* $\mathtt{m}.\mathsf{normalPost}$ *and exception function* $\mathtt{m}.\mathsf{excPostSpec}$. *Assume that the execution starts in state*
$< \mathrm{H}_0, \mathrm{Cntr}_0, \mathrm{St}_0, \mathrm{Reg}_0, \mathrm{Pc}_0 >$ *and the there are made n execution steps causing the transitive state transition* $< \mathrm{H}_0, \mathrm{Cntr}_0, \mathrm{St}_0, \mathrm{Reg}_0, \mathrm{Pc}_0 > \hookrightarrow^n < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >$. *Assume that*
$\forall i, (\ 0 \leq i \leq n),\ \ s_i, s_0 \models wp(\ \mathrm{Pc}_i : \mathtt{instr}, \mathtt{m})$ *holds then*

1. *if* $\mathrm{Pc}_n\ :\ \mathtt{instr}\ =\ $ return *then* $<\ \mathrm{H}_n, \mathrm{Reg}_n, \mathrm{St}_n(\mathrm{Cntr}_n)\ >^{norm}, s_0\ \models$ $\mathtt{m}.\mathsf{normalPost}$ *holds.*

2. *if* $\mathrm{Pc}_n : \mathtt{instr} \neq$ athrow *throws a not handled exception of type* Exc $< \mathrm{H}_{n+1}, \mathtt{ref}, \mathrm{Reg}_n >^{exc}, s_0 \models \mathtt{m}.\mathsf{excPostSpec}(\mathrm{Exc})$ *holds where* $\mathrm{newRef}(\mathrm{H}_n, \mathrm{Exc}) = (\mathrm{H}_{n+1}, \mathtt{ref})$.

3. *if* $\mathrm{Pc}_n : \mathtt{instr} =$ athrow *throws a not handled exception of type* Exc $< \mathrm{H}_n, \mathrm{St}(\mathrm{Cntr}), \mathrm{Reg}_n >^{exc}, s_0 \models \mathtt{m}.\mathsf{excPostSpec}(\mathrm{Exc})$ *holds*

4. *else exists a state* s_{n+1} *such that another execution step can be done* $s_n \hookrightarrow s_{n+1}$ *and* $s_{n+1}, s_0 \models wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$ *holds*

Proof : The proof is by case analysis on the type of instruction that will be next executed.

We consider three cases: the case when the next execution step doesnot enter a cycle (the next instruction is not a loop entry in the sense of Def.4.3.2) the case when the current instruction is a loop end and the next instruction to be executed is a loop entry instruction (the execution step is $\rightarrow_l$) and the case when the current instruction is not a loop end and the next instruction is a loop entry instruction (corresponds to the first iteration of a loop)

1. the next instruction to be executed is not a loop entry instruction.

$\{$ *following Def. 4.5.1 of the function inter in this case* $\}$
(1) $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m}) = wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$
$\{$ *by initial hypothesis* $\}$
(2) $s_n, s_0 \models wp(\ \mathrm{Pc}_n : \mathtt{instr}, \mathtt{m})$
$\{$ *from the previous lemma 5.2.1 and (2) , we know that* $\}$
(3) $s_{n+1}, s_0 \models\ inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$
$\{$ *from (1) and (3)* $\}$
$s_{n+1}, s_0 \models wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$

2. Pc_n : $\mathtt{instr}$ is not a loop end and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of bytecode instructions of the method $\mathtt{m}$(i.e. the execution step is of kind $\rightarrow^l$). Thus, there exists a natural number $i, 0 \leq i < \mathtt{m}.\mathsf{loopSpecS}.length$ such that $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{pos} = loopEntry$, $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{invariant} = I$ and $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$. We look only at the case when the current instruction is a load instruction

{ *by initial hypothesis* }

$s_n, s_0 \models wp(\ \mathrm{Pc}_n \ \ \mathrm{load}\ i, \mathtt{m})$

{ *by defintion of the wp function in section 4.5 of the previous chapter* }

$$s_n, s_0 \models \ inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m}) \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(j)]\end{array}$$

{*by the definition 4.5.1 for the case when the execution step is not a backedge but the target instruction is a loop entry* }

$$\begin{array}{l} s_n, s_0 \models \\ \quad I \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(i)]\end{array} \\ \quad \wedge \\ \quad \forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})) \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(i)]\end{array} \end{array}$$

{ *from lemmas 5.1.5 and 5.1.4* }

$\Longleftrightarrow$

$$\begin{array}{l} s_n \begin{array}{l}[\mathrm{Cntr} \leftarrow \ eval(\mathbf{cntr} + 1, s_n, s_0)] \\ [\mathrm{St} \leftarrow \mathrm{St}[\oplus(\ eval(\mathbf{cntr} + 1, s_n, s_0)) \rightarrow \ eval(\mathbf{reg}(i), s_n, s_0)]]\end{array}, s_0 \models \\ \quad I \wedge \\ \quad \forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})) \end{array}$$

{ *from the Def. 4.2.2 of the evaluation function* }

$\equiv$

$$\begin{array}{l} s_n \begin{array}{l}[\mathrm{Cntr} \leftarrow \mathrm{Cntr} + 1] \\ [\mathrm{St} \leftarrow \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \mathrm{Reg}(i)]]\end{array}, s_0 \models \\ \quad I \wedge \\ \quad \forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})) \end{array}$$

{ *from the operational semantics of* load }

$$s_{n+1}, s_0 \models \begin{array}{l} I \wedge \\ \forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})) \end{array}$$

{ *we can get from the last formulation* }

(1) $s_{n+1}, s_0 \models I$

(2)$s_{n+1}, s_0 \models I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$

{ *from (1) and (2)* }

$s_{n+1}, s_0 \models wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$

3. Pc_n : $\mathtt{instr}$ is an end of a cycle and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of

bytecode instructions of the method $\mathtt{m}$(i.e. the execution step is of kind $\to^{l}$). Thus, there exists a natural number $i, 0 \leq i < \mathtt{m}.\mathsf{loopSpecS}.length$ such that $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{pos} = loopEntry$, $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{invariant} = I$ and $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$. We consider the case when the current instruction is a sequential instruction. The cases when the current instruction is a jump instruction are similar.

{ *by hypothesis we get* }

$$s_n, s_0 \models wp(\ \mathrm{Pc}_n : \mathtt{instr}, \mathtt{m})$$

{ *from Def. 4.5.1 and transformation over the above statement* }

$$\textit{(1)} \quad s_{n+1}, s_0 \models I$$

{ *by hypothesis, loopEntry* $= \mathrm{Pc}_{n+1}$. *From def. 4.3.2, we conclude that there is a prefix subP* $= \mathtt{m}.\mathsf{body}[0] \to^{*}$ *loopEntry* : $\mathtt{instr}$ *of the current execution path which does not pass through* $\mathrm{Pc}_n : \mathtt{instr}$. *We can conclude that the transition between loopEntry* : $\mathtt{instr}$ *and its predecessor* $k : \mathtt{instr}$ *(which is at index k in* $\mathtt{m}.\mathsf{body}$*) in the path subP is not a backedge. By hypothesis we know that* $\forall i, 0 \leq i \leq n, s_i, s_0 \models wp(\ \mathrm{Pc}_i : \mathtt{instr}, \mathtt{m})$. *From def.4.5.1 and lemma 5.2.1 we conclude* }

$$\exists k, 0 \leq k \leq n \Rightarrow$$

$$\textit{(2)} \qquad \begin{array}{l} s_k, s_0 \models \\ I \\ \wedge \forall mod_i, i = 1..s(\begin{array}{l} I \Rightarrow \\ wp(\ loopEntry : \mathtt{instr}, \mathtt{m}) \end{array}) \end{array}$$

$$s_k =^{\mathsf{modif}} s_{n+1}$$

{ *by we have* $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$ *and from (2)* }

$$\textit{(3)}\ s_{n+1}, s_0 \models I \Rightarrow wp(\ loopEntry : \mathtt{instr}, \mathtt{m})$$

{ *from (1) and (3)* }

$$\begin{array}{l} s_{n+1}, s_0 \models wp(\ loopEntry : \mathtt{instr}, \mathtt{m}) \\ \Longleftrightarrow \\ s_{n+1}, s_0 \models wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m}) \end{array}$$

Qed.

Lemma 5.2.3 (*wp* of method entry point instruction) *Assume we have a method* $\mathtt{m}$. *Assume that execution of method* $\mathtt{m}$ *starts execution in state* s_0 *and*

$s_0, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[0], \mathtt{m})$ *where and makes* n *steps to reach state* s_n*:* $s_0 \hookrightarrow^n s_n$*, then*

$$\forall i, 0 < i \leq n,\ s_i, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[\mathrm{Pc}_i], \mathtt{m})$$

Proof : Induction over the number of execution steps n

1. $s_0 \hookrightarrow s_1$. From the initial hypothesis we can apply lemma 5.2.2, we get that $s_1, s_0 \vDash wp(\ \mathrm{Pc}_1 : \mathtt{instr}, \mathtt{m})$ and thus, the case when one step is made from the initial state s_0 holds

2. Induction hypothesis: $s_0 \hookrightarrow^{n-1} s_{n-1}$ and $\forall i, 0 < i \leq n-1,\ s_i, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[\mathrm{Pc}_i], \mathtt{m})$ and there can be made one step $s_{n-1} \hookrightarrow s_n$. Lemma 5.2.2 can be applied and we get that

   (1) $s_n, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[\mathrm{Pc}_n], \mathtt{m})$. From the induction hypothesis and *(1)* follows that
   $$\forall i, 0 < i \leq n,\ s_i, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[\mathrm{Pc}_i], \mathtt{m})$$

Lemma 5.2.4 (Validity of *wp* for a method implies that postcondition holds) *Assume we have a method* $\mathtt{m}$ *with normal postcondition* $\mathtt{m}.\mathsf{normalPost}$ *and exception function* $\mathtt{m}.\mathsf{excPostSpec}$.

Assume that execution of methodd $\mathtt{m}$ *starts in state* $_0$ *and* $s_0, s_0 \vDash wp(0\ \mathtt{m}.\mathsf{body}[0], \mathtt{m})$ *Then if the method* $\mathtt{m}$ *terminates, i.e. there exists a state* s_n*,* $s_0 \hookrightarrow^* s_n$ *such that* $\mathrm{Pc}_n : \mathtt{instr} =\ \mathrm{return}$ *or* $\mathrm{Pc}_n : \mathtt{instr}$ *throws an unhandled exception of type* $\mathtt{Exc}$ *the following holds:*

- *if* $\mathrm{Pc}_n : \mathtt{instr} =\ \mathrm{return}$ *then* $s_{n+1}, s_0 \vDash \mathtt{m}.\mathsf{normalPost}$
- *if* $\mathrm{Pc}_n : \mathtt{instr}$ *throws a not handled exception of type* $\mathtt{Exc}$ *then* $s_{n+1}, s_0 \vDash \mathtt{m}.\mathsf{excPostSpec}(\mathtt{Exc})$

Proof: Let $s_0 \hookrightarrow^* s_n$ and $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_n]$ is a return or an instruction that throws a not handled exception. Applying lemma 5.2.3, we can get that $s_n, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[\mathrm{Pc}_n], \mathtt{m})$. We apply lemma 5.2.1 for the case for a return or instruction that throws an unhandled exception which allows to conclude that the current statement holds.

Now, we are ready to state the theorem which expresses the correctness of our verification condition generator w.r.t. the operational semantics of our language

Theorem 5.2.5 *For any method* $\mathtt{m}$ *if the verification condition is valid:*

$$\vDash \mathtt{m}.\mathsf{pre} \Rightarrow wp(\ \mathrm{m}\ .\mathsf{body}[0], \mathtt{m})$$

then $\mathtt{m}$ *is correct in the sense of the definition 5.2.1.*

Proof: From lemma 5.2.4 and the initial hypothesis that the weakest precondition of the entry point holds we conclude that the method $\mathtt{m}$ is correct

Chapter 6

Equivalence between Java source and bytecode proof Obligations

Chapter 7

A compact verification condition generator

Chapter 8

Applications

Chapter 9

Conclusion

Bibliography

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

[2] Fabian Bannwart and Peter Muller. A program logic for bytecode. In *Bytecode 2005*, ENTCS, 2005.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In "G.Barthe, L.Burdy, M.Huisman, J.Lanet, and T.Muntean", editors, *CASSIS workshop proceedings*, LNCS, pages 49–69. Springer, 2004.

[4] Gilles Barthe, Guillaume Dufay, Line Jakubiec, and Simao Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *VMCAI*, pages 32–45, 2002.

[5] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A formal executable semantics of the JavaCard platform. *Lecture Notes in Computer Science*, 2028:302+, 2001.

[6] Nick Benton. A typed logic for stack and jumps. DRAFT, 2004.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.

[13] Stephen N. Freund and John C. Mitchell. A formal framework for the java bytecode language and verifier. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 147–166, New York, NY, USA, 1999. ACM Press.

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

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[16] B. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective, 2003.

[17] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

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

[19] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. In *Journal of Automated Reasoning 2003*, 2003.

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

[21] M.Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs.

[22] Cornelia Pusch. Proving the soundness of a java bytecode verifier in isabelle/hol, 1998.

[23] Zhenyu Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[24] C.L. Quigley. A programming logic for Java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

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

[26] R.W.Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *volume 19 of Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[27] I. Siveroni. Operational semantics of the java card virtual machine, 2004.

[28] Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. In *Proceedings of the 15th European Symposium on Programming (ESOP05)*, 2005. to appear.