Bytecode Verification and its Applications

Contents

Chapter 1

Java bytecode language and its operational semantics

The purpose of this chaprter 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 [25](JVM for short). However, it abstracts from some of the JVM language features while supporting others. Thus, we concentrate on the the most important features of the JVM. In the following, we 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. In the following, we consider only non void methods. We restrict our modelization for the sake of simplicity without losing any specific feature of the Java language.
- object manipulation and creation. We support field access and update as well as object creation.
- exception throwing and handling. Our bytecode language supports runtime and programmatic exceptions as the JVM does. An example for a situation where a runtime exception is thrown is a null object dereference.

- 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` which is the root of the class hiearchy.
- the unique basic type which is supported 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 and short. However, it is true that the formalization of the long type and manipulation of long values can be more complicated because of the fact that long values are stored in two adjacent registers. For our purposes, we do not consider that long values represent an interesting case and we discard them.

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. Note that the current formalization can be extended to void methods without major difficulties.
- static fields and methods. Static data is shared between all the instances of the class where it is declared. We can extend our formalization to deal with static fields and methods, however it would have made the presentation heavier without gaining new feature from the JVM bytecode language
- static initialization. This part of the JVM is discarded as its formal understanding is difficult and complex. Static initialization is a good candidate for a future work
- 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 on bytecode level.
- interface types. These are reference types whose methods are not implemented and whose variables are constants. Such interface types are then implemented by classes and allow that a class get more than one behavior. A class may implement several interfaces. The class must give an implementation for every method declared in any interface that it implements. If a class implements an interface then every object which has as type the class is also of the interface type. Interfaces are the cause of problems in the bytecode verifier as the type hierarchy is no more a lattice in the presence of interface types and thus, the least common super type of two types is not unique. However, in the current thesis we do not deal with bytecode verification but we will be interested in the program functional behaviour. For instance, if a method overrides a method from the super class or implements a method from an interface, our objective will be to establish that the method respects the specification of the method it overrides or implements. In this sense, super classes or interfaces are treated similarly in our verification tool.

Moreover, considering interfaces would have complicated the current formalization without gaining more new features of Java. For instance, in the presence of interfaces, we should have extended the subtyping relation.

- floating point arithmetic. We omit this data in our bytecode language for the following reasons. There is no support for floating point data by automated tools. For instance, the automatic theorem prover Simplify which interfaces our verification tool lacks support for floating point data, see [22]. Although larger and more complicated than integral data, formalization of floating point arithmetic is possible. For example, the specification of IEEE for floating point arithmetic as well as a proof for its consistency is done in the interactive theorem prover Coq. However, including floating point data would not bring any interesting part of Java but would rather turn more complicated and less understandable the formalizations in this thesis.

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. This is different from most of the existing formalization of the JVM (or JVM like languages), which use usually a small step semantics. 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 1.1 is a discussion about our choice for operational semantics, subsection 1.2 is an overview of existing formalisations of the JVM semantics, subsection 1.3 gives some particular notations that will be used from now on along the thesis, subsection 1.4 introduces the structures classes, fields and methods used in the virtual machine, subsection 1.5 gives the type system which is supported by the bytecode language, subsection 1.6 introduces the notion of state configuration, subsection 1.6.1 gives the modelisation of the memory heap, subsection 1.8 gives the operational semantics of our language.

1.1 Design choices for the operational semantics

Before proceeding with the motivations for the choice of the operational semantics, we shall first look at a brief description of the semantics of the Java Virtual Machine (JVM).

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.

Most of the existing formalizations of the JVM semantics model the method frame stack and use a small step operational semantics. This approach is necessary when reasoning about the properties of the JVM or the bytecode verifier.

However the purpose of the operational semantics presented in this chapter is to give a model w.r.t.which a proof of correctness of our verification calculus will be done. Because the latter is modular and assumes termination, i.e. the verification calculus assumes the correctness and the termination of the invoked methods, we do not need a model for reasoning about the termination or the correctness of invoked methods. A big step semantics provides a suitable level of abstraction as it does not express those details. In the following, we give a short review of the formalizations of the JVM.

1.2 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 [16] by N.Freund and J.Mitchell and [28] 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 [27] 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 [20], 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 [6, 5] 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 [32] in terms of a small step operational semantics.

1.3 Notation

Here we introduce 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 $f : A \to B$. 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 $f : A_1 * ... * A_n \rightarrow B$. The set of elements which represent the domain of the function f is given by the function $\mathsf{Dom}(f)$ and the elements in its range are given by $\mathsf{Range}(f)$.

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 type $list$ is used to represent a sequence of elements. The empty list is denoted with [ ]. If it is true that the element e is in the list l, we use the notation $e \in l$. The function :: receives two arguments an element e and a list l and returns a new list $e::l$ whose head and tail are respectively e and l. The number of elements in a list l is denoted with $l.length$. The i-th element in a list l is denoted with $l[i]$. Note that the indexing in a list l starts at 0, thus the last index in l being $l.length - 1$.

1.4 Classes, Fields and Methods

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

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

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

A field object is a tuple that contains the unique field id (Name) and a field type (Type) and the class where it is declared (declaredIn):

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

From the above definition, we can notice that the field declaredIn may have a value $\bot$. This is because we model the length of a reference pointing to an array

object as an element from the set **Field** . Because the length of an array is not declared in any class, we assign to its attribute declaredIn the value $\perp$. The special field which stands for the array length (the name of the object and its field Name have the same name) is the following:

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

Note that there are other possible approaches for modeling the array length. For instance, the array length can be part of the array reference. We consider that both of the choices are equivalent. However, the current formalization follows closely our implementation of the verification condition generator which is necessary if we want to do a proof of correctness of the implementation.

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

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

We assume that for every method `m` the entrypoint is the first instruction in the list of instructions of which the method body consists, i.e. `m`.entryPnt = `m`.body[0].

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

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

We require that startPc, endPc and handlerPc fields in any exception handler attribute `m`.excHndlS for any method `m` are valid indexes in the list of instructions of the method body `m`.body:

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

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

$$\begin{array}{ll}JType & ::= \textbf{int} \mid RefType \\ RefType & ::= RefTypeCl \mid RefTypeArr \\ RefTypeCl & ::= \textbf{Class} \\ RefTypeArr & ::= JType[]\end{array}$$

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

$$\begin{array}{ll}Values & ::= i \mid RefVal \\ RefVal & ::= RefValCl \mid RefValArr \mid \textbf{null} \\ RefValArr & ::= refArr\end{array}$$

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

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

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

We define also a subtyping relation as follows:

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

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

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

1.6 State configuration

In this section, we introduce the notion of program state. A state configuration Smodels the program state in particular execution program point by specifying what is the memory heap in the state, 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:

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

The set S^{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 $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \in S^{interm}$ has the following elements:

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

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

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

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

- $< \mathrm{H}, \mathrm{Res} >^{norm} \in S^{norm}$ 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

  - Res stands for the return value of the method

- $< \mathrm{H}, \mathrm{Exc} >^{exc} \in S^{exc}$ 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
  - 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:

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

We will denote with $< \mathrm{H}, \mathit{Final} >^{final}$ for any configuration which belongs to the set S^{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.

1.6.1 Modeling the Object Heap

An important issue for the modelization of an object oriented programming language and its operational semantics is the memory heap. 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 memory space.

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 1.4) into partial functions from references (*RefType*) into values (*Values*).
- a component Arr which maps the components of arrays into their values
- a component Loc which stands for the list of references that the heap has allocated
- a component TypeOf is a partial function which maps references to their dynamic type

Formally, the data type HeapType has the following structure:

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

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

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

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

Also, we assume that the heap must contain well formed values. By this, we mean that the heap maps any field object f : **Field** which has a reference type (i.e. the component f .Type contains a reference type) into a function which may only return references which are already defined in the heap. The same condition is required for array references whose elements are references, i.e. the value of an array elements is either a reference defined in the heap or **null**. The next formalization expresses the restriction for field functions :

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

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

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

Formally, the operation is defined as follows:

$$\mathrm{allocator}(\mathrm{H}, \mathtt{ref})\ = \mathrm{H}' \iff^{def}$$

$$\begin{array}{l} \mathtt{ref} \notin \mathrm{H.Loc} \\ \mathrm{H}'.\mathsf{Loc} = \mathtt{ref}::\mathrm{H.Loc} \wedge \\ \mathrm{H.Fld} = \mathrm{H}'.\mathsf{Fld} \wedge \\ \mathrm{H.Arr} = \mathrm{H}'.\mathsf{Arr} \wedge \end{array}$$

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

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

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

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

$$\text{newRef} : \text{H} \rightarrow \mathit{RefTypeCl} \ \rightarrow \text{H} * \mathit{RefValCl}$$

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

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

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

$$\text{newArrRef} : \text{H} \rightarrow \mathit{RefTypeArr} \rightarrow \text{H} * \mathit{refArr}$$

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

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

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

$$\mathrm{H.Fld} := \mathrm{H.Fld}[\oplus f \rightarrow \mathrm{H.Fld}(f)[\oplus \mathtt{ref} \rightarrow val]]$$

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

$$\mathrm{H}[\oplus f \rightarrow f[\oplus \mathtt{ref} \rightarrow val]]$$

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

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

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

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

1.6.2 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.

1.6.3 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.

1.6.4 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.

1.7 Throwing and handling exceptions

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

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

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

The language also supports programming exceptions. Those exceptions are forced by the programmer, by a special bytecode intruction as we shall see later in the coming section. The modelization of the exception handling mechanism involves several functions. The function *getStateOnExc* deals with bytecode instructions that may throw exceptions. The function returns the state configuration after the current instruction during the execution of `m` throws an exception of type `E`. If the method `m` has an exception handler which can handle

exceptions of type E thrown at the index of the current instruction, the execution will proceed and thus, the state 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:

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

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

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

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 $\bot$:

$$findExcHandler : ExcType * nat * \mathbf{ExcHandler}[] \rightarrow nat$$

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

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

1.8 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. 1.1, we give the list of instructions that constitute our bytecode language.

I ::= if_cond
| goto
| return
| arith_op
| load
| store
| push
| pop
| dup
| iinc
| new
| newarray
| putfield
| getfield
| type_astore
| type_aload
| arraylength
| instanceof
| checkcast
| athrow
| invoke

Figure 1.1: Bytecode Language instructions

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 1.8.1 (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 1.8.2 (Transitive closure of a method state transition relation) *If the method* m *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *with* m.body*[0]*

and there exists a transitive state transition to the state $< \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$ *we denote this with:*

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

Definition 1.8.3 (Termination of method execution) *If the method* `m` *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *with* `m.body`*[0] and there is a transitive state transition to* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, k >$ *such that the instruction* `m.body`*[k] is either a* return *instruction or an instruction which terminates execution with an uncaught exception and the configuration after its execution is* $< \mathrm{H}', \mathit{Final} >^{final}$ *then we denote this with:*

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

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

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

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 1.6. Note that we do not model the method frame stack of the JVM which is not needed for our purposes.

- Control transfer instructions

1. Conditional jumps : if_cond

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

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

The condition `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

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

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

- Arithmetic operations

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

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

- Load Store instructions

1. load

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

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

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

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

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

3. getfield

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

[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`

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

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 *RefValClNullPointerExc* is pushed onto the operand stack

4. newarray `T`

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

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

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

5. type_astore

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

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

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

The three top stack elements St(Cntr), St(Cntr - 1) and St(Cntr - 2) are popped from the operand stack. The type value contained

in St(Cntr) must be assignment compatible with the type of the elements of the array reference contained in St(Cntr - 2), St(Cntr - 1) must be of type int.

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

6. type_aload

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

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

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

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 *RefTypeCl* 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

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

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

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

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

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

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

9. checkcast

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

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

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

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

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

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. If the object on the stack top is **null**, a `NullPntrExc` is thrown.

- Method Invokation. invoke [2]

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

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

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

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

1.9 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. Note that in the following, the control flow graph corresponds to a method body.

Recall from Section 1.4 that every method `m` has an array of bytecode instructions `m.body`. The $k-th$ instruction in the bytecode array `m.body` is denoted with `m.body`$[k]$. A method entry point instruction is an instruction at which an execution of a method starts. We assume that a method body has exactly one entry point and this is the first element in the method body `m.body`[0].

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

The array of bytecode instructions of a method m determine the control flow graph $G(V, \rightarrow)$ of method m 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 following definition defines the set of edges in the control flow graph.

Definition 1.9.1 (Edge in control flow graph) *The set of edges $\rightarrow$ is a relation between the vertices elements*

$$\rightarrow : V * V$$

and is defined as follows:

$$\begin{array}{l}
(\mathtt{m.body}[j], \mathtt{m.body}[k]) \in \rightarrow \\
\Longleftrightarrow \\
\mathtt{m.body}[j] \neq \text{ return } \wedge (\\
\mathtt{m.body}[j] = \text{ if_cond } k \vee \\
\mathtt{m.body}[j] = \text{ goto } k \vee \\
\mathtt{m.body}[j] \neq \text{ goto } \wedge k = j + 1 \vee \\
\mathtt{m.body}[j] = \text{ putfield} \wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ getfield} \wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ type_astore} \wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ type_astore} \wedge \mathit{findExcHandler}(\ \mathtt{ArrIndBndExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ type_aload} \wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ type_aload} \wedge \mathit{findExcHandler}(\ \mathtt{ArrIndBndExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ invoke n} \wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ invoke n} \wedge \forall \mathtt{Exc}, \exists s, \mathtt{n.exceptions}[s] = \mathtt{Exc} \wedge \\
\qquad\qquad \mathit{findExcHandler}(\mathtt{Exc}, j, \mathtt{m.excHndlS}) = k \vee \\
\mathtt{m.body}[j] = \text{ athrow} \wedge \forall \mathtt{Exc}, \mathit{findExcHandler}(\mathtt{Exc}, j, \mathtt{m.excHndlS}) = k \vee \\
)
\end{array}$$

From the Def. 1.9.1 follows that there is an edge between two vertices $\mathtt{m.body}[j]$ and $\mathtt{m.body}[k]$ if they may execute immediately one after another. We say that $\mathtt{m.body}[j]$ is a predecessor of $\mathtt{m.body}[k]$ and that $\mathtt{m.body}[k]$ is a successor of $\mathtt{m.body}[j]$. The definition states the return instruction does not have successors. If $\mathtt{m.body}[j]$ is the jump instruction if_cond k then its successors are the instruction at index k in the method body $\mathtt{m.body}[k]$ and the instruction and the instruction $\mathtt{m.body}[j+1]$. From the definition, we also get that every instruction which potentially may throw an exception of type Exc has as successor the first instruction of the exception handler that may handle the exception type Exc. For instance, a successor of the instruction putfield is the exception handler entry point which can handle the NullPntrExc exception. The possible successors of the instruction athrow are the entry point of any exception handler in the method m. In the following, we will rather use the infix notation $\mathtt{m.body}[j] \rightarrow \mathtt{m.body}[k]$.

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

- *for every execution path* P *from* `m.body`$[0]$ *to* f : `instr`*:* P = `m.body`$[0] \rightarrow^+ f$: `instr` *there exists a subpath which is a prefix of* P $subP$ = `m.body`$[0] \rightarrow^*$ $loopEntry$: `instr` *such that* f : `instr` $\notin$ $subP$
- *there is a path in which* $loopEntry$: `instr` *is executed immediately after the execution of* f : `instr` *(* f : `instr` $\rightarrow$ $loopEntry$: `instr`*)*

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

Note that in [1] reducibility is defined in terms of the dominator relation. Although not said exlicitely, the first condition in the upper definition corresponds to the dominator relation[3].

We illustrate the above definition with the control flow graph of the example from Fig. 2.1 in Fig. 1.2. In the figure, we rather show the execution relation between basic blocks which is a standard notion denoting a sequence of instructions which execute sequentially and where only the last one may be a jump and the first may be a target of a jump. The black edges represent a sequential execution relation, while dashed edges represent a backedge, i.e. the edge which stands for the execution relation between a final instruction (instruction at index `18`) in the bytecode cycle and the entry instruction of the cycle (instruction at index `19`).

[3] we decided to not introduce the standard definitions as it has several technical details for the exposition of which we would need more space and which are of not particular interest for the current thesis

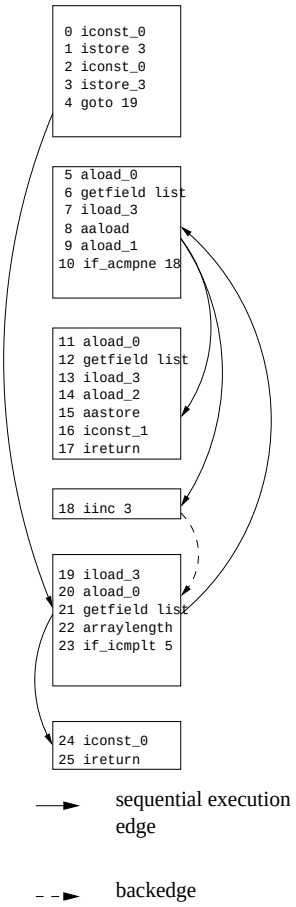

Figure 1.2: THE CONTROL FLOW GRAPH OF THE SOURCE PROGRAM FROM FIG.2.1

Chapter 2

Bytecode modeling language

2.1 Introduction

This section presents a bytecode level specification language, called for short BML and a compiler from a subset of the high level Java specification language JML to BML which from now we shall call JML2BML.

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 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 or 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 2.2, we give an overview of the main features of JML. A de-

tailed overview of BML is given in section 2.4. 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 2.6. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

2.2 Overview of JML

JML [17] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications which follows the design-by-contract approach (see [8]).

Over the last few years, JML has become the de facto specification language for Java source code programs. 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. [9]). One of the reasons for its success is that JML uses a Java-like syntax. Other important factors for the success of JML are its expressiveness and flexibility.

JML is supported by several verification tools. Originally, it has been designed as a language of the runtime assertion checker [12] created by G.T. Leavens and . The JML runtime assertion checker compiles both the Java code and the JML specification into executable bytecode and thus, in this case, the verification consists in executing the resulting bytecode. Several static checkers based on formal logic exist which use JML as a specification language. Esc/java [23] whose first version used a subset of JML [1] is among the first tools supporting JML. Among the static checkers with JML are the Loop tool developed by the Formal group at the University of Nijmegen, the Jack tool developed at Gemplus, the Krakatoa tool created by the Coq group at Inria, France. The tool Daikon [15] tool uses a subset of JML for detecting loop invariants by run of programs. A detailed overview of the tools which support JML can be found in [10].

Specifications in JML are written using different predicates which are side-effect free Java expressions, extended with specification-specific keywords. 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 the keywords **requires** and **ensures** are used respectively, **modifies** keyword introduces the locations that can be modified by the method, **loop_invariant** stands for a loop invariant, the **loop_modifies** keyword gives the locations modified by a loop etc. The latter is not standard in JML and is an extension introduced in [11]. 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.

[1] the current version of the tool esc/java 2 supports almost all JML constructs

Figure 2.1 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. Thus the method specification between lines 5 and 9 which exposes the method contract states the following. First the precondition (line 5) requires from any caller to assure that the instance variable `list` is not `null`. The frame condition (line 6) states that the method may only modify any of the elements in the instance field `list`. The method postcondition (lines 7—9) states the method will return `true` only if the replacement has been done. The method body contains a loop (lines 17—22) which is specified with a loop fame condition and a loop invariant (lines 13—16). The loop invariant (lines 14—16) says that all the elements of the list that are inspected up to now are different from the parameter object `obj1` as well as the local variable `i` is a valid index in the array `list`. The loop frame condition (line 13) states that only the local variable `i` and any element of the array field `list` may be modified in the loop.

```
1  public class ListArray {
2
3    private Object[] list;
4
5    //@ requires list != null;
6    //@ modifies list[*];
7    //@ ensures \result ==(\exists int i;
8    //@          0 <= i && i < list.length &&
9    //@          \old(list[i]) == obj1 && list[i] == obj2);
10   public boolean replace(Object obj1,Object obj2){
11     int i = 0;
12
13     //@ loop_modifies i, list[*];
14     //@ loop_invariant i <= list.length && i >=0
15     //@ && (\forall int k;0 <= k && k < i ==>
16     //@      list[k] != obj1);
17     for (i = 0; i < list.length; i++){
18       if ( list[i] == obj1){
19         list[i] = obj2;
20         return true;
21       }
22     }
23     return false;
24   }
25 }
```

Figure 2.1: CLASS ListArray WITH JML ANNOTATIONS

JML also allows the declaration of special JML variables, that are used only for specification purposes. These variables are declared in comments with the **ghost** modificator and may be used only in specification clauses. Those variables can also be assigned. Ghost variables are usually used for expressing properties which can not be expressed with the program variables.

Fig. 2.2 is an example for how ghost variables are used. The example shows the class `Transaction` which manages transactions in the program. The class is provided with a method for opening transactions `beginTransaction` and a method for closing transactions (`commitTransaction`). The specification declares a ghost variable `TRANS` (line 3) which keeps track if there is a running transaction or not, i.e. if the value of `TRANS` is 0 then there is no running transaction and if it has value 1 then there is a running transaction. The specification of the methods `beginTransaction` and `commitTransaction` models the property for no nested transactions. Thus, when the method `beginTransaction` is invoked the precondition (line 5) requires that there should be no running transaction and when the method is terminated the postcondition guarantees (line 6) that there is already a transaction running. We can also remark that the variable `TRANS` is set to its new value (line 8) in the body `beginTransaction`. Note that this high level property is difficult to express without the presence of the ghost variable `TRANS`.

```
1  public class Transaction {
2
3    //@ ghost static private int TRANS = 0;
4
5    //@ requires TRANS == 0;
6    //@ ensures TRANS == 1;
7    public void beginTransaction() {
8      //@ set TRANS = 1;
9        ...
10   }
11
12   //@ requires TRANS == 1;
13   //@ ensures TRANS == 0;
14   public void commitTransaction() {
15     //@ set TRANS = 0;
16       ...
17   }
18
19 }
```

Figure 2.2: SPECIFYING NO NESTED TRANSACTION PROPERTY WITH GHOST VARIABLE

A useful feature of JML is that it allows two kinds of method specification,

a *light* and *heavy* weight specification. An example for a *light* specification is the annotation of method `replace` (lines 5—9) in Fig. 2.1. The specification in the example states what is the expected behavior of the method and under what conditions it might be called. The user, however in JML, has also the possibility to write very detailed method specifications. This style of specification is called a *heavy* weight specification. It is introduced by the JML keywords **normal_behavior** and **exceptional_behavior**. As the keywords suggest every of them specifies a specific normal or exceptional behavior of a method. (see [21]).

The keyword **normal_behavior** introduces a precondition, frame condition and postcondition such that if the precondition holds in the prestate of the method then the method will terminate normally and the postcondition will hold in the poststate. Note that this clause guarantees that the method will not terminate on an exception and thus the exceptional postcondition for any kind of exception (i.e. for the exception class `Exception`) is **false**. An example for a *heavy* weight specification is given in Fig. 2.3. In the figure, method `divide` has two behaviors, one in case the method terminates normally (lines 11—14) and the other (lines 17—20) in case the method terminates by throwing an object reference of `ArithmeticException`. In the normal behavior case, the exceptional postcondition is ommitted specification as by default if the precondition (line 12) holds this assures that no exceptional termination is possible. Another observation over the example is that the exceptional behavior is introduced with the JML keyword **also**. The keyword **also** serves for introducing every new behavior of a method except the first one. Note that the keyword **also** is used in case a method overrides a method from the super class. In this case, the method specification (*heavy* or *light* weight) is preceded by the keyword **also** to indicate that the method should respect also the specification of the super method.

JML can be used to specify not only methods but also properties of a class or interfaces. A Java class may be specified with an invariant or history constraints. An invariant of a class is a predicate which holds at all visible states of every object of this class (see for the definition of visible state in the JML reference manual [13]). An invariant may be either static (i.e. talks only about static fields) or instance (talks about instance fields). A Class history constraints is a property which relates the initial and terminal state of every method in the corresponding class. The class `C` in Fig.2.3 has also an instance invariant which states that the instance variable `a` is always greater than `0`.

2.3 Design features of BML

Before proceeding with the syntax and semantics of BML, we would like to discuss the design choices made in the language. Particularly, we will see what are the benefits of our approach as well as the restrictions that we have to adopt. Now, we focus on the desired features of BML, how they compare to JML and what are the motivations that led us to these decisions:

- **Java compiler independance**

```
1  public class C {
2      int a;
3
4      //@ public instance invariant a > 0 ;
5
6      //@ requires val > 0 ;
7      public C(int val){
8         a = val ;
9      }
10
11     //@ public normal_behavior
12     //@ requires b > 0;
13     //@ modifies a;
14     //@ ensures  a == \old(a) / b;
15     //@
16     //@ also
17     //@ public exceptional_behavior
18     //@ requires b == 0;
19     //@ modifies \nothing;
20     //@ exsures (ArithmeticException) a == \old(a);
21     public void divide(int b) {
22         a = a / b;
23     }
24 }
```

Figure 2.3: An example for a method with a heavy weight specification in JML

Class files containing BML specification must not depend on any non optimizing compiler.

To do this, the process of the Java source compilation is separate from the JML compilation. More particularly, the JML2BML(short for the compiler from JML to BML) compiler takes as input a Java source file annotated with JML specification and its Java class produced by a non optimizing compiler containing a debug information.

- **JVM compatibility**
  The class files augmented with the BML specification must be executable by any implementation of the JVM specification. Because the JVM specification does not allow inlining of any user specific data in the bytecode instructions BML annotations must be stored separately from the method body (the list of bytecode instructions which represents its body).

  In particular, the BML specification is written in the so called user defined attributes in the class file. The JVM specification defines the format of

those attributes and mandates that any user specific information should be stored in such attributes. Note, that attribute which encodes the specification referring to a particular bytecode instruction contains information about the index of this instruction. For instance, BML loop invariants are stored in a user defined attribute in the class file format which contains the invariant as well as the index of the entry point instruction of the loop.

Thus, BML encoding is different from the encoding of JML specification where annotations are written directly in the source text as comments at a particular point in the program text or accompany a particular program structure. For instance, in Fig. 2.1 the reader may notice that the loop specification refers to the control structure which follows after the specification and which corresponds to the loop. This is possible first because the Java source language is structured, and second because writing comments in the source text does not violate the Java or the JVM specifications.

- **Compactness** and **Efficiency**
  Although opposite, we consider those two features together because they are mutually dependent. By the first, we mean that the class files augmented with BML should be as compact as possible. The second feature refers to that tools supporting BML should not be slowed down by the processing of the BML specification and more precisely we refer verification condition generator tools. This is an important condition if verification is done on devices with limited resources.

  For fulfilling these conditions, BML is designed to correspond to a subset of the desugared version of JML. In particular, it brings a relative compactness of the class file as well as makes the verification procedure more efficient.

  We first see in what sense this allows the class file compactness. Because every kind of BML specification clause is stored in a different user defined attribute, supporting all constructs of JML would mean that class files may contain a large number of attributes which would increase considerably the class file size. Of course, the size of a BML specification depends also on how much detailed is the specification, the more detailed it is, the larger size it would have.

  Because BML corresponds to a desugared version of JML, this means that on verification time the BML specification does not need much processing and thus, it can be easily translated to the data structures used in the verification scheme. This makes BML suitable for verification on devices with limitted resources.

As the attentive reader has noticed, we impose some restrictions on the structure of the class file format and the bytecode programs. These restrictions are the following:

- **Debug Information**
  A requirement to the class file format is that it must contain a debug

information, more particularly the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [25], yet almost all standard non optimizing compilers can generate these data. The **Line_Number_Table** is part of the compilation of a method and describes the link between the Java source lines and the Java bytecode. The **Local_Variable_Table** describes the local variables that appear in a method. This debug information is necessary for the compiler from JML to BML, as we shall see later in Section 2.6.

- **Reducible control flow graph**
  The control flow graph corresponding to the list of bytecode instructions resulting from the compilation of a method body must be a reducible control flow graph. An intuition to the notion of reducibility is that every cycle in the graph must have exactly one entry point, or in other words a cycle can not be jumped from outside inside (see [1] for the definition of reducibility). This condition is necessary for the compilation phase of the loop invariants as well as for the verification procedure (Section 4). Note, that this restriction is realistic as nonoptomizing Java compilers produce reducible control flow graphs and in practice even hand written code is in most cases reducible.

2.4 The subset of JML supported in 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 Section 2.4.1 gives the notation conventions adopted here and Section 2.4.2 gives the formal grammar of BML as well as an informal description of its semantics.

2.4.1 Notation convention

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

2.4.2 BML Grammar

$constants_{bml}$ ::= *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*

E_{bml} ::= $constants_{bml}$
| **reg**(*digits*)
| E_{bml}**.***ident*
| *ident*
| **arrayAccess**(E_{bml}, E_{bml})
| E_{bml} *op* E_{bml}
| **cntr**
| **st**(E_{bml})
| **\old(**E_{bml}**)**
| **\EXC**
| **\result**
| *boundVar*
| **\typeof(**E_{bml}**)**
| **\type(***ident***)**
| **\elemtype(**E_{bml}**)**
| **\TYPE**

op ::= **+** | **-** | **mult** | **div** | **rem**

$\mathcal{R}$ ::= $=|\neq|\leq|\leq|\geq|>|<:$

P_{bml} ::= E_{bml} $\mathcal{R}$ E_{bml}
| **true**
| **false**
| *not* P_{bml}
| $P_{bml} \wedge P_{bml}$
| $P_{bml} \vee P_{bml}$
| $P_{bml} \Rightarrow P_{bml}$
| $P_{bml} \iff P_{bml}$
| $\forall$ *boundVar*, P_{bml}
| $\exists$ *boundVar*, P_{bml}

classSpec ::= **invariant** *modifier* P_{bml}
| **classConstraint** P_{bml}
| **declare ghost** *ident ident*

modifier ::= **instance** | **static**

intraMethodSpec ::= **atIndex** *nat*;
assertion;

assertion ::= *loopSpec*

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

specCase ::= {|
requires P_{bml};
modifies *list locations*;
ensures P_{bml};
exsuresList
|}

exsuresList ::= [] | **exsures** (*ident*) P_{bml}; *exsuresList*

locations ::= E_{bml}.*ident*
| **reg**(i)
| *arrayModAt*(E_{bml}, *specIndex*)
| **everything**
| **nothing**

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

bmlKeyWords ::= **requires**
| **ensures**
| **modifies**
| **assert**
| **set**
| **exsures**
| **also**
| **invariant**
| **classConstraint**
| **atIndex**
| **loop_invariant**
| **loop_decreases**
| **loop_modifies**
| \ **typeof**
| \ **elemtype**
| **TYPE**
| **result**

2.4.3 Syntax and semantics of BML

In the following, we will discuss informally the semantics of the syntax structures of BML. Note that most of them have an identical counterpart in JML and their semantics in both languages is the same. In the following, we will concentrate more on the specific syntactic features of BML and will just briefly comment the BML features which it inherits from JML like for instance, preconditions

and which we have mentioned already[2].

BML expressions

Among the common features of BML and JML are the following expressions: field access expressions E_{bml}.*ident*, array access (**arrayAccess**(E^1_{bml}, E^2_{bml})), arithmetic expressions (E_{bml} *op* E_{bml}). Like JML, BML may talk about expression types. As the BML grammar shows, **typeof(**E_{bml}**)** denotes the dynamic type of the expression E_{bml}, **type(***ident***)** is the class described at index *ident* in the constant pool of the corresponding class file. The construction **elemtype(**E_{bml}**)** denotes the type of the elements of the array E_{bml}, and **TYPE** , like in JML, stands for the Java type `java.lang.Class`.

However, expressions in JML and BML differ in the syntax more particularly this is true for identifiers of local variables, method parameters, field and class identifiers. In JML, all these constructs are represented syntactically by their names in the Java source file. This is not the case in BML.

We first look at the syntax of method local variables and parameters. The class file format stores information for them in the array of local variables. That is why, both method parameters and local variables are represented in BML with the construct **reg**(i) which refers to the element at index i in the array of local variables of a method. Note that the `this` expression in BML is encoded as **reg**(0). This is because the reference to the current object is stored at index 0 in the array of local variables.

Field and class identifiers in BML are encoded by the respective number in the constant pool table of the class file. For instance, the syntax of field access expressions in BML is E_{bml}.*ident* which stands for the value in the field at index *ident* in the class constant pool for the reference denoted by the expression E_{bml}.

The BML grammar defines the syntax of identifiers differently from their usual syntax. Particularly, in BML those are positive numbers preceded by the symbol # while usually the syntax of identifiers is a chain of characters which always starts with a letter. The reason for this choice in BML is that identifiers in BML are indexes in the constant pool table of the corresponding class.

Fig.2.4 gives the bytecode as well as the BML specification of the code presented in Fig.2.3. As we can see, the names of the local variables, field and class names are compiled as described above. For instance, at line 3 in the specification we can see the precondition of the first specification case. It talks about **reg**(1) which is the element in the array of local variables of the method and which is the compilation of the method parameter `b` (see Fig. 2.3).

About the syntax of class names, after the **exsures** clause at line 5 follows a BML identifier (#25) enclosed in parenthesis. This is the constant pool index at which the Java exception type `java.lang.Exception` is declared.

A particular feature of BML is that it supports stack expressions which do not have a counterpart in JML. These expressions are related to the way

[2]because we have already discussed in Section 2.2 the JML constructs for pre and post-conditions, loop invariants, operators like **old**, **result**, etc. we would not return to them anymore as their semantics is exactly the same as the one of JML

```
1
2  Class instance invariant:
3     lv(0).#19 > 0;
4
5
6  Method specification:
7     {|
8      requires lv(1) > 0;
9      modifies lv(0).#19;
10     ensures  lv(0).#19 == \old( lv(0).#19 ) / lv(1);
11     exsures  ( #25 ) false;
12    |}
13    also
14    {|
15     requires lv(1) == 0;
16     modifies \nothing;
17     ensures false;
18     exsures ( #26 ) lv(0).#19  == \old(lv(0).#19);
19    |}
20
21  public void divide(int lv(1))
22        0 aload 0
23        1 dup
24        2 getfield #19 // instance field a
25        3 iload 1
26        4 idiv
27        5 putfield #19 // instance field a
28        6 return
```

Figure 2.4: An example for a heavy weight specification in BML

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. Stack expressions are represented in BML as follows:

- **cntr** represents the stack counter.

- **st**(E_{bml}) stands for the element in the operand stack at position E_{bml}. For instance, the element below the stack top is represented with **st**(**cntr** − 1) Note that those expressions may appear in predicates that refer to intermediate instructions in the bytecode.

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

Class Specification

The nonterminal *classSpec* in the BML grammar defines syntax constructs for the support of class specification. Note that these specification features exist in JML and have exactly the same semantics. However, we give a brief description of the syntax. Class invariants are introduced by the terminal **invariant**, history constraints are introduced by the terminal **classConstraint**. For instance, in Fig. 2.4 we can see the BML invariant resulting from the compilation of the JML specification in Fig. 2.3.

Like JML, BML supports ghost variables. As we can notice in the BML grammar, their syntax in the grammar is **declare ghost** *ident ident*. The first *ident* is the index in the constant pool which contains a description of the type of the ghost field. The second *ident* is the index in the constant pool which corresponds to the name of the ghost field.

Frame conditions

BML supports frame conditions for methods and loops. These have exactly the same semantics as in JML. The nonterminal that defines the syntax for frameconditions is *locations*. We look now what are the syntax constructs that may appear in the frame condition:

- E_{bml}**.***ident* states that the method or loop modifies the value of the field at index *ident* in the constant pool for the reference denoted by E_{bml}
- **reg**(i) states that the local variable may modified by a loop. Note that this kind of modified expression makes sense only for expressions modified in a loop. Indeed, 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 from the outside of the method.
- *arrayModAt*$(E_{bml}, specIndex)$ states that the components at the indexes specified by *specIndex* in the array denoted by E_{bml} may be modified. The indexes of the array components that may be modified *specIndex* have the following syntax:
  - i is the index of the component at index i. For instance, *arrayModAt*(E_{bml}, i) means that the array component at index i might be modified.

  - all specifies that all the components of the array may be modified, i.e. the expression $arrayModAt(E_{bml}, \text{all})$ means that any element in the array may potentially be modified.
  - $i_1..i_2$ specifies the interval of array components between the index i_1 and i_2.

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

Inter — method specification

In this subsection, we will focus on the method specification which is visible by the other methods in the program or in other words the method pre, post and frame conditions. The syntax of those constructs is given by the nonterminal *methodSpec*. As their meaning is exactly the same as in JML and we have already discussed the latter in Section 2.2, we shall not spend more lines here on those.

The part of the method specification which deserves more attention is the syntax of heavy weight method specification in BML. In Section 2.2, we saw that JML supports syntactic sugar for the definition of the normal and exceptional behavior of a method. The syntax BML does not support these syntactic constructs but rather supports their desugared version (see [30] for a detailed specification of the JML desugaring process). A specification in BML may declare several method specification cases like in JML. The syntactic structure of a specification case is defined by the nonterminal *specCase*.

We illustrate this with an example in Fig. 2.4. In the figure, we remark that BML does not have the syntactic sugar for normal and exceptional behavior. On the contrary, the specification cases now explicitely declare their behavior. The first specification case (the first bunch of specification enclosed in {| |}) corresponds to the **normal_behavior** specification case in the code from Fig. 2.3. Note that it does not have an analog for the JML keyword **normal_behavior** and that it declares explicitely what is the behavior of the method in this case, i.e. the exceptional postcondition is declared **false** for any exceptional termination.

The second specification case in Fig.2.4 corresponds to the **exceptional_behavior** case of the source code specification in Fig.2.3. It also specifies explicitely all details of the expected behavior of the method, i.e. the method postcondition is declared to be **false**.

Intra — method specification

As we can see from the formal grammar in subsection 2.4.2, 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*.

Note that a particularity of BML specification, i.e. loop specifications or assertion at particular program point contains information about the point in the method body at which it refers. For instance, the loop specification in BML given by the nonterminal *loopSpec* may contain apart from the loop invariant predicate (**loop_invariant**), the list of modified variables (**loop_modifies**) and the decreasing expression (**loop_decreases**) but also the index of the loop entry point instruction (**atIndex**).

We illustrate this with the example in Fig. 2.5 which represents the bytecode and the BML specification from the example in Fig. 2.1. The first line of the BML specification specifies that the loop entry is the instruction at index 19 in the array of bytecode instructions. The predicate representing the loop invariant introduced by the keword **loop_invariant** respects the syntax for BML expressions and predicates that we described above.

2.5 Well formed BML specification

In the previous Section 2.4, we gave the formal grammar of BML. However, we are interested in a strict subset of the specifications that can be generated from this grammar. In particular, we want that a BML specification is well typed and respects structural constraints. The constraints that we impose here are similar to the type and structural constraints that the bytecode verifier imposes over the class file format.

Examples for type constraints that a valid BML specification must respect :

- the array expression $\mathbf{arrayAccess}(E^1_{bml}, E^2_{bml})$ must be such that E^1_{bml} is of array type and E^2_{bml} is of integer type.
- the field access expression $E_{bml}.ident$ is such that E_{bml} is of subtype of the class where the field described by the constant pool element at index *ident* is declared
- For any expression $E^1_{bml} op E^2_{bml}$, E^1_{bml} and E^2_{bml} must be of a numeric type.
- in the predicate $E^1_{bml} r E^2_{bml}$ where $r = \leq, <, \geq, >$ the expressions E^1_{bml} and E^2_{bml} must be of numerical type.
- in the predicate $E^1_{bml} <: E^2_{bml}$, the expressions E^1_{bml} and E^2_{bml} must be of type $\backslash\mathbf{TYPE}$ (which is the same as `java.lang.Class`).
- the expression $\backslash\mathbf{elemtype}(E_{bml})$ must be such that E_{bml} has an array type.
- etc.

Example for structural constraint are :

```
1
2
3 Loop specification :
4
5     atIndex 19;
6     loop_modifies lv(0).#19[*], lv(3);
7     loop_invariant
8       lv(3) >= 0 &&
9       lv(3) < lv(0).#19.arrLength &&
10      \forall  bv_1 ;
11               ( bv_1 >= 0 &&
12               bv_1 < lv(0).#19.arrLength ==>
13                    lv(0).#19[bv_1] != lv(1) )
14
15  public int replace(Object  lv(1),Object lv(2) )
16  0 const 0
17  1 store 3
18  2 const 0
19  3 store 3
20  4 goto 19
21  5 load 0
22  6 getfield #19 // instance field list
23  7 load 3
24  8 aaload
25  9 load 1
26 10 if_acmpne 18
27 11 load 0
28 12 getfield #19 // instance field list
29 13 load 3
30 14 load 2
31 15 aastore
32 16 const 1
33 17 return
34 18 iinc 3
35 19 load 3    // loop entry
36 20 load 0
37 21 getfield #19 // instance field list
38 22 arraylength
39 23 if_icmplt 5
40 24 const 0
41 25 return
```

Figure 2.5: An example for a loop specification in BML

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

- every *ident* in a BML specification must be a correct index in the constant pool table.

- if the expression **reg**(i) appears in a method BML specification, then i must be a valid index in the array of local variables of the method

An extension of the bytecode verifier may perform the checks if a BML specification respects this kind of structural and type constraints. However, we have not worked on this problem and is a good candidate for a future work. For the curious reader, it will be certainly of interest to turn to the Java Virtual Machine specification [25] which contains the official specification of the Java bytecode verifier or to the existing literature on bytecode verification (see the overview article [24])

2.6 Compiling JML into BML

In this section, we turn to the JML2BML compiler. As we shall see, the compilation consists of several phases, namely compiling the Java source file, preprocessing of the JML specification, resolution and linking of names, locating the position of intra — method specification, processing of boolean expressions and finally encoding the BML specification in user defined class file attributes. (their structure is predefined by JVMS). In the following, we look in details at the phases of the compilation process:

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. Those attributes are important for the next phases of the JML compilation.

2. Compilation of Ghost field declarations
   JML specification is invisible by the Java compilers. Thus Java compilers omit the compilation of ghost variables declaration. That is why it is the responsibility of the JML2BML compiler to do this work. For instance, the compilation of the declaration of the ghost variable from Fig. 2.2 is given in Fig.2.6 which shows the data structure **Ghost_field_Attribute** in which the information about the field `TRANS` is encoded in the class file format. Note that, the constant pool indexes **#18** and **#19** which contain its description were not in the constant pool table of the class file `Transaction.class` before running the JML2BML compiler on it.

```
Ghost_field_Attribute {
    ...
    {  access_flag 10;
       name_ index = #18;
       descriptor_index = #19
    } ghost[1];
}
```

- **access_flag**: The kind of access that is allowed to the field
- **name_index**: The index in the constant pool which contains information about the source name of the field
- **descriptor_index**: The index in the constant pool which contains information about the name of the field type

Figure 2.6: Compilation of ghost variable declaration

3. Desugaring of the JML specification
   The phase consists in converting the JML method heavy-weight behaviours and the light - weight non complete specification into BML specification cases. It corresponds to part of the standard JML desugaring as described in [30]. For instance, the BML compiler will produce from the specification in Fig.2.3 the BML specification given in Fig.2.4

4. 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 their corresponding data structures in the class file. 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.

   For instance, consider once again the example in Fig. 2.3 and more particularly the first specification case of method `divide` whose precondition `b > 0` contains the method parameter identifier `b`. In the linking phase, the identifier `b` is resolved to the local variable **reg**(1) in the array of local variables for the method `divide`. We have a similar situation with the postcondition `a == \old(a) / b` which mentions also the field `a` of the current object. The field name `a` is compiled to the index in the class constant pool which describes the constant field reference. The result of the linking process is in Fig.2.4.

   If, in the JML specification a field identifier appears for which no constant pool index exists, it is added in the constant pool and the identifier in question is compiled to the new constant pool index. This happens when

declarations of JML ghost fields are compiled.

5. Locating the points for the intra —method specification

   In this phase 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 in the bytecode. For this, the **Line_Number_Table** attribute is used. The **Line_Number_Table** attribute describes the correspondence between the Java source line and the instructions of its respective bytecode. In particular, for every line in the Java source code the **Line_Number_Table** specifies the index of the beginning of the basic block[3] in the bytecode which corresponds to the source line. Note however, that a source line may correspond to more than one instruction in the **Line_Number_Table**.

   This poses problems for identifying loop entry instruction of a loop in the bytecode which corresponds to a particular loop in the source code. For instance, for method `replace` in the Java source example in Fig. 2.1 the java compiler will produce two lines in the **Line_Number_Table** which correspond to the source line **17** as shown in Fig. 2.7. The problem is that none of the basic bloks determined by instructions **2** and **18** contain the loop entry instruction of the compilation of the loop at line **17** in Fig. 2.1. Actually, the loop entry instruction in the bytecode in Fig. 2.5 (remember that this is the compilation in bytecode of the Java source in Fig. 2.1) which corresponds to the in the bytecode is at index **19**.

   Thus for identifying loop entry instruction corresponding to a particular loop in the source code, we use an heuristics. It consists in looking for the first bytecode loop entry instruction starting from one of the **start_pc** indexes (if there is more than one) corresponding to the start line of the source loop in the **Line_Number_Table**. The algorithm works under the assumption that the control flow graph of the method bytecode is reducible. This assumption guarantees that the first loop entry instruction found starting the search from an index in the **Line_Number_Table** corresponding to the first line of a source loop will be the loop entry corresponding to this source loop. However, we do not have a formal argumentation for this algorithm because it depends on the particular implementation of the compiler. From our experiments, the heuristic works successfully for the Java Sun non optimizing compiler.

6. Compilation of the JML boolean expressions into BML

   Another important issue in this stage of the JML compilation is how the type differences on source and bytecode level are treated. By type

[3] a basic block is a sequence of instructions which does not contain jumps except may be for the last instruction and neither contains target of jumps except for the first instruction. This notion comes from the compiler community and more information on this one can find at [1]

Line_Number_Table

start_pc	**line**
...	
2	**17**
18	**17**

Figure 2.7: **Line_Number_Table** FOR THE METHOD `replace` IN FIG. 2.1

differences we refer to the fact that the JVM (Java Virtual Machine) does not provide direct support for integral types like byte, short, char, neither for boolean. Those types are rather encoded as integers in the bytecode. Concretely, this means that if a Java source variable has a boolean type it will be compiled to a variable with an integer type.

For instance, in the example for the method `replace` and its specification in Fig.2.1 the postcondition states the equality between the JML expression **result** and a predicate. This is correct as the method `replace` 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 `replace` 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[4].

Finally, the compilation of the postcondition of method `replace` is given in Fig. 2.8. 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.2.1 is compiled into logical equivalence.

7. Encoding BML specification into user defined class attributes
   The specification expression and predicates are 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.

   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

[4] when generating proof obligations we add for every source boolean expression an assumption that it must be equal to 0 or 1. A reasonable compiler would encode boolean values in a similar way

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

$$\Longleftrightarrow$$

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

Figure 2.8: The compilation of the postcondition in Fig. 2.1

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

- **index**: The index in the `LineNumberTable` where the beginning of the corresponding loop is described
- **modifies**[]: The array of locations that may be modified
- **invariant** : The predicate that is the loop invariant. It is a compilation of the JML formula in the low level specification language
- **decreases**: The expression which decreases at every loop iteration

Figure 2.9: Structure of the Loop Attribute

is given in Fig. 2.9. This attribute is an array of data structures each describing a single loop from the method source code. From the figure, we notice that every element describing the specification for a particular loop contains the index of the corresponding loop entry instruction **index**, the loop modifies clause (**modifies**), the loop invariant (**invariant**), an expression which guarantees termination (**decreases**).

Chapter 3

Assertion language for the verification condition generator

In this chapter we shall focus on a particular fragment of BML which will be extended with few new constructs. The part of BML in question is the assertion language that our verification condition generator manipulates as we shall see in the next Chapter **??**.

The assertion language presented here will abstract from most of the BML specification clauses described in Section 2.4. Our interest will be focused only on method and loop specification. Some parts of BML will be completely ignored either for keeping things simple or because those parts are desugared and result into the BML fragment of interest. For instance, we do not consider here multiple method specification cases, neither assertions in particular program point for the first reason. The assertion language presented here discards also class invariants and history constraints because they boil down to method pre and postconditions.

The rest of this chapter is organized as follows. Section 3.1 presents what is exactly the BML fragment of interest and its extensions. Section 3.4 shows how we encode method and loop specification as well as presents a discussion how some of the ignored BML specification constructs are transformed into method pre and postconditions. The last two sections are concentrated on the formal meaning of the assertion language, i.e. Section 3.2 defines the substitution for the assertion language and Section 3.3 gives formal semantics of the assertion language.

3.1 The assertion language

The assertion language in which we are interested corresponds to the BML expressions (nonterminal E_{bml}) and predicates (nonterminal P_{bml}) extended with several new constructs. The extensions that we add are the following:

- Extensions to expressions. The assertion language that we present here must be suitable for the verification condition calculus. Because the verification calculus talks about updated field and array access we should be able to express them in the assertion language. Thus we extend the grammar of BML expression with the following constructs concerning update of fields and arrays :
  - update field access expression $f[\oplus E_{bml} \rightarrow E_{bml}](E_{bml})$.
  - update array access expression
    $\text{arrAccess}[\oplus(E_{bml}, E_{bml}) \rightarrow E_{bml}](E_{bml}, E_{bml})$

  The verification calculus will need to talk about reference values. Thus we extend the BML expression grammar to support reference values *RefVal*. Note that in the following integers **int** and *RefVal* will be referred to with *Values*.

- Extensions to predicates. Our bytecode language is object oriented and thus supports new object creation. Thus we will need a means for expressing that a new object has been created during the method execution.

  We extend the language of BML formulas with a new user defined predicate **instances**(*RefVal*). Informally, the semantics of the predicate **instances**(`ref`) where `ref` $\in$ *RefVal* means that the reference `ref` has been allocated when the current method started execution.

The assertion language will use the names of fields and classes for the sake of readability instead of their corresponding indexes in the constant pool as is in BML.

We would like to discuss in the following how and why BML constructs like class invariants and history constraints can be expressed as method pre and postconditions:

- Class invariants. A class invariant (**invariant**) is a property that must hold at every visible state of the class. This means that a class invariant must hold when a method is called and also must be established at the end of a method execution. A class invariant must be established in the poststate of the constructor of this class. Thus the semantics of a class invariant is part of the pre and postcondition of every method and is a part of the postcondition of the constructor of the class.

- History constraints. A class history constraint (**classConstraint**) gives a relation between the pre and poststate of every method in the class. A class history constraint thus can be expressed as a postcondition of every method in the class.

3.2 Substitution

In this section we focus on how substitution is defined in our assertion language. Basically, it is is defined inductively in a standard way over the expression structure. Still, we extend substitution to deal with field and array update as follows:

$$E_{bml}[f \leftarrow f[\oplus E_{bml} \rightarrow E_{bml}]]$$

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

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

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

$$E_{bml}.f^1[f^2 \leftarrow f^2[\oplus E^1_{bml} \longrightarrow E^2_{bml}]] =$$

$$\begin{cases} E_{bml}.f^1 & if\ f^1 \neq f^2 \\ E_{bml}.f^2[\oplus E^1_{bml} \longrightarrow E^2_{bml}] & else \end{cases}$$

$$E_{bml}.f^1[\oplus E^1_{bml} \rightarrow E^2_{bml}][f^2 \leftarrow f^2[\oplus E^3_{bml} \rightarrow E^4_{bml}]] =$$

$$\begin{cases} f^1[\oplus E^1_{bml}[f^2 \leftarrow f^2[\oplus E^3_{bml} \rightarrow E^4_{bml}]] \rightarrow E^2_{bml}[f^2 \leftarrow f^2[\oplus E^3_{bml} \rightarrow E^4_{bml}]]] & if\ f^1 \neq f^2 \\ f^1\ \begin{array}{l}[\oplus E^1_{bml}[f^2 \leftarrow f^2[\oplus E^3_{bml} \rightarrow E^4_{bml}]] \longrightarrow E^2_{bml}[f^2 \leftarrow f^2[\oplus E^3_{bml} \rightarrow E^4_{bml}]]] \\ [\oplus E^3_{bml} \longrightarrow E^4_{bml}]\end{array} & else \end{cases}$$

For example, consider the following substitution expression:

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

This results in the new expression :

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

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

3.3 Interpretation

We discuss the evaluation of expressions and interpretation of predicates in a particular program state configuration. Thus, we first define a function for

expression evaluation, as well as a function which for a given state and predicate returns the interpretation of the given predicate in the given state. The function *eval* which evaluates expressions in a state has the following signature:

$$eval : E_{bml} \to S \to S \rightharpoonup Values \cup JType$$

Note that the evaluation function is partial and takes as arguments an expression of the assertion language presented in the previuos Section 3.1 and two states (see Section 1.6) and returns a value as defined in Section 1.5.

Definition 3.3.0.1 (Evaluation of expressions) *The evaluation in a state* $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *or* $s =< \mathrm{H}, \mathrm{Reg} >^{final}$ *Final of an expression* E_{bml} *w.r.t. an initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *is denoted with* $\|E_{bml}\|_{s_0,s}$ *and is defined inductively over the grammar of expressions* E_{bml} *as follows:*

$$\|v\|_{s_0,s} = v$$
$$where\ v \in\ \mathbf{int}\ \vee\ v \in RefVal$$

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

$$\|f[\oplus E^1_{bml} \to E^2_{bml}](E^3_{bml})\|_{s_0,s} =$$
$$= \mathrm{H}[\oplus f \to f[\oplus \|E^1_{bml}\|_{s_0,s} \to \|E^2_{bml}\|_{s_0,s}]](f)(\|E^3_{bml}\|_{s_0,s})$$

$$\|\mathbf{arrayAccess}(E^1_{bml}, E^2_{bml})\|_{s_0,s} =$$
$$= \mathrm{H}(\|E^1_{bml}\|_{s_0,s}, \|E^2_{bml}\|_{s_0,s})$$

$$\|\mathrm{arrAccess}[\oplus(E^1_{bml}, E^2_{bml}) \to E^3_{bml}](E^4_{bml}, E^5_{bml})\|_{s_0,s} =$$
$$= \mathrm{H}[\oplus(\|E^1_{bml}\|_{s_0,s}, \|E^2_{bml}\|_{s_0,s}) \to \|E^3_{bml}\|_{s_0,s}]$$
$$(\|E^4_{bml}\|_{s_0,s}, \|E^5_{bml}\|_{s_0,s})$$

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

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

$$\|E^1_{bml}\ op\ E^2_{bml}\|_{s_0,s} = \|E^1_{bml}\|_{s_0,s} op \|E^2_{bml}\|_{s_0,s}$$

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

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

$$\|\backslash\mathbf{TYPE}\|_{s_0,s} = \ \mathtt{java.lang.Class}$$

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

$$\|\mathbf{cntr}\|_{s_0,s} = \mathrm{Cntr}$$

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

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

$$\|\backslash\mathbf{result}\|_{s_0,s} = \mathrm{Res} \quad where\ s =< \mathrm{H}, \mathrm{Reg} >^{norm} \mathrm{Res}$$
$$\|\backslash\mathbf{EXC}\|_{s_0,s} = \mathrm{Exc} \quad where\ s =< \mathrm{H}, \mathrm{Reg} >^{exc} \mathrm{Exc}$$

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

Definition 3.3.0.2 (Interpretation of predicates) *The interpretation* $s \vDash P$ *of a predicate* P *in a state configuration* $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *w.r.t. an initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *is defined inductively as follows:*

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

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

$s, s_0 \vDash \neg\ P$ *if and only if not* $s, s_0 \vDash P$

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

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

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

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

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

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

$$s, s_0 \vDash E^1_{bml}\ \mathcal{R}\ E^2_{bml} \text{ if and only if } \begin{array}{l} \|E^1_{bml}\|_{s_0,s} \neq \perp \wedge \\ \|E^2_{bml}\|_{s_0,s} \neq \perp \wedge \\ \|E^1_{bml}\|_{s_0,s}\ rel(\mathcal{R}) \|E^2_{bml}\|_{s_0,s} \text{ is true} \end{array}$$

$s, s_0 \vDash$ **instances**(`ref`), *where* `ref` $\in$ *RefVal if and only if inList*(`ref`, getLoc(H_0))

3.4 Extending method declarations with specification

In the following, we propose an extension of the method formalization given in Section 1.4. The extension takes into account the method specification. The extended method structure is given below:

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

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

- `m`.pre gives the precondition of the method, i.e. the predicate that must hold whenever `m` is called
- `m`.normalPost is the postcondition of the method in case `m`terminates normally
- `m`.modif is also called the method frame condition. It is a list of expressions that the method may modify during its execution
- `m`.excPostSpec is a total function from exception types to formulas which returns the predicate `m`.excPostSpec(`Exc`) that must hold in the method's poststate if the method `m` terminates on an exception of type `Exc`. Note that this function is constructed from the **exsures** clause of a method introduced in Chapter 2.1, section 2.4. For instance, if method `m` has an exsures clause:

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

  then for every exception type `SExc` such that subtype (`SExc` ,`Exc`) the function the result of the function `m`.excPostSpec for `SExc` is `m`.excPostSpec(`SExc`) = $\textbf{reg}(1) = \textbf{null}$. If for an exception `Exc` there is not specified exsures clause then the function excPostSpec returns the default exceptional postcondition predicate *false* , i.e. `m`.excPostSpec(`Exc`) = *false*
- `m`.loopSpecS is an array of **LoopSpec** data structures which give the specifcication information for a particular loop in the bytecode

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

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

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

- the field $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}$ is a valid index in the body of $\mathtt{m}$: $0 \leq \mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos} < \mathtt{m}.\mathsf{body}.length$ and is a loop entry instruction in the sense of Def.1.9.2
- $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{invariant}$ is the predicate that must hold whenever the instruction $\mathtt{m}.\mathsf{body}[\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}]$ is reached in the execution of the method $\mathtt{m}$
- $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{modif}$ are the locations such that for any two states $state_1$, $state_2$ in which the instruction $\mathtt{m}.\mathsf{body}[\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}]$ executes agree on local variables and the heap modulo the locations that are in the list modif. We denote the equality between $state_1$, $state_2$ modulo the modifies locations like this $state_1 =^{\mathsf{modif}} state_2$

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 1.8 and thus, it deals with stack manipulation, object creation and manipulation, field access and update, as well as exception throwing and handling.

Different ways of generating verification conditions exist. The verification condition generator presented propagates the weakest precondition and exploits the information about the modified locations by methods and loops. In Section 4.1, we discuss the existing approaches and motivate the choice done here.

Bytecode verification has become lately quite fashionable, thus several works exist on bytecode verification. Section 4.2 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 2.1, our approach consists in compiling source

specification into bytecode specification. When compiling a loop invariant, we need to know where exactly in the bytecode the invariant must hold. Section 1.9 introduces the notion of a loop in an unstructured program.

As we stated earlier, our verification condition generator is based on a weakest precondition (wp) calculus. As we shall see in Section 4.3 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 Discussion

In this section, we make an overview of the different ways for generating verification conditions. Next, we shall see how we argument our design decisions of the verification condition generator presented here.

Our verification condition generator has the following features :

- it is based on a weakest precondition predicate transformer
  The weakest precondition generates a precondition predicate starting from the end of the program with a specified postcondition and "goes " in a backward direction to the entry point of the program. There is an alternative for generating verification condition which works in a forward direction called a strongest postcondition predicate transformer. However, strongest postcondition tends to generate large formulas which is less practical than the more concise formulae generated by the weakest precondition calculus. Next, it generates existential quantification for every assignment expression in a program which are not easily treated by automatic theorem provers. For more detailed information on strongest postcondition calculus the reader may refer to [14].

- it works directly on the bytecode
  Another possible approach is to generate verification conditions over a guarded command language program. This in particular would mean that the verification procedure would have one more stage where the bytecode programs is transformed in a program in a guarder command language. A guarded command language is useful for an interactive verification and is the case for the extended static checker ESC/java ([23]) and Spec# ([4]). The reason for this is that its representation is close to the semantics of the original program and thus is understandable by programmers.

  However, we consider that a guarded command language is impractical for our purposes for several reasons. First, the transformation is usually a complex procedure which needs computational resources. This could be a problem, if the verification procedure is done on a small device with limitted resources. Second, proving the transformation correct is not trivial. We consider that performing the verification procedure directly over the original bytecode program avoids the aformentioned problems.

- it propagates the verification conditions up to the program entry instruction
  For this feature we also have an alternative solution. An alternative is that verification conditions are discharged immediately when a loop entry is reached by the verifiction condition generator (see). These verification conditions (in the case of a weakest precondition predicate calculus) state that the loop invariant implies the postcondition of the loop if the loop condition is not true and that the invariant implies the weakest precondition of the loop body if the loop condition holds. Although, this verification condition generator is simpler than our approach it needs much stronger invariants than the verification condition generator proposed here. In particular, the specification required for this alternative approach may increase the size of the program considerably which will be not desirable if for instance the program and its specification must be sent via the network.

4.2 Related work

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

Floyd is among the first to work on program verification using logic methods for unstructured program languages (see [31]). Following the Floyd's approach, T.Hoare gives a formal logic for program verification in [18] known today under the name Hoare logic. Dijkstra [14] 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.

As Java has been gaining popularity in industry since the nineties of the twentieth century, it also attracted the research interest. Thus the nineties upto nowadays give rise to several verification tools tailored to Java based on Hoare logic. Among the ones that gained most popularity are esc/java developed at Compaq [23], the Loop tool [19], Krakatoa, Jack [11] etc.

Few works have been dedicated to the definition of a bytecode logic. Among the earliest work in the field of bytecode verification is the thesis of C.Quigley [29] 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 [7] 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 [3] 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.

The Spec# ([4]) 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 [26] that is performed includes several stages of processing the bytecode program: elimination of irreducible loops, transformation into an acyclic control flow graph, translation of the bytecode into a guarded passive command language program. Despite that here in our implementation we also do a transformation in the graph into an acyclic program, we consider that in a mobile code scenario one should limit the number of program transformations for several reasons. First, we need a verification procedure as simple as possible, and second every transformation must be proven correct which is not always trivial.

4.3 Weakest precondition calculus

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

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

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

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

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

In order to define the *wp* function, we will need two other notions. The first one is a function which will determine the predicate between two instructions

that are in execution relation as defined in Def. 1.9.1. Note that this is not necessary for structured programs. However, for unstructured programs with loops annotated with invariants and frame conditions, this is a necessary step. The definition of the intermediate predicate is given in the next subsection 4.3.1. We will also see how the weakest precondition is defined in presence of exceptions. This is done in subsection **??**.

4.3.1 Intermediate predicates

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

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

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

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

Definition 4.3.1 (Intermediate predicate between two instructions) *Assume that* j : `instr` $\rightarrow k$: `instr`. *The predicate* $inter(j, k, \mathtt{m})$ *must hold after the execution of* j : `instr` *and before the execution of* k : `instr` *and is defined as follows:*

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

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

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

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

- *else*

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

4.3.2 Weakest precondition in the presence of exceptions

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

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

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

Definition 4.3.2.1 (Postcondition in case of a thrown exception)

$$\mathtt{m}.\mathsf{getExcPostIns}(i,\ \mathtt{Exc}) = \begin{cases} inter(i, \mathsf{handlerPc}, \mathtt{m}) & if\ findExcHandler(\ \mathtt{Exc}, i, \mathtt{m}.\mathsf{excHndlS}) = \mathsf{handlerPc} \\ \mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{Exc}) & findExcHandler(\ \mathtt{Exc}, i, \mathtt{m}.\mathsf{excHndlS}) = \bot \end{cases}$$

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

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

m.excPost(i, Exc) returns the predicate that must hold in the preststate of the instruction at index i which may throw a runtime exception of type Exc. Note that the function m.excPost does not deal with programmatic exceptions thrown by the instruction athrow , neither exception caused by a method invokation (execution of instruction invoke) as the exceptions thrown by those instructions are handled in a different way as we shall see later in the definition of the *wp* function in Section 4.3.

The function application m.excPost(i, Exc) is defined as follows:

Definition 4.3.2.2 (Auxuliary function for instructions throwing runtime exceptions)

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

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

4.3.3 Rules for single instruction

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

- Control transfer instructions
  1. unconditional jumps

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

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

  2. conditional jumps

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

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

  3. return

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

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

- load and store instructions
  1. load a local variable on the operand stack

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

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

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

2. store the stack top element in a local variable

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

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

3. push an integer constant on the operand stack

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

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

4. incrementing a local variable

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

- arithmetic instructions

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

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

We illustrate this rule with an example. Let us have the arithmetic instruction add at index i such that the predicate $inter(i, i+1, \mathtt{m}) \equiv$

$\mathbf{st}(\mathbf{cntr}) \geq 0$. In this case, applying the rule we get that the weakest precondition is $\mathbf{st}(\mathbf{cntr}-1)+\mathbf{st}(\mathbf{cntr}) \geq 0$:

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

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

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

- object creation and manipulation

1. create a new object

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

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

2. array creation

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

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

3. field access

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

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

4. field update

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

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

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

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

After applying the substitution following the rules described in Section 3.2, we obtain that the precondition is

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

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

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

5. access the length of an array

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

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

6. checkcast

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

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

7. instanceof

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

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).

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

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

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

The third part of the rule deals with the exceptional termination of the method invokation. In this case, if the invoked method `n` terminates on

any exception which belongs to the array of exceptions n.exceptions that n may throw. Two cases are considered - either the thrown exception can be handled by m or not. If the thrown exception Exc can not be handled by the method m (i.e. *findExcHandler*(n .excPostSpec(n .exceptions[j]), i, m.excHndlS) = $\perp$) then if the exceptional postcondition predicate n .excPostSpec(Exc) of n holds then m.excPostSpec(Exc) for any value of the thrown exception object. In case the thrown exceptionExc is handled by m, i.e. *findExcHandler*(n .excPostSpec(n .exceptions[j]), i, m.excHndlS) = k then if the exceptional postcondition n .excPostSpec(Exc) of n holds then the intermediate predicate *inter*(i, k, m) that must hold after i : instr and before k : instr must hold once again for any value of thrown exception.

- throw exception instruction

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

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

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

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

4.4 Example

In the following, we will consider an example of the application of the verification procedure with *wp*. Consider Fig. 4.1, which gives an example of a Java method which calculates the square of its input which is stated in its postcondition. The calculation of the square of the parameter i is done with an iteration which sums all the impair numbers 2∗s + 1, 0<=s <=i in the local variable sqr. The invariant states that whenever the loop entry is reached the variable sqr will

```
1  //@ ensures \result == i*i;
2  public int square( int i ) {
3    int sqr  = 0;
4    if ( i < 0) {
5      i = -i;
6    }
7    //@ loop_modifies s, sqr;
8    //@ loop_invariant (0 <= s) && (s <= i) && sqr == s*s ;
9    for (int s = 0 ; s < i; s++ ) {
10     sqr = sqr + 2*s + 1;
11   }
12   return sqr;
13 }
```

Figure 4.1: JAVA METHOD WHICH CALCULATES THE SQUARE OF ITS INPUT

contain the square of the local variable s and that 0 <=s <=i. In Fig.4.2, we show the bytecode of method square. The weakest precondition for a fragment of the instructions are shown in Fig. 4.3.

Fig.4.3 shows the resulting preconditions for some of the instructions in the bytecode of the method square. In the figure, the line before every instruction gives the calculated weakest precondition of the instruction in the execution path which reaches the end of the method. Thus, the weakest precondition of the instruction return at line 28 states that before the instruction is executed the stack top element **st**(**cntr**) must contain the square of the local variable **reg**(1) . Note that this precondition is calculated from the method postcondition which is given in curly brackets at line 38. The instruction before the return instruction has as precondition that the local variable **reg**(2) must be equal to the square of **reg**(1) . The instruction if_cond at line 24 has as weakest precondition that if the stack element below the stack top element **st**(**cntr** - 1) is not smaller than **st**(**cntr**) then **reg**(2) == **reg**(1) ***reg**(1) . Note that we give only a part of the precondition of this instruction for the sake of clarity. In particular, we give the precondition which must hold if the condition is not true, or in other words the precondition of the if_cond instruction for the execution path which goes to the end of the method. The case which deserves more attention is the instruction goto at line 14 which jumps to the loop entry instruction at line 21. As discussed in Section 4.3.1, the weakest precondition of this goto consists in the specified loop invariant and the a formula which states that the invariant implies the precondition of the loop entry.

Another point to notice is that the instruction at line 19 which is a loop end instruction w.r.t. Def 1.9.2 has as precondition the loop invariant where the **reg**(3) is incremented.

```
1  0  const 0
2  1  store 2
3  2  load 1
4  3  if\_ge 7
5  4  load 1
6  5  neg
7  6  store 1
8  7  const 0
9  8  store 3
10 9  goto 19
11 10 load 2
12 11 const 2
13 12 load 3
14 13 mul
15 14 add
16 15 const 1
17 16 add
18 17 store 2
19 18 iinc 3
20 //loop start
21 19 load 3
22 20 load 1
23 21 if\_icmplt 10
24 22 load 2
25 23 return
```

$$\text{square.normalPost} = \backslash\mathbf{result} == \mathbf{reg}(1) * \mathbf{reg}(1)$$

$$\text{square.loopSpecS} = \begin{cases} \mathsf{pos} = 19 \\ \\ \mathsf{invariant} = \begin{array}{l} 0 <= \mathbf{reg}(3) \wedge \\ \mathbf{reg}(3) <= \mathbf{reg}(1) \wedge \\ \mathbf{reg}(2) = \mathbf{reg}(3) * \mathbf{reg}(3) \end{array} \\ \\ \mathsf{modif} = \mathbf{reg}(3), \mathbf{reg}(2) \end{cases}$$

Figure 4.2: BYTECODE OF METHOD SQUARE AND ITS SPECIFICATION

```
1  ...
2  // invariant initialization
3  {   0 <= lv(3) &&
4      lv(3) <= lv(1) &&
5      lv(2) = lv(3) * lv(3) }
6  // invariant implies the loop postcondition
7  { forall lv(3), forall lv(2),
8      0 <= lv(3) &&
9      lv(3) <= lv(1) &&
10     lv(2) = lv(3) * lv(3) &&
11     not(lv(3) < lv(1))==>
12        lv(2) = old(lv(1)) * old(lv(1)) }
13 9  goto 19
14
15 ...
16
17 {  0 <= lv(3) + 1 &&
18     lv(3) + 1 <= lv(1) &&
19     lv(2) = (lv(3) + 1) * ( lv(3) + 1) }
20 18 iinc 3
21
22 { not (lv(3) < lv(1)) ==>lv(2) == lv(1)*lv(1) }
23 19 load 3 //loop start
24
25 { not (st(cntr) < lv(1))==>lv(2)==lv(1)*lv(1) }
26 20 load 1
27
28 {not ( st(cntr - 1)<st(cntr))==>lv(2) == lv(1)*lv(1)}
29 21 if icmplt 10
30
31 {lv(2) == lv(1)*lv(1)    }
32 22 load 2
33
34 {st(cntr) == lv(1)*lv(1) }
35 23 return
36
37 { \result == lv(1)*lv(1) }
```

Figure 4.3: BYTECODE OF METHOD SQUARE AND WEAKEST PRECONDITIONS FOR A FRAGMENT OF THE EXECUTION PATH WHICH REACHES THE METHOD END

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 1.8. The way in which the proof is done is standard. Note that the formalization of the operational semantics in terms of relation on states serves us to give a model for our assertion language.

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

5.1 Substitution properties

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

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

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

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

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

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

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

     $(right)$ $\|\mathbf{reg}(i)\|_{s_0,s_2} =$
     { *by Def.3.3.0.1 of the evaluation for local variables* }
     (2) $= \|Expr_2\|_{s_0,s_1}$
     { *from (1) and (2) we get that the lemma holds in this case* }

   - $Expr_1 = Expr_3.f$

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

     { *by Def.3.3.0.1 of the evaluation for field access expressions* }
     $(left)$ $\|Expr_3.f[\mathbf{reg}(i) \leftarrow Expr_2]\|_{s_0,s_1} =$
     $= \mathrm{H}(f)(\|Expr_3[\mathbf{reg}(i) \leftarrow Expr_2]\|_{s_0,s_1})$

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

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

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

2. second case of the lemma

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

     $\{$ *from the first part of the lemma we get* $\}$
     (1) $\|Expr'[\mathbf{reg}(i) \leftarrow Expr_2]\|_{s_0,s_1} = \|Expr'\|_{s_0,s_2}$
     (2) $\|Expr''[\mathbf{reg}(i) \leftarrow Expr_2]\|_{s_0,s_1} = \|Expr''\|_{s_0,s_2}$

     $s_1, s_0 \vDash \psi[\mathbf{reg}(i) \leftarrow Expr_2]$
     $\{$ *definition of substitution* $\}$
     (3) $\equiv$
     $s_1, s_0 \vDash Expr'[\mathbf{reg}(i) \leftarrow Expr_2] \ \mathcal{R} \ Expr''[\mathbf{reg}(i) \leftarrow Expr_2]$
     $\{$ *by Def.3.3.0.2 we get* $\}$
     $\Longleftrightarrow$
     $\|Expr'[\mathbf{reg}(i) \leftarrow Expr_2]\|_{s_0,s_1} \ rel(\mathcal{R}) \|Expr''[\mathbf{reg}(i) \leftarrow Expr_2]\|_{s_0,s_1}$ *is true*
     $\{$ *from (1) , (2) and (3)* $\}$
     $\Longleftrightarrow$
     $\|Expr'\|_{s_0,s_2} \ rel(\mathcal{R}) \|Expr''\|_{s_0,s_2}$
     $\equiv$
     $s_2, s_0 \vDash \psi$

   - the rest of the cases are by structural induction

Lemma 5.1.2 (Update of the heap) *For any expressions* $Expr_1, Expr_2, Expr_3$ *and any field* f *if we have that the states* s_1 *and* s_2 *are such that* $s_1 =<$ $\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and* $s_2 =< \mathrm{H}[\oplus f \rightarrow f[\oplus \|Expr_2\|_{s_0,s_1} \rightarrow \|Expr_3\|_{s_0,s_1}]], \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *the following holds*

1. $\|Expr_1[f \leftarrow f[\oplus Expr_2 \rightarrow Expr_3]]\|_{s_0,s_1} = \|Expr_1\|_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[f \leftarrow f[\oplus Expr_2 \rightarrow Expr_3]] \iff s_2, s_0 \vDash \psi$

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

1. $$\|Expr_1 \begin{array}{l}[\mathbf{st}(\mathbf{cntr}) \leftarrow \mathtt{ref}] \\ [f \leftarrow f[\oplus \mathtt{ref} \rightarrow defVal(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)}\end{array} \|_{s_0,s_1} = \|Expr_1\|_{s_0,s_2}$$

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

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

1. $\|Expr_1[\mathbf{st}(Expr_2) \leftarrow Expr_3]\|_{s_0,s_1} = \|Expr_1\|_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\mathbf{st}(Expr_2) \leftarrow Expr_3] \iff s_2, s_0 \vDash \psi$

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

1. $\|Expr_1[\mathbf{cntr} \leftarrow Expr_2]\|_{s_0,s_1} = \|Expr_1\|_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\mathbf{cntr} \leftarrow Expr_2] \iff s_2, s_0 \vDash \psi$

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

1. $\|Expr_1[\backslash\mathbf{result} \leftarrow Expr_2]\|_{s_0,s_1} = \|Expr_1\|_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\backslash\mathbf{result} \leftarrow Expr_2] \iff s_2, s_0 \vDash \psi$

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

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

5.2 Proof of Correctness

The correctness of our verification condition generator is established w.r.t. to the operational semantics described in Section 1.8. We look only at partial correctness, i.e. we assume that programs always terminate and we assume that there are no recursive methods.

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

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

- $\mathtt{m} : s_0 \Rightarrow s_1$
- $s_0, s_0 \vDash \mathtt{m.pre}$

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

The next issue that is important for understanding our approach is that we follow the design by contract paradigm [8]. This means that when verifying a method body, we assume that the rest of the methods respect their specification in the sense of the previuos definition 5.2.1.

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

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

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

then :

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

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

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

   { *by initial hypothesis* }
   $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp($`m` return $, \mathrm{Pc}_n)$
   { *by definition the weakest precondition for* return }
   $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$ `m.normalPost`$[\backslash \mathbf{result} \leftarrow \mathbf{st}(\mathrm{cntr})]$
   { *by the substitution property 5.1.6* }
   $\Longleftrightarrow$
   $< \mathrm{H}_n, \|\mathbf{st}(\mathrm{cntr})\|_{s_0, <\mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n>} >^{norm}, s_0 \vDash$ `normalPost`
   { *by definition of the evaluation function eval* }
   $\Longleftrightarrow$
   $< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}, s_0 \vDash$ `normalPost`

2. Pc_n : **instr** = load i

$\{$ *by initial hypothesis* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\mathrm{Pc}_n \ \text{load}\ i, \mathtt{m})$
$\{$ *definition of the wp function* $\}$
$\equiv$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})\ [\mathrm{cntr} \leftarrow \mathrm{cntr} + 1]\ [\mathbf{st}(\mathrm{cntr} + 1) \leftarrow \mathbf{reg}(i)]$
$\{$ *applying the substitution properties 5.1.5 and 5.1.4* $\}$
$\Longleftrightarrow$
$< \mathrm{H}_n, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus \mathrm{Cntr}_n + 1 \rightarrow \mathrm{Reg}_n(i)], \mathrm{Reg}_n, \mathrm{Pc}_{n+1} >, s_0 \vDash$
$\mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})$
$\{$ *from the operational semantics of the* load *instruction in section 1.8*$\}$
$s_{n+1}, s_0 \vDash \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})$
$\{$ *and the lemma holds in this case* $\}$

3. new C

$\{$ *by initial hypothesis* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\mathrm{Pc}\ \text{new}\ C, \mathtt{m})$
$\{$ *definition of the wp function* $\}$
$\equiv$

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$
$\forall \mathtt{ref}, \mathit{not}\ \mathbf{instances}(\mathtt{ref}) \wedge$
$\mathtt{ref} \neq \mathbf{null} \Rightarrow$

(1) $\mathit{inter}(i, i+1, \mathtt{m})$ $[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1]$ $[\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathtt{ref}]$ $[f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)}$ $[\backslash \mathbf{typeof}(\mathtt{ref}) \leftarrow C]$

$\{$ *from the operational semantics of* new *in section 1.8* $\}$

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

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

{ *following Def. 3.3.0.2 instantiate (1) with* $\mathtt{ref}'$ }
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$
not **instances**$(\mathtt{ref}') \wedge$
$\mathtt{ref}' \neq \mathbf{null} \Rightarrow$

(4) $inter(i, i+1, \mathtt{m})$ $[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1]$ $[\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathtt{ref}']$ $[f \leftarrow f[\oplus \mathtt{ref}' \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)}$ $[\backslash \mathbf{typeof}(\mathtt{ref}) \leftarrow C]$

{ *from (3)* }

(5) $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$ *not* **instances**$(\mathtt{ref}') \wedge$ $\mathtt{ref}' \neq \mathbf{null}$

{ *from (4) and (5) and Def. 3.3.0.2* }
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$

$inter(i, i+1, \mathtt{m})$ $[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1]$ $[\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathtt{ref}']$ $[f \leftarrow f[\oplus \mathtt{ref}' \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}\ (f.\mathsf{declaredIn}, C)}$ $[\backslash \mathbf{typeof}(\mathtt{ref}) \leftarrow C]$

{*from lemmas 5.1.5, 5.1.4 and 5.1.2, 5.1.3*
and the operational semantics of the instruction new }
$s_{n+1}, s_0 \vDash inter(i, i+1, \mathtt{m})$

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

{ *by initial hypothesis* }
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\ \mathrm{Pc}_n\ \ \text{putfield}\ f, \mathtt{m})$
{ *definition of the wp function* }
$\equiv$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$
$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow$

(1) $inter(i, i+1, \mathtt{m})$ $[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2]$ $[f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]$

$\wedge$
$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPost}(i,\ \mathtt{NullPntrExc})$
{ *we get three cases* }

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

`instr` is in its scope

$$
\begin{array}{l}
\{\ \textit{thus, we get the hypothesis}\ \} \\
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \\
\{\ \textit{from the above conclusion and (1) we get}\ \} \\
< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathtt{m}.\mathsf{excPost}(\mathrm{Pc}_n,\ \mathtt{NullPntrExc}) \\
\{\ \textit{from Def. 4.3.2.2 of the function}\ \mathtt{m}.\mathsf{excPost} \\
???\ \textit{and the assumption that the exception is handled we get}\ \} \\
\quad < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \\
\quad\ \forall \mathtt{ref}, \\
\qquad \neg\ \mathbf{instances}(\mathtt{ref}) \wedge \\
\qquad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\qquad\quad \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m}) \\
\qquad\quad [\mathbf{cntr} \leftarrow 0] \\
\qquad\quad [\mathbf{st}(0) \leftarrow \mathtt{ref}] \\
\qquad\quad [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathit{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\ \mathsf{subtype}\ (f.\mathsf{declaredIn},\ \mathtt{Exc})} \\
\{\ \textit{from lemmas 5.1.5, 5.1.2, 5.1.4 and 5.1.3} \\
\quad \textit{and the operational semantics of}\ \ \mathrm{putfield}\ \} \\
s_{n+1}, s_0 \vDash\ \ \mathit{inter}(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})
\end{array}
$$

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

$$
\begin{array}{l}
\quad < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \\
\quad\ \forall \mathtt{ref}, \\
\qquad \neg\ \mathbf{instances}(\mathtt{ref}) \wedge \\
\qquad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\quad\ \mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{NullPntrExc}) \\
\qquad [\backslash \mathbf{EXC} \leftarrow \mathtt{ref}] \\
\qquad [f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathit{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\mathsf{subtype}\ (f.\mathsf{declaredIn},\ \mathtt{Exc})} \\
\{\ \textit{from lemmas 5.1.4, 5.1.2, 5.1.3 and} \\
\quad \textit{the operational semantics of}\ \ \mathrm{putfield}\ \} \\
s_{n+1}, s_0 \vDash \mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{NullPntrExc})
\end{array}
$$

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

$$
\begin{array}{l}
\quad \{\ \textit{thus, we get the hypothesis}\ \} \\
\quad < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, s_0 \vDash \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \\
\quad \{\ \textit{from the above conclusion and (1) we get}\ \} \\
\quad\ < \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, s_0 \vDash \\
\quad\ \mathit{inter}(i, i+1, \mathtt{m})\ \begin{array}{l}[\mathbf{cntr} \leftarrow \mathbf{cntr} - 2] \\ [f \leftarrow f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]\end{array} \\
\quad \{\ \textit{applying lemmas 5.1.5 and 5.1.2 and} \\
\quad\ \textit{of the operational semantics of}\ \ \mathrm{putfield}\ \} \\
\quad s_{n+1}, s_0 \vDash\ \ \mathit{inter}(i, i+1, \mathtt{m})
\end{array}
$$

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

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

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

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

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

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

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

We consider three cases: the case when the next execution step doesnot enter a cycle (the next instruction is not a loop entry in the sense of Def.1.9.2) the case when the current instruction is a loop end and the next instruction to be executed is a loop entry instruction (the execution step is $\rightarrow_l$) and the case when the current instruction is not a loop end and the next instruction is a loop entry instruction (corresponds to the first iteration of a loop)

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

{ *following Def. 4.3.1 of the function inter in this case* }
(1) $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m}) = wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$
{ *by initial hypothesis* }
(2) $s_n, s_0 \vDash wp(\ \mathrm{Pc}_n : \mathtt{instr}, \mathtt{m})$
{ *from the previous lemma 5.2.1 and (2) , we know that* }
(3) $s_{n+1}, s_0 \vDash inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$
{ *from (1) and (3)* }
$s_{n+1}, s_0 \vDash wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m})$

2. Pc_n : `instr` is not a loop end and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of bytecode instructions of the method `m`(i.e. the execution step is of kind $\rightarrow^l$, see Def.1.9.2). Thus, there exists a natural number $i, 0 \leq i < \mathtt{m.loopSpecS}.length$ such that $\mathtt{m.loopSpecS}[i].\mathsf{pos} = loopEntry$, $\mathtt{m.loopSpecS}[i].\mathsf{invariant} = I$ and $\mathtt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$. We look only at the case when the current instruction is a load instruction

$\{$ *by initial hypothesis* $\}$
$s_n, s_0 \vDash wp(\ \mathrm{Pc}_n\ \ \mathrm{load}\ i, \mathtt{m})$
$\{$ *by defintion of the wp function in section 4.3 of the previous chapter* $\}$
$s_n, s_0 \vDash \ inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})\ \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(j)]\end{matrix}$
$\{$*by the definition 4.3.1 for the case when*
the execution step is not a backedge but the target instruction is a loop entry $\}$
$s_n, s_0 \vDash$
$I\ \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(i)]\end{matrix}$
$\wedge$
$\forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m}))\ \begin{matrix}[\mathbf{cntr} \leftarrow \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \leftarrow \mathbf{reg}(i)]\end{matrix}$
$\{$ *from lemmas 5.1.5 and 5.1.4* $\}$
$\Longleftrightarrow$
$s_n\ \begin{matrix}[\mathrm{Cntr} \leftarrow \|\mathbf{cntr} + 1\|_{s_0,s_n}] \\ [\mathrm{St} \leftarrow \mathrm{St}[\oplus(\|\mathbf{cntr} + 1\|_{s_0,s_n}) \rightarrow \|\mathbf{reg}(i)\|_{s_0,s_n}]]\end{matrix}\ , s_0 \vDash$
$I \wedge$
$\forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m}))$
$\{$ *from the Def. 3.3 of the evaluation function* $\}$
$\equiv$
$s_n\ \begin{matrix}[\mathrm{Cntr} \leftarrow \mathrm{Cntr} + 1] \\ [\mathrm{St} \leftarrow \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \mathrm{Reg}(i)]]\end{matrix}\ , s_0 \vDash$
$I \wedge$
$\forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m}))$
$\{$ *from the operational semantics of* load $\}$
$s_{n+1}, s_0 \vDash \begin{matrix}I \wedge \\ \forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1} : \mathtt{instr}, \mathtt{m}))\end{matrix}$
$\{$ *we can get from the last formulation* $\}$
(1) $s_{n+1}, s_0 \vDash I$

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

3. Pc_n : `instr` is an end of a cycle and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of bytecode instructions of the method `m`(i.e. the execution step is of kind $\rightarrow^l$

). Thus, there exists a natural number $i, 0 \leq i < \texttt{m.loopSpecS}.length$ such that $\texttt{m.loopSpecS}[i].\mathsf{pos} = loopEntry$, $\texttt{m.loopSpecS}[i].\mathsf{invariant} = I$ and $\texttt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$. We consider the case when the current instruction is a sequential instruction. The cases when the current instruction is a jump instruction are similar.

{ *by hypothesis we get* }

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

{ *from Def. 4.3.1 and transformation over the above statement* }

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

{ *by hypothesis, loopEntry* $= \mathrm{Pc}_{n+1}$. *From def. 1.9.2, we conclude that there is a prefix subP* $= \texttt{m.body}[0] \rightarrow^* loopEntry : \texttt{instr}$ *of the current execution path which does not pass through* $\mathrm{Pc}_n : \texttt{instr}$. *We can conclude that the transition between loopEntry* : `instr` *and its predecessor k* : `instr` *(which is at index k in* `m.body`*) in the path subP is not a backedge. By hypothesis we know that* $\forall i, 0 \leq i \leq n, s_i, s_0 \vDash wp(\ \mathrm{Pc}_i : \texttt{instr}, \texttt{m})$. *From def.4.3.1 and lemma 5.2.1 we conclude* }

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

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

$$(3) \quad s_k =_{\mathsf{modif}} s_{n+1}$$

{ *because* $\texttt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$ *and from (2) and (3)* }

$$(4)\ s_{n+1}, s_0 \vDash I \Rightarrow wp(\ loopEntry : \texttt{instr}, \texttt{m})$$

{ *from (1) and (4)* }

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

Qed.

Lemma 5.2.3 (*wp* of method entry point instruction) *Assume we have a method* `m`. *Assume that execution of method* `m` *starts execution in state* s_0 *and* $s_0, s_0 \vDash wp(\ \texttt{m.body}[0], \texttt{m})$ *where and makes n steps to reach state* s_n*:* $s_0 \hookrightarrow^n s_n$, *then*

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

Proof : Induction over the number of execution steps n

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

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

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

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

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

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

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

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

Theorem 5.2.5 *For any method* m *if the verification condition is valid:*

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

then m *is correct in the sense of the definition 5.2.1.*

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

Bibliography

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

[2] Fabian Bannwart. A logic for bytecode and the translation of proofs from sequential java. Technical report, ETHZ, 2004.

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

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

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] Draft Revision December. Jml reference manual.

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

[15] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[16] 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.

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

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

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

[20] 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.

[21] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[22] "K. Rustan M. Leino, Greg Nelson, , and James B. Saxe ". Esc/java user's manual.

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

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

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

[26] M.Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA.

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

[28] 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.

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

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

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

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