Bytecode Verification and its Applications

June 6, 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 17
2.7 Bytecode Language and its Operational Semantics 19

3 Specification language for Java bytecode programs **27**
3.1 A quick overview of JML 28
3.2 BML 29
3.2.1 Notation convention 29
3.2.2 BML Grammar 30
3.2.3 Interpretation of the BML grammar 32
3.3 Background information of the class file format 39
3.3.1 The constant pool table 39
3.3.2 Representation of method in the class file format 40
3.4 BML type checker 42
3.4.1 Well formed BML expressions 42
3.5 Compiling JML into BML 43

4 Verification condition generator for Java bytecode **47**
4.1 Related work 48
4.2 Representing bytecode programs as control flow graphs 49
4.3 Extending method declarations with specification 51
4.4 Weakest precondition calculus 52
4.4.1 Intermediate predicates 52

4.4.2 Weakest precondition in the presence of exceptions 53
4.4.3 Rules for single instruction 54
4.5 Example 60

5 Correctness of the verification condition generator 63
5.1 Substitution 63
5.2 Interpretation of assertions in a state 64
5.3 Proof of Correctness 67

6 Equivalence between Java source and bytecode proof Obligations 73

7 A compact verification condition generator 75

8 Applications 77

9 Conclusion 79

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 [12](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.

we do not know how to verify programs in presence of 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 [8] by N.Freund and J.Mitchell and [14] 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 [13] 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 [10], 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 [3, 2] 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 [**?**] 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 $A_1 \ \ldots \ A_n$ 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:

$$inList : list\ A * A \rightarrow\ bool$$

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

$$\text{cons} : list\ A * A \rightarrow list\ A$$

The function $inDom(f, e)$ determines if the element e is in the domain 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:

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

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

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

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):

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

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*) and the entry point instruction of the method (*entryPnt*) (the instruction at which every execution of the method starts), the exception handler table (*excHndlS*) and the list of exceptions (*exceptions*) that the method may throw

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

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. $\texttt{m}.\mathit{entryPnt} = \texttt{m}.\mathit{body}[0]$.

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.

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

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

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

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

$$\begin{array}{ll} \mathit{JType} & ::= \text{ int} \mid \text{RefType} \\ \text{RefType} & ::= \text{RefCl} \mid \text{RefArr} \\ \text{RefCl} & ::= \mathit{Class} \\ \text{RefArr} & ::= \mathit{JType}[] \end{array}$$

Our language supports two kinds of values : values of the basic type int and reference values

$$\begin{array}{ll} \text{Values} & ::= i, i \in \text{ intliteral} \mid \text{RefVal} \\ \text{RefVal} & ::= ref : C, C \in \text{RefCl} \mid \text{RefValArr} \mid \textbf{null} \\ \text{RefValArr} & ::= refArr : C[], C[] \in \text{RefArr} \end{array}$$

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

$$\mathit{defVal}(\mathtt{T}) = \begin{cases} \textbf{null} & \mathtt{T} \in \text{RefVal} \\ 0 & \mathtt{T} = \text{ int} \end{cases}$$

We define also a subtyping relation as follows:

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

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

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

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
- Pcstands 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 which is a partial function mapping references of the heap into their dynamic types

Formally, the data type HeapType has the following structure:

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

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.

A heap object H must assure 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:

$$\begin{array}{ll} \forall f : \mathit{Field}, \forall\ \texttt{ref}\ : C', & \mathit{inDom}\ (\mathrm{H.Fld}(f),\ \texttt{ref}\ ) \Rightarrow \\ & \mathit{inList}(\ \texttt{ref}\ , \mathrm{getLoc(H)}\ ) \wedge \\ & f.\mathit{declaredIn} = C \wedge \\ & \mathsf{subtype}\ (C', C) \\ \wedge & \\ \forall\ \texttt{ref}\ : T[\,], & \mathit{inDom}\ (\mathrm{H.Arr}, (\ \texttt{ref}\ , i)) \Rightarrow \\ & \mathit{inList}(\mathrm{H.Loc},\ \texttt{ref}\ ) \wedge \\ & 0 \leq i < \mathrm{H.Fld(arrLength)}(\ \texttt{ref}\ ) \end{array}$$

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:

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

Formally, the operation is defined as follows:

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

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:

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

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

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:

$$\mathrm{newRef} : \mathrm{H} \rightarrow \mathrm{RefCl}\ \rightarrow \mathrm{H} * ref$$

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

$$\text{newRef}(\text{H}, C) = (\text{H}',\ \texttt{ref}\ ) \iff^{def}$$

$$\text{allocator}(\text{H},\ \texttt{ref}\ ) = \text{H}' \wedge \left\{ \begin{array}{ll} \forall f : \mathit{Field}, & \mathit{instFlds}(f, C) \Rightarrow \\ & \text{H}'.\mathsf{Fld} := \text{H}'.\mathsf{Fld}[\oplus f \rightarrow f[\oplus\ \texttt{ref}\ \rightarrow \mathit{defVal}(f.\mathit{Type})]] \\ & \text{H}'.\mathsf{TypeOf}\ := \text{H}'.\mathsf{TypeOf}\ [\oplus\ \texttt{ref}\ \rightarrow C] \end{array} \right\}$$

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

$$\text{newArrRef} : \text{H} \rightarrow \text{RefArr} \rightarrow \text{H} * refArr$$

$$\text{newArrRef}(\text{H}, \texttt{T}[\,], l)\ = (\text{H}',\ \texttt{ref}\ ) \iff^{def}$$

$$\begin{array}{l} \text{allocator}(\text{H},\ \texttt{ref}\ ) = \text{H}' \wedge \\ \text{H}'.\mathsf{Fld} := \text{H}'.\mathsf{Fld}[\oplus\text{arrLength} \rightarrow \text{arrLength}[\oplus\ \texttt{ref}\ \rightarrow l]] \wedge \\ \forall i, 0 \geq i < l \Rightarrow \text{H}'.\mathsf{Arr} := \text{H}'.\mathsf{Arr}[\oplus(\ \texttt{ref}\ , i) \rightarrow \mathit{defVal}(\texttt{T})] \\ \text{H}'.\mathsf{TypeOf}\ := \text{H}'.\mathsf{TypeOf}\ [\oplus\ \texttt{ref}\ \rightarrow \texttt{T}[\,]] \end{array}$$

In the following, we adopt few more naming convention which do not create any ambiguity. Getting the function corresponding to a field f in a heap H : $\text{H}.\mathsf{Fld}(f)$ is replaced with $\text{H}(f)$ 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: $\text{H}.\mathsf{Arr}(\ \texttt{ref}\ , i)$ becomes $\text{H}(\ \texttt{ref}\ , i)$.

Whenever the field f for the object pointed by reference `ref` is updated with the value *val*, the component $\text{H}.\mathsf{Fld}$ 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 $\text{H}.\mathsf{Arr}$:

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

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:

$$\text{H}[\oplus(\ \texttt{ref}\ , i) \rightarrow val]$$

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 also 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.

We have several functions which model the exception handling mechanism. 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:

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

$$getStateOnExc\ (< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, E, \mathrm{Pc}, excH[]) =$$
$$\begin{cases} < \mathrm{H}', 0, \mathrm{St}[\oplus 0 \rightarrow \texttt{ ref }], \mathrm{Reg}, handlerPc > & \text{if } findExcHandler(\mathrm{E},\mathrm{Pc},excH[]) = handlerPc \\ < \mathrm{H}', \mathrm{Reg}, \texttt{ ref } >^{exc} & \text{if } findExcHandler(\mathrm{E},\mathrm{Pc},excH[]) = \bot \end{cases}$$

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

If an exception E is thrown by instruction at position i while executing the method `m`, the exception handler table `m`.*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 $\perp$:

$$findExcHandler \ : ExcType * nat * ExcHandler[] \rightarrow nat$$

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

where

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

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.

Definition 2.7.0.1 (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 2.7.0.2 (Transitive closure of a state transition relation) *If the body of the method* m m.*body starts execution in a state with configuration* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and there is an execution path from* m *.entryPnt to an instruction* m *.body[k] which is either a* return *instruction or an instruction*

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

Figure 2.1: Bytecode Language instructions

which terminates execution with an uncaught exception and the configuration after its execution is $< \mathrm{H}', \mathrm{Reg}, \mathit{Final} >^{final}$ *we denote it with*

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

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:

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

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

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

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

The condition $\texttt{cond} = \{=, \neq, \leq, <, >, \geq\}$ 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

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

Transfers control to the instruction at position n.

3. return

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

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

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

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).

case for arithmetic instructions that throw exception

- Load Store instructions

1. load

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

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

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

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

3. iinc

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

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

4. push

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

5. pop

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

- Object creation and manipulation

1. new `Cl`

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

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

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

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

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

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

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

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 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 $\mathtt{ref}_{NullPointerExc}$ is pushed onto the operand stack

4. newarray `T`

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

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

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

[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`

5. type_astore

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

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

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

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.

say what assignment compatible is

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

one more case of exceptional termination if it terminates on an exception

6. type_aload

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

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

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

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 RefCl 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

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

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

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

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

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

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

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

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

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

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

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

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]

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

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

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

The first top $meth.nArgs$ 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.

[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

Chapter 3

Specification language for Java bytecode programs

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

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

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

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

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

In section 3.1, we give an overview of the main features of JML. A detailed overview of BML is given in section 3.2. As we stated before, we support also a compiler from the high level specification language JML into BML. The

compilation process from JML to BML is discussed in section 3.5. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

3.1 A quick overview of JML

JML [9] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications. JML follows the design-by-contract approach (see [4]), 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 [5]). 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.* [**?**]). 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 [6]. 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 [11] or dynamic checking by tools such as the assertion checker jmlrac [7]. An overview of the JML tools can be found in [5].

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

3.2 BML

BML corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties. The following Def. 3.2.2 gives the formal grammar of BML. The formal grammar of BML is given in the next definition.

3.2.1 Notation convention

- Nonterminals are written with a *italics* font

- Terminals are written with a **boldface** font

- brackets [ ] surround optional text.

3.2.2 BML Grammar

constants	::= *intLiteral* | *signedIntLiteral* | **null** | *ident*
signedIntLiteral	::= $+$*nonZerodigit*[*digits*] | $-$*nonZerodigit*[*digits*]
intLiteral	::= *digit* | *nonZerodigit*[*digits*]
digits	::= *digit*[*digits*]
digit	::= **0** | *nonZerodigit*
nonZerodigit	::= **1** | ... | **9**
ident	::= # *intLiteral*
boundVar	::= # **bv_***intLiteral*
$\mathcal{E}$	::= *constants* | **reg**(*digits*) | $\mathcal{E}$**.***ident* | *ident* | **arrayAccess**($\mathcal{E}, \mathcal{E}$) | $\mathcal{E}$ *op* $\mathcal{E}$ | **cntr** | **st**($\mathcal{E}$) | **\typeof(**$\mathcal{E}$**)** | **\type(***ident***)** | **\elemtype(**$\mathcal{E}$**)** | **\old(**$\mathcal{E}$**)** | **\EXC** | **\result** | *boundVar*
op	::= **+** | **-** | **mult** | **div** | **rem**
$\mathcal{R}$	::= $=\|\neq\|\leq\|\leq\|\geq\|>\|<:$

$$
\begin{array}{lll}
\mathcal{F}^{bc} & ::= & \mathcal{E}_1\ \mathcal{R}\ \mathcal{E}_2 \\
& | & \textbf{true} \\
& | & \textbf{false} \\
& | & \mathit{not}\ \mathcal{F}^{bc} \\
& | & \mathcal{F}^{bc} \wedge \mathcal{F}^{bc} \\
& | & \mathcal{F}^{bc} \vee \mathcal{F}^{bc} \\
& | & \mathcal{F}^{bc} \Rightarrow \mathcal{F}^{bc} \\
& | & \mathcal{F}^{bc} \iff \mathcal{F}^{bc} \\
& | & \forall\ \mathit{boundVar}, \mathcal{F}^{bc} \\
& | & \exists\ \mathit{boundVar}, \mathcal{F}^{bc} \\
\\
\mathit{classSpec} & ::= & \textbf{ClassInv}\ \mathcal{F}^{bc} \\
& | & \textbf{ClassHistoryConstr}\ \mathcal{F}^{bc} \\
& | & \textbf{declare ghost}\ \mathit{ident}\ \mathit{ident} \\
\\
\mathit{intraMethodSpec} & ::= & \textbf{atIndex}\ \mathit{nat}; \\
& & \mathit{assertion}; \\
\\
\mathit{assertion} & ::= & \mathit{loopSpec} \\
& | & \textbf{assert}\ \mathcal{F}^{bc} \\
& | & \textbf{set}\ \mathcal{E}\ \mathcal{E} \\
\\
\mathit{loopSpec} & ::= & \textbf{loopInv}\ \mathcal{F}^{bc}; \\
& & \textbf{loopModif}\ \mathit{list}; \\
& & \textbf{loopDecreases}\ \mathcal{E}; \\
\\
\mathit{methodSpec} & ::= & \mathit{specCase} \\
& | & \mathit{specCase}\ \textbf{also}\ \mathit{methodSpec} \\
\\
\mathit{specCase} & ::= & \textbf{requires}\ \mathcal{F}^{bc}; \\
& & \textbf{modifies}\ \mathit{list}\ \mathit{locations}; \\
& & \textbf{ensures}\ \mathcal{F}^{bc}; \\
& & \mathit{exsuresList} \\
\\
\mathit{exsuresList} & ::= & [] \mid \textbf{exsures}\ (\mathit{ident})\ \mathcal{F}^{bc}; \mathit{exsuresList} \\
\\
\mathit{locations} & ::= & \mathcal{E}.\mathit{ident} \\
& | & \textbf{reg}(i) \\
& | & \mathit{arrayModAt}(\mathcal{E}, \mathit{specIndex}) \\
& | & \textbf{everything} \\
& | & \textbf{nothing} \\
\\
\mathit{specIndex} & ::= & \text{all} \mid i_1..i_2 \mid i
\end{array}
$$

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

3.2.3 Interpretation of the BML grammar

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

BML expressions

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

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

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

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

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

The expressions that cannot be translated in JML are related to the way in which the virtual machine works, i.e. we refer to the stack and the stack counter. Because intermediate calculations are done by using the stack, often we will need stack expressions in order to characterise the states before and after an instruction execution. Let's see how stack expressions are represented in BML:

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

BML predicates

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

Class Specification

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

```
class C {
   int a ;

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

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

Figure 3.2: An example for class specification

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

We give in Fig.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.

reference to JML desugaring

Method specification case

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

- **requires** $\mathcal{F}^{bc}$ which represent the precondition of the specification case. If such a clause is not explicitely written in the specification, then the default precondition *true* is implicite
- **ensures** $\mathcal{F}^{bc}$ which stands for the normal postcondition of the method in case the precondition held in the prestate. In case this clause is not written in the specification explicitely, then the default postcondition *true* must hold.
- **modifies** *list locations* which is the frame condition of the specification case and denotes the the locations that may be modified by the method if the precondition of this specification case holds in the prestate. This in particular means that a location that is not mentioned in the **modifies** clause may be modified. If the modifies clause is omitted, then the default modifies specification is **modifies everything**
- *exsuresList* is the list of the exceptional postconditions that should hold in this specification case. In particular, every element in the list of exceptional postconditions has the following structure **exsures** (*ident*) $\mathcal{F}^{bc}$. Note that at index *ident* there is a constant which stands for some exception class `Exc`. The semantics of such a specification expression is that if the method containing the exceptional postcondition terminates on an exception of type `Exc` then the predicate denoted by $\mathcal{F}^{bc}$ must hold in the poststate. Note that the list of exceptional postcondition may be empty. Also the list of exceptional postconditions might not be complete w.r.t. exceptions that may be thrown by the method. In both cases, for every exception that might be thrown by the method for which no explicite exceptional postcondition is given, we take the default exceptional postcondition *false*

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

```
class C {
   int a ;

   invariant a > 0;
   historyConstraint  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

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

- **atIndex** *nat* specifies the index of the instruction which identifies the instruction to which the specification refers. We would like to note here that the style of specification in BML is slightly different from the JML style. First, JML specification is written directly in the source code in comments at the point in the program text where the specification must hold. Second, as the Java source language is structured, JML allows to specify a particular program structure. For instance, in Fig. 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 the instruction at the index specified by **atIndex** *nat*. We allow the following local assertions:
  - *loopSpec* gives the specification of a loop. It has the following syntax:
    - **loopInv** $\mathcal{F}^{bc}$ where $\mathcal{F}^{bc}$ is the property that must hold whenever the corresponding loop entry instruction is reached during execution
    - **loopModif** *list loc* is the list of locations modified in the loop. This means that at the borders of every iteration (beginning and end), all the expressions not mentioned in the loop frame condition must have the same value.
    - **loopDecreases** $\mathcal{E}$ specifies the expression $\mathcal{E}$ which guarantees loop termination. The values of $\mathcal{E}$ must be from a well founded set (usually from int type) and the values of $\mathcal{E}$ should decrease at every iteration
  - **assert** $\mathcal{F}^{bc}$ specifies the predicate $\mathcal{F}^{bc}$ that must hold at the corresponding position in the bytecode
  - **set** $\mathcal{E}$ $\mathcal{E}$ is a special expression that allows to set the value of a specification ghost variable. This means that the first argument must denote a reference to a ghost variable, while the second expression is the new value that this ghost variable is assigned to.

Frame conditions

As we already saw, method or loop specifications might declare the locations that are modified by the method / loop. We use the same syntax in both

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

- $\mathcal{E}$**.***ident* states that the method / loop modifies the value of the field at index *ident* in the constant pool for the reference denoted by $\mathcal{E}$
- **reg**(i) states that the local variable may modified by a loop. Note that this kind of modified expression 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*$(\mathcal{E}, \mathit{specIndex})$ states that the components at the indexes specified by *specIndex* in the array denoted by $\mathcal{E}$ may be modified. The indexes of the array components that may be modified *specIndex* have the following syntax:
  - i is the index of the component at index i. For instance, *arrayModAt*$(\mathcal{E}, i)$ means that the array component at index i might be modified. Of course, in order that such a specification make sense the following must hold: $0 \leq i <$ arrLength$(\mathcal{E})$
  - all specifies that all the components of the array may be modified, i.e. the expression *arrayModAt*$(\mathcal{E}, \text{all})$ is a syntactic sugar for

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

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

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

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

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

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

3.3 Background information of the class file format

In this section, we introduce few data structures of the class file format which are important for the understanding of the coming sections.

In particular, we will focus on the
CP data structure as well as of several optional data structures contained in the class file: the **LV** and
the **Line_Number_Table**. In what follows, we give a consize description of these class file data structures.

3.3.1 The constant pool table

Recall that a class file defines a single class or interface and contains information about the class name, interfaces implemented by the class, super class, methods and fields declared in the class and references. The JVM [12] mandates that the class file contains data structure usually referred as the constant pool table which is used to construct the runtime constant pool upon class or interface creation. The runtime constant pool serves for loading, linking and resolution of references used in the class. In the rest of the chapter we will denote the constant pool with **CP**. The **CP**has the following structure:

$$\mathbf{CP} = \left\{ \begin{array}{ll} \mathsf{cpLength} & : nat \\ \mathsf{cpElems} & : \mathbf{cpConst}[] \end{array} \right\}$$

The first field cpLength contains the length of the second component cpElems. cpElemsis an array containing the class constants. The JVM specification defines seven kinds of data structures that can be elements of the constant pool. However, here we will look only on the following elements

$$\begin{array}{rl} \mathbf{cpConst} & ::= \mathbf{CONSTANT\_Fieldref\_info} \\ & \mid \mathbf{CONSTANT\_Class\_info} \\ & \mid \mathbf{CONSTANT\_Method\_info} \\ & \mid \mathbf{CONSTANT\_Utf8\_info} \\ & \mid \mathbf{CONSTANT\_NameAndType\_info} \end{array}$$

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

- every field which is dereferenced in any of the methods of the current class. The corresponding data structure which stands for a particular field constant reference is **CONSTANT_Fieldref_info**. Its structure is given in Fig.3.4. The figure shows that a constant field reference data structure contains information about the class or interface where the field is declared (the second element of the data structure, **class_index**) as well a description of its name and type (the field **name_and_type_index**)

- every class referenced in the current class. A class constant is stored in the constant pool in a **CONSTANT_Class_info** data structure

$$\textbf{CONSTANT\_Fieldref\_info} = \left\{ \begin{array}{ll} \textbf{tag} & : nat \\ \textbf{class\_index} & : nat \\ \textbf{name\_and\_type\_index} & : nat \end{array} \right\}$$

- **tag** is a tag of one byte whose value determines without ambiguity that the current attribute describes a field reference constant
- **class_index** The value of this item must be a valid index into the **CP** table. The **CP** entry at that index must be a **CONSTANT_Class_info** structure representing the class or interface type that contains the declaration of the field or method.
- **name_and_type_index** The value of the item must be a valid index into the **CP** table. The **CP** entry at that index must be a **CONSTANT_NameAndType_info** structure. This **CP** entry indicates the name and descriptor of the field.

Figure 3.4: STRUCTURE OF THE **CONSTANT_Fieldref_info** ATTRIBUTE

- every method which is called in any of the methods of the current class. It is represented by a **CONSTANT_Method_info** data structure
- every string constant value is represented as **CONSTANT_Utf8_info**
- **CONSTANT_NameAndType_info** structure is used to represent a field or method, without indicating which class or interface type it belongs to. It contains only information about the type and the source name of the field or method. (see for more detailed explanation [12], section 4.4)

3.3.2 Representation of method in the class file format

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

The local variable table attribute

The local variable table attribute is an optional variable-length attribute of a method **Code** attribute. We will denote this data structure as **LV**. It may be used by debuggers to determine the value of a given local variable during the execution of a method. In Fig. 3.3.2 we give a modelization of this attribute specified by the JVM specification as well as the meaning of its components. Note that the JVM allows that there might be more than one **LV** attribute per local variable in the **Code** attribute. This means that a bytecode local variable might contain even values from incompatible types at different places of the program. Note, that in our compiler presented later in Section 3.5, we assume that every source method local variable and parameter is compiled to a unique bytecode local variable which is different from the compilation of any other local variable or parameter in the method. The structure of **LV** follows hereafter:

$$\mathbf{LV} = \left\{ \begin{array}{ll} \mathsf{attribute\_name\_index} & : nat \\ \mathsf{attribute\_length} & : nat \\ \mathsf{local\_variable\_table} & : \mathbf{localVarTableElem}[] \end{array} \right\}$$

Let us see what is the meaning of the components of this data structure:

- **attribute_name_index** The value of this item must be a valid index into the **CP** table.
- **attribute_length**. The value of the item indicates the length of the attribute, excluding the initial six bytes.
- **local_variable_table_length**. The value of the item indicates the number of entries in the **local_variable_table** array.
- **local_variable_table**[]. Each entry in the **local_variable_table** array indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry is of type **localVarTableElem**

The data structure **localVarTableElem**has the following structure:

$$\mathbf{localVarTableElem} = \left\{ \begin{array}{ll} \mathsf{start\_pc} & : nat \\ \mathsf{length} & : nat \\ \mathsf{name\_index} & : nat \\ \mathsf{descriptor\_index} & : nat \\ \mathsf{index} & : nat \end{array} \right\}$$

The meaning of the elements of **localVarTableElem**is the following:

- **start_pc**, **length**. The given local variable must have a value at indices into the code array in the interval [**start_pc**, **length**], that is, between **start_pc** and **start_pc** + **length** inclusive. The value of **start_pc** must be a valid index into the code array of this **Code** attribute and must be

the index of the opcode of an instruction. Either the value of **start_pc** + **length** must be a valid index into the code array of this **Code** attribute and be the index of the opcode of an instruction, or it must be the first index beyond the end of that code array.

- **name_index**,**descriptor_index**. The value of the **name_index** item must be a valid index into the **CP** table. The **CP** entry at that index must contain a data structure representing a valid local variable name

  The value of the **descriptor_index** item must be a valid index into the **CP** table. The **CP** entry at that index must contain a data structure representing a field descriptor encoding the type of a local variable in the source program.

- **index**. The given local variable must be at index in the local variable array of the current frame. If the local variable at index is of type double or long, it occupies both index and index+1.

The line number table attribute

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

3.4 BML type checker

In Section 3.2, 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 what follows, we scatch the rules of a typechecker which recognizes a well formed specification

3.4.1 Well formed BML expressions

We now turn to few rules which check if a BML expression is correct. We first need to define several functions that will be used by the type checker.

Note that it is a partial function and is defined

$$\frac{\begin{array}{l}\mathbf{CP}, \vdash \mathcal{E} \\ 0 \leq \mathit{ident} < \mathbf{CP}.\mathit{length} \\ \mathbf{CP}(\mathit{ident}) = \mathbf{CONSTANT\_Fieldref} \\ \mathsf{getType}(\mathcal{E}) = \mathbf{CP}(\mathbf{CP}(\mathit{ident}).\mathit{class\_index})\end{array}}{\mathbf{CP}, \vdash \mathcal{E}.\mathit{ident}} \; wf\ field\ access$$

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([12], 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 [12], 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 [9]). 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 [**?**] For instance, a Java method which has two normal behaviours is given in Fig. 3.5. 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. 3.5 and particularly, let's look at the first specification case of method `m` whose precondition `a¿` b contains the identifier `a`. In the linking phase, this identifier is resolved to the field named `a` which is declared in the same class as shown in the figure. Also in this precondition, the identifier `b` which is resolved to the parameter of method `m`.

4. Compilation of the JML specification into BML

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

```
class C {
   int a ;

   invariant a > 0;
   historyConstraint  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 3.5: 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. Concretely, this means that if a Java source variable has a boolean type

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

$$\Longleftrightarrow$$

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

Figure 3.6: The compilation of the postcondition in Fig. 3.1

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. 3.6. From the postcondition compilation, one can see that the expression **result** has integer type and the equality between the boolean expressions in the postcondition in Fig.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. 3.7. This attribute is an array of data structures each describing a single loop from the method source code. Also for each loop in the source code there must be a corresponding element in the array. More precisely, every element contains information about the instruction where the loop starts as specified in the **Line_Number_Table**, the locations that can be modified in a loop iteration, the invariant associated to this

[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 3.7: Structure of the Loop Attribute

loop and the decreasing expression in case of total correctness,

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

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 **??**, 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 4.2 introduces the notion of a loop in an unstructured program.

As we stated earlier, our verification condition generator is based on a weak-

est precondition (wp) calculus. As we shall see in Section 4.4 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

There exist several formal techniques for establishing that a program respects certain property.

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 [**?**].

Floyd is among the first to work on program verification using logic methods for unstructured languages (see [**?**]). Following the Floyd's approach, T.Hoare gives a formal logic for program verification in [**?**] known today under the name Hoare logic. Dijkstra [**?**] 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.

The nineties upto nowadays give rise to several verification tools based on Hoare logic. Among the ones that gained most popularity are esc/java developed at Compaq [11], the Loop tool [**?**], Krakatoa, Jack [6] 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 [**?**] 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 [**?**] 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 [**?**] 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 [**?**], 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.

not well explained the verification technique they have

The Spec# ([**?**]) 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 [**?**] 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 transformations for several reasons. First, we need a verification procedure as simple as possible, and then every transformation must be proven correct which is not always trivial.

Another topic related to the present work is PCC. PCC and the certifying compiler were proposed by Necula (see [**?**, **?**, **?**]). PCC is an architecture for establishing trust in untrusted code in which the code producer supplies a proof for correctness with the code. The initial idea for PCC was that the producer automatically infers annotation for properties like well typedness, correct read/writes and automatically generates the proof for their correctness using the certifying compiler. However, such properties guarantee that a program executes correctly w.r.t. to the semantics of the abstract machine, but cannot guarantee if a program executes correctly w.r.t to a functional specification. The verification condition generator presented in the following is tailored to deal with functional properties.

4.2 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 $\mathtt{m}.body$ is denoted with $\mathtt{m}.body[k]$. 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 $\mathtt{m}.body[0]$. The array of bytecode instructions of a method $\mathtt{m}$ determine an oriented graph $G(V, \rightarrow)$ in which the vertices are the instructions of the method body, i.e.

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

The set of edges $\rightarrow$ is a relation between the vertices elements

$$\rightarrow : V * V$$

and is defined in the following way:

$$(\mathtt{m}.body[j], \mathtt{m}.body[k]) \in \rightarrow \iff \begin{array}{l} \mathtt{m}.body[j] \neq \text{ return } \wedge (\\ \mathtt{m}.body[j] = \text{ if\_cond } k \vee \\ \mathtt{m}.body[j] = \text{ goto } k \vee \\ \mathtt{m}.body[j] \neq \text{ goto } \Rightarrow k = j + 1) \end{array}$$

i.e. there is an edge between two vertices $\mathtt{m}.body[j]$ and $\mathtt{m}.body[k]$ if they may execute immediately one after another. In the following, we will rather use the infix notation $\mathtt{m}.body[j] \rightarrow \mathtt{m}.body[k]$.

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 4.2.1 (Backedge Definition) *Let's have the method* $\mathtt{m}$ *with body* $\mathtt{m}.body$ *which determine the control flow graph* $G(V, \rightarrow)$. *G is such that the vertice* $\mathtt{m}.body[0]$ *does not have predecessors and any path that reaches any other instruction in the graph passes through* $\mathtt{m}.body[0]$. *In such a graph G, we say that* $instr_{loopEntry}$ *is a loop entry instruction and* $instr_f$ *is a loop end instruction of the same loop if the following conditions hold:*

- *every path in the control flow graph starting at the entry point* $\mathtt{m}.body[0]$ *that reaches* $instr_f$, *passes before reaching* $instr_f$ *through* $instr_{loopEntry}$
- *there is a path in which* $instr_{loopEntry}$ *is executed immediately after the execution of* $instr_f$ *(* $instr_f \rightarrow instr_{loopEntry}$*)*

We denote the execution relation between $instr_f$ *and* $instr_{loopEntry}$ *with* $instr_f \rightarrow^l instr_{loopEntry}$ *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 (a standard notion denoting a sequence of instructions where only the last one may be a jump and the first may be a target of a jump) the execution relation between the intstructions in a block being evident.

The next lemma states a property about execution paths in a control flow graph that contains backedges. This lemma will be used in the proof of correctness of our calculus in section 5.3.

Lemma 4.2.1 *Let's have a control flow graph with an entry point instruction* $\mathtt{m}.body[0]$ *and two instructions* $instr_{loopEntry}$ *and* $instr_f$ *such that* $instr_f \rightarrow^l instr_{loopEntry}$. *If there exists an execution path* P *from* $\mathtt{m}.body[0]$ *to* $instr_f$*:* $P = \mathtt{m}.body[0] \rightarrow^+ instr_f$ *then there exists a subpath which is a prefix of* P $subP = \mathtt{m}.body[0] \rightarrow^* instr_{loopEntry}$ *such that* $instr_f \notin subP$

4.3 Extending method declarations with specification

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

$$Method = \left\{ \begin{array}{ll} Name & : MethodName \\ retType & : JType \\ args & : (name * JType)[] \\ nArgs & : nat \\ body & : \mathrm{I}[] \\ entryPnt & : \mathrm{I} \\ excHndlS & : ExcHandler[] \\ exceptions & : Class_{exc}[] \\ pre & : \mathcal{F}^{bc} \\ modif & : locations[\,] \\ excPost & : ExcType \longrightarrow \mathcal{F}^{bc} \\ normalPost & : \mathcal{F}^{bc} \\ loopSpecS & : LoopSpec[\,] \end{array} \right\}$$

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

- $\mathtt{m}.pre$ gives the precondition of the method, i.e. the predicate that must hold whenever $\mathtt{m}$ is called
- $\mathtt{m}.normalPost$ is the postcondition of the method in case $\mathtt{m}$terminates normally
- $\mathtt{m}.modif$ is also called the method frame condition. It is a list of locations that the method may modify during its execution

- $\mathtt{m}.excPost$ is a function from exception types to formulas which returns the predicate $\mathtt{m}.excPost(\mathtt{Exc})$ that must hold in the method's poststate if the method $\mathtt{m}$ terminates by throwing an exception of type $\mathtt{Exc}$

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

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

$$LoopSpec = \left\{ \begin{array}{ll} pos & : nat \\ \textbf{loopInv} & : \mathcal{F}^{bc} \\ \textbf{loopModif} & : locations[\,] \end{array} \right\}$$

define modifies locations in the grammar

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

- the field $\mathtt{m}.loopSpecS[k].pos$ is a valid index in the body of $\mathtt{m}$: $0 \leq \mathtt{m}.loopSpecS[k].pos < \mathtt{m}.body.length$ and is a loop entry instruction in the sense of Def.4.2.1

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

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

4.4 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

4.4.1 Intermediate predicates

First, we define the function *inter* that for two instructions that may execute one after another in a control graph determines the predicate $inter(k, j)$ which must hold in between them. This predicate depends on the execution relation between the two instructions $instr_k$ and $instr_j$.

Definition 4.4.1 (intermediate predicate between two instructions) *Assume that $instr_k \rightarrow instr_j$. The predicate $inter(k, j)$ must hold after the execution of $instr_k$ and before the execution of $instr_j$ and is defined as follows:*

- *if* $instr_j$ *is a loop entry instruction and* $instr_k \rightarrow^l instr_j$ *then the corresponding loop invariant* I^{instr_j} *must hold*

$$inter(k, j) \equiv \textbf{loopInv}$$

- *else if* $instr_j$ *is a loop entry then the corresponding loop invariant* **loopInv** *must hold before* $instr_j$ *is executed, i.e. after the execution of* $instr_k$*. We also require that* I^{instr_j} *implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations* $\textbf{loopModif} = \{mod_i \mid i = 1..s\}$ *that may be modified in the loop*

$$inter(k, j) \equiv \textbf{loopInv} \ \wedge \ \forall_{i=1..s} mod_i.(\textbf{loopInv} \Rightarrow \ wp(instr_j, j, \psi^{exc},))$$

- *else*

$$inter(k, j) \equiv \ wp(instr_j, j, \psi^{exc}, normalPost)$$

4.4.2 Weakest precondition in the presence of exceptions

For every method `m` we define also the function $\texttt{m}.\psi^{exc}$ with signature:

$$\texttt{m}.\psi^{exc} : int \longrightarrow ExcType \longrightarrow \mathcal{F}^{bc}$$

$\texttt{m}.\psi^{exc}(\ i,\ \texttt{Exc})$ returns the predicate that must hold in the poststate of the instruction at index i if this instruction throws an exception of type `Exc`. Thus, $\texttt{m}.\psi^{exc}$ will either return the weakest precondition of the exception handler that protects the instruction from the thrown exception `Exc`, if such an exception handler exists. Otherwise, the function will return the exceptional postcondition if method `m` terminates on an exception `Exc`. Note that in the second case the specification variable $\backslash\textbf{EXC}$, which stands for the thrown object in exceptional postocnditions, is substituted with the reference to the thrown object.

The formal definition is given hereafter.

Definition 4.4.2 (Postcondition in case of throwing an exception) *The function application* $\texttt{m}.\psi^{exc}(\ i,\ \texttt{Exc})$ *is defined as follows:*

- *if* $findExcHandler(\ \texttt{Exc}, i, \texttt{m}.excH) = handlerPc$ *then*

$$\begin{aligned}&\texttt{m}.\psi^{exc}(\ i,\ \texttt{Exc}) = \\ &\ wp(\texttt{m}.body[handlerPc], handlerPc, \texttt{m}.\psi^{exc}, \texttt{m}.normalPost) \\ &\ [\textbf{cntr} \leftarrow 0] \\ &\ [\textbf{st}(\textbf{cntr}) \leftarrow \ \texttt{ref}\ ] \\ &\ [f \leftarrow f[\oplus\ \texttt{ref}\ \rightarrow defVal(f.Type)]]_{\forall f:Field,\ \mathsf{subtype}\ (f.declaredIn,\ \texttt{Exc})}\end{aligned}$$

- *else if findExcHandler*($\texttt{Exc}, i, \texttt{m}.excH) = \perp$ *then*

  $\texttt{m}.\psi^{exc}(\ i,\ \texttt{Exc}) =$
  $\texttt{m}.excPost(\ \texttt{Exc})$
  $[f \leftarrow f[\oplus\ \texttt{ref}\ \rightarrow defVal(f.Type)]]_{\forall f:Field,\mathsf{subtype}\ (f.declaredIn,\ \texttt{Exc})}$
  $[\backslash\mathbf{EXC} \leftarrow\ \texttt{ref}\ ]$

4.4.3 Rules for single instruction

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:

$$wp:\ \mathrm{I} \longrightarrow int \longrightarrow (int \longrightarrow ExcType \longrightarrow \mathcal{F}^{bc}) \longrightarrow \mathcal{F}^{bc} \longrightarrow \mathcal{F}^{bc}$$

The function *wp* takes four arguments : the kind of the instruction *ins*, the position of the instruction in the body of the method m to which the instruction belongs *pos*, an exceptional function $\texttt{m}.\psi^{exc}$ as described in the previous subsection 4.4.2, the postcondition predicate $\texttt{m}.normalPost$. The function will return a predicate $wp(ins, pos, \texttt{m}.\psi^{exc}, \texttt{m}.normalPost)$ such that if it holds in the prestate of the method m and if the m terminates normally then the normal postcondition $\texttt{m}.normalPost$ holds when m terminates execution, otherwise if m terminates on an exception Exc the exceptional postcondition $\texttt{m}.\psi^{exc}(\ \texttt{Exc})$ 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 cannot guarantee program termination.

Note that in what follows we ignore the Java virtual machine errors. Thus, we deal only with user defined exceptions and `JavaRuntimeException` subclasses.

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

- Control transfer instructions

  1. unconditional jumps

     $$wp(\ \text{goto}\ \ n, i, \psi^{exc}, normalPost) =\ \ inter(i, n)$$

     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

     $$\begin{array}{l} wp(\ \text{if\_cond}\ \ n, i, \psi^{exc}, normalPost) = \\ \quad \texttt{cond}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow \\ \qquad inter(i, n)[\mathbf{cntr} \leftarrow \mathbf{cntr}-2] \\ \wedge \\ not(\ \ \texttt{cond})(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1))) \Rightarrow \\ \qquad inter(i, i+1)[\mathbf{cntr} \leftarrow \mathbf{cntr}-2] \end{array}$$

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 $inter(i, n)[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2]$ 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 $inter(i, i+1)[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2]$.

3. return

$$wp(\text{ return },i,\psi^{exc},normalPost) = normalPost[\backslash\mathbf{result} \leftarrow \mathbf{st}(\mathbf{cntr})]$$

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

$$\begin{array}{ll} wp(\text{ load j},i,\psi^{exc},normalPost) = & \\ inter(i,i+1) & [\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ & [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(j)] \end{array}$$

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 $inter(i, i+1)$ is equal to $\mathbf{st}(counter) == 3$ then we get that the precondition of instruction is $\mathbf{reg}(j) == 3$:

$$\left\{\begin{array}{ll} (\mathbf{st}(\mathbf{cntr}) == 3) & [\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ & [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(j)] \\ \equiv & \\ \mathbf{reg}(j) == 3 & \end{array}\right\}$$
$i:$ load j
$\{\mathbf{st}(\mathbf{cntr}) == 3\}$
$i+1: \ldots$

2. store the stack top element in a local variable

$$\begin{array}{ll} wp(\text{ store j},i,\psi^{exc},normalPost) = & \\ inter(i,i+1) & [\mathbf{cntr} \leftarrow \mathbf{cntr}-1] \\ & [\mathbf{reg}(j) \leftarrow \mathbf{st}(\mathbf{cntr})] \end{array}$$

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

$$\begin{array}{ll} wp(\text{ push } \mathrm{j}, i, \psi^{exc}, normalPost) = & \\ inter(i, i+1) & [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ & [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathrm{j}\ ] \end{array}$$

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

$$\begin{array}{l} wp(\text{ iinc } \mathrm{j}, i, \psi^{exc}, normalPost) = \\ inter(i, i+1)[\mathbf{reg}(j) \leftarrow \mathbf{reg}(j) + 1] \end{array}$$

- arithmetic instructions

1. instructions that cannot cause exception throwing (`arithOp` = add , sub , mult , and , or , xor , ishr , ishl ,)

$$\begin{array}{ll} wp(\text{ arith\_op}, i, \psi^{exc}, normalPost) = & \\ inter(i, i+1) & [\mathbf{cntr} \leftarrow \mathbf{cntr} - 1] \\ & [\mathbf{st}(\mathbf{cntr}-1) \leftarrow \mathbf{st}(\mathbf{cntr}) \texttt{op}\ \mathbf{st}(\mathbf{cntr}-1)] \end{array}$$

2. instructions that may throw exceptions (`arithOp` = rem , div)

$$\begin{array}{l} wp(\text{ arithOp }, i, \psi^{exc}, normalPost) = \\ \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\ \qquad \begin{array}{ll} inter(i, i+1) & [\mathbf{cntr} \leftarrow \mathbf{cntr} - 1] \\ & [\mathbf{st}(\mathbf{cntr}-1) \leftarrow \mathbf{st}(\mathbf{cntr})\ \texttt{op}\ \mathbf{st}(\mathbf{cntr}-1)] \end{array} \\ \\ \wedge \\ \\ \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \psi^{exc}(i,\ \texttt{NullPntrExc}) \end{array}$$

- object creation and manipulation

1. create a new object

$$\begin{array}{ll} wp(\text{ new } C, i, \psi^{exc}, normalPost) = & \\ & [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ inter(i, i+1) & [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \texttt{ref}\ ] \\ & [f \leftarrow f[\oplus\ \texttt{ref}\ \rightarrow defVal(f.Type)]]_{\forall f:Field.\mathsf{subtype}\ (f.declaredIn, C)} \end{array}$$

The postcondition of the instruction new is the intermediate predicate $inter(i, i+1)$. Then the precondition is the same predicate but in which the stack counter is incremented, the stack top is substituted by a fresh reference `ref` and the fields for the `ref` are initialized with their default values

2. array creation

$$
\begin{array}{l}
wp(\text{ newarray } \mathtt{T}, i, \psi^{exc}, normalPost) = \\
\mathbf{st}(\mathbf{cntr}) \geq 0 \Rightarrow \\
\quad inter(i, i+1) \\
\quad [\mathbf{st}(\mathbf{cntr}) \leftarrow \texttt{ ref }] \\
\quad [\text{arrAccess} \leftarrow \text{arrAccess}[\oplus(\texttt{ ref }, j) \rightarrow defVal(\mathtt{T})]]_{\forall j, 0 \leq j < \mathbf{st}(\mathbf{cntr})} \\
\quad [\text{arrLength} \leftarrow \text{arrLength}[\oplus \texttt{ ref } \rightarrow \mathbf{st}(\mathbf{cntr})]] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) < 0 \Rightarrow \\
\quad \psi^{exc}(i, \texttt{ NegArrSizeExc})
\end{array}
$$

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 **st**(**cntr**) is smaller than 0. In this case, function ψ^{exc} will search for the corresponding postcondition of the instruction at position i and the exception `NegArrSizeExc`

3. field access

$$
\begin{array}{l}
wp(\text{ getfield } f, i, \psi^{exc}, normalPost) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\quad inter(i, i+1)[\mathbf{st}(\mathbf{cntr}) \leftarrow f(\mathbf{st}(\mathbf{cntr}))] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\quad \psi^{exc}(i, \texttt{ NullPntrExc})
\end{array}
$$

The instruction for accessing a field value takes as postcondition the predicate that must hold between it and its next instruction $inter(i, i+1)$. 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 ψ^{exc} for position i in the bytecode and exception `NullPntrExc`

4. field update

$$
\begin{array}{l}
wp(\text{ putfield } f, i, \psi^{exc}, normalPost) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad inter(i, i+1)\ \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]\end{array} \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \psi^{exc}(i,\ \texttt{NullPntrExc})
\end{array}
$$

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 $inter(i, i+1)$ 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 and its postcondition be $inter(i, i+1) \equiv f(\mathbf{reg}(1)) \neq \mathbf{null}$. The function wp will return :

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

After applying the substitution following the rules described in Section 5.1, we obtain that the precondition is

$$
f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})](\mathbf{reg}(1)) \neq \mathbf{null}
$$

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

$$
\left\{
\begin{array}{l}
(f(\mathbf{reg}(1)) \neq \mathbf{null})\ \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]\end{array} \\
\equiv \\
f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})](\mathbf{reg}(1)) \neq \mathbf{null}
\end{array}
\right\}
$$

$i:$ putfield f
$\{f(\mathbf{reg}(1)) \neq \mathbf{null}\}$
$i+1 : \ldots$

5. access the length of an array

$$
\begin{array}{l}
wp(\text{ arraylength}, i, \psi^{exc}, normalPost) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad inter(i, i+1)[\mathbf{st}(\mathbf{cntr}) \leftarrow \text{arrLength}(\mathbf{st}(\mathbf{cntr}))] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\qquad \psi^{exc}(i,\ \texttt{NullPntrExc})
\end{array}
$$

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 $inter(i, i+1)$ 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

$$
\begin{array}{l}
wp(\text{ checkcast } C, i, \psi^{exc}, normalPost) = \\
\backslash\mathbf{typeof(st(cntr))} <: C \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\quad inter(i, i+1) \\
\wedge \\
not(\backslash\mathbf{typeof(st(cntr))} <: C) \Rightarrow \\
\quad \psi^{exc}(i, \texttt{CastExc})
\end{array}
$$

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 $\mathbf{st}(\mathbf{cntr}\ )$ is of type which is a subtype of class C or is **null**then the predicate $inter(i, i+1)$ holds in the prestate. Otherwise, if $\mathbf{st}(\mathbf{cntr}\ )$ is not of type which is a subtype of class C, the instruction terminates on `CastExc` and the predicate returned by ψ^{exc} for the position i and exception `CastExc` must hold

7. instanceof

$$
\begin{array}{l}
wp(\text{ instanceof}C, i, \psi^{exc}, normalPost) = \\
\backslash\mathbf{typeof(st(cntr))} <: C \Rightarrow \\
\quad inter(i, i+1)[\mathbf{st}(\mathbf{cntr}) \leftarrow 1] \\
\wedge \\
not(\backslash\mathbf{typeof(st(cntr))} <: C) \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\quad inter(i, i+1)[\mathbf{st}(\mathbf{cntr}) \leftarrow 0]
\end{array}
$$

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).

$$
\begin{array}{l}
wp(\text{ invoke } \mathtt{m}, i, \psi^{exc}, normalPost) = \\
\mathtt{m}.pre[\mathbf{reg}(s) \leftarrow \mathbf{st}(\mathbf{cntr} + s - \mathtt{m}.nArgs)]_{s=0}^{\mathtt{m}.nArgs} \\
\wedge \\
\forall mod, (mod \in \text{ m } .modif), \forall freshVar(\\
\quad \mathtt{m}.normalPost \begin{array}{l}[\backslash\mathbf{result} \leftarrow freshVar] \\ [\mathbf{reg}(s) \leftarrow \mathbf{st}(\mathbf{cntr} + s - numArgs(m))]_{s=0}^{\mathtt{m}.nArgs}\end{array} \Rightarrow \\
\qquad inter(i, i+1) \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} - \mathtt{m}.nArgs] \\ [\mathbf{st}(\mathbf{cntr} - \mathtt{m}.nArgs) \leftarrow freshVar]\end{array} \ ) \\
\wedge_{j=0}^{\mathtt{m}.exceptions.length-1} \\
(\mathtt{m}.excPost(\mathtt{m}.exceptions[j]) \Rightarrow \psi^{exc}(i, \mathtt{m}.exceptions[j]))
\end{array}
$$

This rule is may be the more complicated one. First, note that because we are following a contract based approach the caller, i.e. the current method must establish several facts. First, we require that the precondition m.*pre* of the invoked method m holds where the formal parameters are correctly initialized with the first m.*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 m.*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 $freshVar$ and correctly initialized formal parameters is true we want to establish that the predicate *inter*(i, i+1) holds . This implication is quantified over the locations m .*modif* that a method may modify and the variable $freshVar$ which stands for the result that the invoked method m returns.

The third part of the rule deals with the exceptional termination of the method invokation. In this case, if the invoked method terminates on any exception which belongs to the array of exceptions m.*exceptions* that m may throw, its exceptional postcondition for the corresponding exception must guarantee the predicate returned by the function ψ^{exc} of the current method

- throw exception instruction

$$
\begin{aligned}
&wp(\text{ athrow },i,\psi^{exc},normalPost) = \\
&\mathbf{st(cntr)} \neq \mathbf{null} \Rightarrow \psi^{exc}(i,\backslash\mathbf{typeof(st(cntr))}) \\
&\wedge \\
&\mathbf{st(cntr)} = \mathbf{null} \Rightarrow \psi^{exc}(i,\ \texttt{NullPntrExc})
\end{aligned}
$$

The thrown object is on the top of the stack **st**(**cntr**). If the thrown object is not **null**, then the function ψ^{exc} will return the precondition of the handler that may handle exceptions of type **typeof(st(cntr))** on position i in the bytecode, otherwise if such a handler does not exist ψ^{exc} will return the exceptional postcondition that must hold if the current method throws an exception **typeof(st(cntr))**. If the stack top object **st**(**cntr**) is **null**, then the instruction athrow will terminate on an exception NullPntrExcwhere the predicate returned by the function ψ^{exc} must hold.

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:

$$\texttt{m}.pre \Rightarrow\ wp(\text{ m }.body[0], 0, \texttt{m}.\psi^{exc}, \texttt{m}.normalPost)$$

4.5 Example

.

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.

Another point which deserves our attention, is that we will prove the partial correctness of our methodology, i.e. we suppose that our programs always terminate.

In the following, we will proceed with a section which establishes how we define substitution. In section 5.2, we give a meaning of formulas from our assertion language in a state. The last section 5.3 skatches the proof of correctness of the verification condition generator.

5.1 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. This is done by establishing the substitution rule

for field access as follows:

$$f(o)[\mathcal{E}_1 \leftarrow \mathcal{E}_2] = f[\mathcal{E}_1 \leftarrow \mathcal{E}_2](o[\mathcal{E}_1 \leftarrow \mathcal{E}_2])$$

In the next, we define a substitution over field function objects:

$$f[\mathcal{E}_1 \leftarrow \mathcal{E}_2] = \begin{cases} f & if\ \mathcal{E}_1 \neq f \\ \mathcal{E}_2 & if\ \mathcal{E}_1 = f \wedge \\ & \mathcal{E}_2 = f[\oplus\ \mathtt{r}\ \longrightarrow\ \mathtt{v}\ ] \end{cases}$$

$$f[\oplus\ \mathtt{r}\ \rightarrow\ \mathtt{v}\ ][\mathcal{E}_1 \leftarrow \mathcal{E}_2] = \begin{cases} f[\oplus\ \mathtt{r'}\ [\mathcal{E}_1 \leftarrow \mathcal{E}_2] \rightarrow\ \mathtt{v'}\ [\mathcal{E}_1 \leftarrow \mathcal{E}_2]] & if\ \mathcal{E}_1 \neq f \\ f\ \begin{matrix}[\oplus\ \mathtt{r'}[\mathcal{E}_1 \leftarrow \mathcal{E}_2] \longrightarrow\ \mathtt{v'}[\mathcal{E}_1 \leftarrow \mathcal{E}_2]] \\ [\oplus\ \mathtt{r}\ \longrightarrow\ \mathtt{v}\ ]\end{matrix} & if\ \mathcal{E}_1 = f \wedge \\ & \mathcal{E}_2 = f[\oplus\ \mathtt{r}\ \longrightarrow\ \mathtt{v}\ ] \end{cases}$$

For example, consider the following substitution

$$f(o)[f \leftarrow f[\oplus e \rightarrow v]]$$

This results in the new expression :

$$f[\oplus e \rightarrow v](o[f \leftarrow f[\oplus e \rightarrow v]])$$

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

5.2 Interpretation of assertions in a state

We discuss the evaluation of expressions and interpretation of predicates in a particular program state configuration. Thus, we need a function for expression evaluation, a function for interpretation of predicates The function *eval* will evaluate expressions in a given state:

$$eval : \mathcal{E} \rightarrow K \rightarrow \text{Values}$$

Definition 5.2.1 (Evaluation of expressions) *The evaluation in a state* $s =< \text{H}, \text{Cntr}, \text{St}, \text{Reg}, \text{Pc} >$ *or* $s =< \text{H}, \text{Reg}, Final >^{final}$ *of an expression* $\mathcal{E}$ *is denoted with* $eval(\mathcal{E}, s)$ *and is defined inductively on the grammar of expressions as follows:*

$$
\begin{array}{l}
eval(v, s) = v \\
where\ v \in \quad \text{intliteral} \quad \vee \quad v \in \text{RefVal}
\end{array}
$$

$$
\begin{array}{l}
eval(f(\mathcal{E}), s) = \\
= \text{H}(f)(\ eval(\mathcal{E}, s))
\end{array}
$$

$$
\begin{array}{l}
eval(f[\oplus \mathcal{E}_1 \rightarrow \mathcal{E}_2](\mathcal{E}_3), s) = \\
= \text{H}[\oplus f \rightarrow f[\oplus\ eval(\mathcal{E}_1, s) \rightarrow \ \ eval(\mathcal{E}_2, s)]](f)(\ eval(\mathcal{E}_3, s))
\end{array}
$$

$$
\begin{array}{l}
eval(\mathbf{arrayAccess}(\mathcal{E}_1, \mathcal{E}_2), s) = \\
= \text{H}(\ eval(\mathcal{E}_1, s),\ \ eval(\mathcal{E}_2, s))
\end{array}
$$

$$
\begin{array}{l}
eval(\text{arrAccess}[\oplus(\mathcal{E}_1, \mathcal{E}_2) \rightarrow \mathcal{E}_3](\mathcal{E}_4, \mathcal{E}_5), s) = \\
= \text{H}[\oplus(\ eval(\mathcal{E}_1, s),\ \ eval(\mathcal{E}_2, s)) \rightarrow \ \ eval(\mathcal{E}_3, s)](\ eval(\mathcal{E}_4, s),\ \ eval(\mathcal{E}_5, s))
\end{array}
$$

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

$$
eval(\mathcal{E}_1\ op\ \mathcal{E}_2, s) = \ \ eval(\mathcal{E}_1, s) op\ eval(\mathcal{E}_2, s)
$$

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

$$
\begin{array}{l}
eval(\mathbf{cntr}, s) = \text{Cntr} \\
\\
eval(\mathbf{st}(\mathcal{E}), s) = \text{St}(\ eval(\mathcal{E}, s))
\end{array}
$$

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

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

The relation $\vDash$ that we define next, gives a meaning to the formulas from our assertion language $\mathcal{F}^{bc}$.

Definition 5.2.2 (Interpretation of predicates) *The interpretation* $s \vDash \mathcal{F}^{bc}$

of a predicate $\mathcal{F}^{bc}$ *in a state configuration* s *is defined inductively as follows:*

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

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

$s \vDash \mathcal{F}_1^{bc} \wedge \mathcal{F}_2^{bc}$ *iff* $s \vDash \mathcal{F}_1^{bc}$ *and* $s \vDash \mathcal{F}_2^{bc}$

$s \vDash \mathcal{F}_1^{bc} \vee \mathcal{F}_2^{bc}$ *iff* $s \vDash \mathcal{F}_1^{bc}$ *or* $s \vDash \mathcal{F}_2^{bc}$

$s \vDash \mathcal{F}_1^{bc} \Rightarrow \mathcal{F}_2^{bc}$ *iff if* $s \vDash \mathcal{F}_1^{bc}$ *then* $s \vDash \mathcal{F}_2^{bc}$

$s \vDash \forall x : T.\mathcal{F}^{bc}(x)$ *iff forall value* $\mathbf{v}$ *of type* T $s \vDash \mathcal{F}^{bc}(\ \mathbf{v})$

$s \vDash \exists x : T.\mathcal{F}^{bc}(x)$ *iff a value* $\mathbf{v}$ *of type* T *exists such that* $s \vDash \mathcal{F}^{bc}(\ \mathbf{v})$

$s \vDash \mathcal{E}_1\ \mathcal{R}\ \mathcal{E}_2$ *iff* $eval(\mathcal{E}_1, s)$ $eval(\mathcal{R}, s)$ $eval(\mathcal{E}_2, s)$ *evaluates to true*

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

Lemma 5.2.1 (Update of the heap) *For any expressions* $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ *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* $s_2 =< \mathrm{H}[\oplus f \rightarrow f[\oplus\ eval(\mathcal{E}_2, s_1) \rightarrow\ eval(\mathcal{E}_3, s_1)]], \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *the following holds*

- $eval(\mathcal{E}_1[f \leftarrow f[\oplus \mathcal{E}_2 \rightarrow \mathcal{E}_3]], s_1) =\ eval(\mathcal{E}_1, s_2)$
- $s_1 \vDash \psi[f \leftarrow f[\oplus \mathcal{E}_2 \rightarrow \mathcal{E}_3]] \iff s_2 \vDash \psi$

Lemma 5.2.2 (Update of the heap with a newly allocated object) *For any expressions* $\mathcal{E}_1$ *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* $s_2 =< \mathrm{H}', \mathrm{Cntr}, \mathrm{St}[\oplus \mathrm{Cntr} \rightarrow$ `ref` $], \mathrm{Reg}, \mathrm{Pc} >$ *where* $\mathrm{newRef}(\mathrm{H}, C) = (\mathrm{H}',$ `ref` $)$ *the following holds*

- $$eval(\mathcal{E}_1\ \begin{matrix}[\mathbf{st}(\mathbf{cntr}) \leftarrow\ \texttt{ref}\ ] \\ [f \leftarrow f[\oplus\ \texttt{ref}\ \rightarrow defVal(f.Type)]]_{\forall f:Field, \mathsf{subtype}\ (f.declaredIn, C)}\end{matrix}\ , s_1) = eval(\mathcal{E}_1, s_2)$$
- $$s_1 \vDash \psi\ \begin{matrix}[\mathcal{E}_2 \leftarrow\ \texttt{ref}\ ] \\ [f \leftarrow f[\oplus\ \texttt{ref}\ \rightarrow defVal(f.Type)]]_{\forall f:Field, \mathsf{subtype}\ (f.declaredIn, C)}\end{matrix} \iff s_2 \vDash \psi$$

Lemma 5.2.3 (Update the stack) *For any expressions $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ 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 $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}[\oplus\ eval(\mathcal{E}_2, s_1) \rightarrow\ eval(\mathcal{E}_3, s_1)], \mathrm{Reg}, \mathrm{Pc} >$ then the following holds:*

- $eval(\mathcal{E}_1[\mathbf{st}(\mathcal{E}_2) \leftarrow \mathcal{E}_3], s_1) =\ eval(\mathcal{E}_1, s_2)$
- $s_1 \vDash \psi[\mathbf{st}(\mathcal{E}_2) \leftarrow \mathcal{E}_3] \iff s_2 \vDash \psi$

Lemma 5.2.4 (Update the stack counter) *For any expressions $\mathcal{E}_1, \mathcal{E}_2$ 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 $s_2 =< \mathrm{H},\ eval(\mathcal{E}_2, s_1), \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ then the following holds:*

- $eval(\mathcal{E}_1[\mathbf{cntr} \leftarrow \mathcal{E}_2], s_1) =\ eval(\mathcal{E}_1, s_2)$
- $s_1 \vDash \psi[\mathbf{cntr} \leftarrow \mathcal{E}_2] \iff s_2 \vDash \psi$

Lemma 5.2.5 (Update a local variable) *For any expressions $\mathcal{E}_1, \mathcal{E}_2$ 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 $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}[\oplus i \rightarrow\ eval(\mathcal{E}_2, s_1)], \mathrm{Pc} >$ then the following holds:*

- $eval(\mathcal{E}_1[\mathbf{reg}(i) \leftarrow \mathcal{E}_2], s_1) =\ eval(\mathcal{E}_1, s_2)$
- $s_1 \vDash \psi[\mathbf{reg}(i) \leftarrow \mathcal{E}_2] \iff s_2 \vDash \psi$

Lemma 5.2.6 (Return value property) *For any expression $\mathcal{E}_1$ and $\mathcal{E}_2$, for any two states s_1 and s_2 such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, \mathrm{Reg},\ eval(\mathcal{E}_2, s_1) >^{norm}$ then the following holds:*

- $eval(\mathcal{E}_1[\backslash\mathbf{result} \leftarrow \mathcal{E}_2], s_1) =\ eval(\mathcal{E}_1, s_2)$
- $s_1 \vDash \psi[\backslash\mathbf{result} \leftarrow \mathcal{E}_2] \iff s_2 \vDash \psi$

The next definition defines a particular set of assertion formulas.

Definition 5.2.3 *If an assertion formula $f \in \mathcal{F}^{bc}$ holds in all states we say that this is a valid formula and we note it with : $\vDash f$*

5.3 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 5.3.1 (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.excPost`*, we say that* `m` *respects its specification if for every two states $state_0$ and $state_1$ such that :*

- $\mathtt{m} : state_0 \rightarrow state_1$
- $state_0 \vDash \mathtt{m}.pre$

Then if m *terminates normally then the normal postcondition holds in the final state* $state_1$*:* $state_1 \vDash \mathtt{m}.normalPost$*. Otherwise, if* m *terminates on an exception* Exc *the exceptional postcondition holds in the poststate* $state_1$ $state_1 \vDash \mathtt{m}.\psi^{exc}(\ \mathtt{Exc})$

The next issue that is important for understanding our approach is that we follow the design by contract paradigm [4]. 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 5.3.1.

Once generated the verification conditions are proved with the first-order predicate logic rules. We denote that a formula f has a proof in the empty context like this

$$\vdash f$$

The following lemma states that the proof rules preserve the meaning of predicates as defined in the previous section 5.2.

Lemma 5.3.1 (Provability implies validity)

$$\forall f, f \in \mathcal{F}^{bc}, \vdash f \ \Rightarrow \vDash f$$

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 1 (Single execution step correctness) *For every instruction* $instr_n$ *belonging to the bytecode of method* m *if the following conditions hold:*

- $instr_s : state_0 \rightarrow state_1$
- $state_0 \vDash \ wp(instr_s, s, \psi^{exc}, normalPost)$
- $instr_k = next(instr_s)$
- $\forall\ \mathtt{n} : Method.\ \mathtt{n} \neq \mathtt{m}$ *n is correct w.r.t. its specification*

then $state_1 \vDash \ inter(s, k)$ *holds*

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 2 (Progress) *Assume we have a method* $\mathtt{m}$ *with normal postcondition* $\mathtt{m}.normalPost$ *and exception function* $\mathtt{m}.\psi^{exc}$. *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* $< \mathrm{H}_0, \mathrm{Cntr}_0, \mathrm{St}_0, \mathrm{Reg}_0, \mathrm{Pc}_0 > \rightarrow^n < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >$. *Assume that*
$\forall i(\ 0 \leq i \leq n)$ $state_i \vDash\ wp(instr_{\mathrm{Pc}_i}, \mathrm{Pc}_i, \psi^{exc},)$ *holds then*

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

2. *if* $instr_{\mathrm{Pc}_n}$ *throws a not handled exception of type* Exc $< \mathrm{H}_{n+1},\ \mathtt{ref}_{\mathrm{Exc}}, \mathrm{Reg}_n >^{exc} \vDash \mathtt{m}.\psi^{exc}(\mathrm{Pc}_n, E)$ *holds where* $\mathrm{newRef}(\mathrm{H}_n, \mathrm{Exc}) = (\mathrm{H}_{n+1},\ \mathtt{ref}_{\mathrm{Exc}})$.

3. *else exists a state* $state_{n+1}$ *such that another execution step can be done* $state_n \rightarrow state_{n+1}$ *and* $state_{n+1} \vDash\ wp(instr_{\mathrm{Pc}_{n+1}}, \mathrm{Pc}_{n+1}, \psi^{exc}, normalPost)$ *holds*

Proof : The proof is done by induction on the length of the execution path and 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. assume that the $instr_{\mathrm{Pc}_n}$ is not an end loop and that $instr_{\mathrm{Pc}_{n+1}}$ is not a loop entry (the next execution step is neither a loop iteration, neither a loop initial entrance). We consider two cases: the case when the next execution step doesnot do an iteration (the execution step is not $\rightarrow_l$) 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 $\rightarrow_l$).

   (a) *Let's* $instr_{\mathrm{Pc}_n} =$ return
   { *by initial hypothesis* }

   $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash\ wp(\text{ return }, \mathrm{Pc}_n, \psi^{exc}, normalPost)$

   { *by definition the weakest precondition for* return }

   $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash \mathtt{m}.normalPost[\backslash \mathbf{result} \leftarrow \mathbf{st}(\mathrm{cntr})]$

   { *by the substitution property 5.2.6* }

$\Longleftrightarrow$

$< \mathrm{H}_n,\ \mathit{eval}(\mathbf{st}(\mathrm{cntr}), < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >), \mathrm{Reg}_n >^{norm} \vDash \mathit{normalPost}$

$\Longleftrightarrow$

$< \mathrm{H}_n, \mathrm{Reg}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm} \vDash \mathit{normalPost}$

(b) *Let's* $\mathit{instr}_{\mathrm{Pc}_n} = \text{ load } i$

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash\ \mathit{wp}(\text{ load } i, \mathrm{Pc}_n, \psi^{exc}, \mathit{normalPost})$

{ *definition of the wp function* }

$\Longleftrightarrow$

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

{ *from defintion 4.4.1 by hypothesis the current instruction is neither a loop entry nor a loop end instruction* }

$\Longleftrightarrow$

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash$

$\mathit{wp}(\mathit{instr}_{\mathrm{Pc}_n+1}, \mathrm{Pc}_n + 1, \mathtt{m}.\psi^{exc}, \mathtt{m}.\mathit{normalPost})\ \begin{array}{l}[\mathrm{cntr} \leftarrow \mathrm{cntr} + 1] \\ [\mathbf{st}(\mathrm{cntr} + 1) \leftarrow \mathbf{reg}(i)]\end{array}$

{ *applying the substitution properties 5.2.4 and 5.2.3* }

$\Longleftrightarrow$

$< \mathrm{H}_n, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus \mathrm{Cntr}_n + 1 \rightarrow \mathrm{Reg}_n(i)], \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash$

$\mathit{wp}(\mathit{instr}_{\mathrm{Pc}_n+1}, \mathrm{Pc}_n + 1, \mathtt{m}.\psi^{exc}, \mathtt{m}.\mathit{normalPost})$

{ *from the operational semantics of the* load *instruction in section 2.7 the lemma holds in this case* }

2. $\mathit{instr}_{\mathrm{Pc}_n}$ is an end of a loop with invariant I and the next instruction to be executed is a loop entry instruction (i.e. the execution step is of kind $\rightarrow^l$ and the assertion attached to it is I). 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* }

$\mathit{state}_n \vDash\ \mathit{wp}(\mathit{instr}_{\mathrm{Pc}_n}, \mathrm{Pc}_n, \mathtt{m}.\psi^{exc}, \mathtt{m}.\mathit{normalPost})$

{ *from 1 and transformation over the above statement* }

$$(1) \quad state_{n+1} \vDash I$$

{ *the next instruction to be executed is a loop entry instruction* $instr_{loopEntry}$ *such that* $loopEntry = \mathrm{Pc}_{n+1}$ *with modified locations* $modifLoop = \{mod_i, i = 1..s\}$ *that can be modified during a loop iteration. From def. 4.2.1 and lemma 4.2.1 , we conclude that there is a prefix* $subP = entryPnt \rightarrow^* instr_{loopEntry}$ *of the current execution path which does not pass through* $instr_{\mathrm{Pc}_n}$. *We can conclude that the transition between* $prev(instr_{loopEntry})$ *and* $instr_{loopEntry}$ *in the path subP is not a backedge. By hypothesis we know that* $\forall i, 0 \leq i \geq n, state_i \vDash \ wp(instr_{\mathrm{Pc}_i}, \mathrm{Pc}_i, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$. *From def.4.4.1 and lemma 1 we conclude* }

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

$$state_k \vDash I$$

(2) $$state_k \vDash \forall mod_i, i = 1..s\Big(\begin{array}{l} I \Rightarrow \\ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost) \end{array} \Big)$$

$$state_k =^{modifLoop} state_{n+1}$$

{ *from (2)* }

(3) $state_{n+1} \vDash I \Rightarrow \ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$

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

$$\begin{array}{l} state_{n+1} \vDash \ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost) \\ \Longleftrightarrow \\ state_{n+1} \vDash \ wp(instr_{\mathrm{Pc}_{n+1}}, \mathrm{Pc}_{n+1}, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost) \end{array}$$

3. $instr_{\mathrm{Pc}_n}$ is not a loop end and the next instruction is a loop entry instruction $instr_{loopEntry}$. We look only at the case when the current instruction is a load instruction
{ *by initial hypothesis* }

$$state_n \vDash \ wp(\text{ load } i, \mathrm{Pc}_n, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$$

{ *by defintion of the wp function in section* **??** }

$$state_n \vDash \ inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1) \begin{array}{l} [\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(j)] \end{array}$$

{ *by the definition 4.4.1 for the case when the execution step is not a backedge but the target instruction is a loop entry* }

$$state_n \vDash I \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(i)]\end{matrix}$$

$$state_n \vDash \forall mod_i, i = 1..s(I \Rightarrow \ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)) \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr}+1 \\ [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{r}\end{matrix}$$

$$\Longleftrightarrow$$

$$state_n \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(i)]\end{matrix} \vDash I \wedge \forall mod_i, i = 1..s(I \Rightarrow \ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost))$$

{ *we can get from the last formulation* }

$$state_n \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(i)]\end{matrix} \vDash I$$

(1)

$$state_n \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(i)]\end{matrix} \vDash I \Rightarrow \ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$$

{ *from (1)* }

$$state_n \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1) \leftarrow \mathbf{reg}(i)]\end{matrix} \vDash \ wp(instr_{loopEntry}, loopEntry, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$$

{ *and this case holds* }
Qed.

Theorem 3 *For any method* m *if the verification condition*

$$\mathtt{m}.pre \Rightarrow \ wp(\ \mathrm{m}\ .body[0], 0, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$$

can be proven then m *is correct in the sense of the definition 5.3.1.*

Proof: the proof is done by using the upper lemma 2 and as well as the property of our deduction system in which the proof of $\mathtt{m}.pre \Rightarrow \ wp(\ \mathrm{m}\ .body[0], 0, \mathtt{m}.\psi^{exc}, \mathtt{m}.normalPost)$ is done.

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] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

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

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

[14] 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.