Thèse de Doctorat

Préparée pour obtenir le titre de
Docteur en Sciences de l'Université de Nice – Sophia Antipolis

Discipline : Informatique

Préparée à l'Institut National de Recherche en Informatique
et en Automatique de Sophia Antipolis

par

Mariela PAVLOVA

Vérification de bytecode et ses applications

Directeurs de thèse : Gilles BARTHE et Lilian BURDY

Soutenue publiquement le 19 janvier 2007
à l'École Supérieure en Sciences Informatiques
de Sophia Antipolis

devant le jury composé de :

MM.	Pierre PARADINAS	Président	CNAM Paris
	Claude MARCHE	Rapporteur	Inria Futurs - ProVal
	John HATCLIFF	Rapporteur	Kansas State University
	David NAUMANN	Rapporteur	Stevens Institute of Technology
MM.	Gerwi KLEIN	Examinateur	University of New South Wales
	Gilles BARTHE	Directeur de thèse	INRIA Sophia Antipolis

Dans cette thèse, nous proposons une architecture qui permet la vérification de code mobile respectant des politiques de sécurité potentiellement complexes. En particulièr, cette architecture est une version du paradigme du code contenant ses preuves proposée par George Necula. Dans cette dernière, le producteur de code mobile utilise le compilateur certifiant pour générer le code éxécutable ainsi qu'un certificat qui montre que le code satisfait les politiques de sécurité du client qui éxécute le code mobile. Le compilateur certifiant étant automatique, seules des politiques de sécurités décidables peuvent être traitées, telles la correction du typage, ou l'accès sécurisé à la mémoire. Le développement rapide de technologies dans des domains critiques comme les cartes à puce requiert des politiques de sécurités plus complexes (par exemple des propriétés fonctionelles) que les politiques de sécurités décidables.

Pour repondre à ce besoin, dans cette thèse nous introduisons les composants suivants: un langage de Modélisation de Bytecode dérivant de JML (Java Modeling Language), un compilateur des annotations du code source au code compilé, un générateur de conditions à vérifier pour le code compilé Java, et l'équivalence entre les obligations de preuve entre le code source et le code compilé. Basée sur ces composants, notre architecture permet au producteur de code de générer le certificat de code mobile interactivement et donc permet de certifier des propriétés qui vont au delà des propriétés décidables.

Mots Clés : program static verification, verification condition generator, program logic, mobile code, PCC, certified code, Java, bytecode, proof preserving compilation, specification languages

Contents

Chapter 1

Introduction

Since the beginning of programming languages, the problem of program correctness has been an issue. Especially, this is the case today when complex software applications enter into our daily life and perform actions which may potentially compromise confidential data (credit cards) or the performance of the computer system they are executed on (e.g. installing software which may format the hard disk of our PC). Also, errors in software conception may have disastrous effects, for instance the crash of the European Ariane 5 launcher due to execution of a data conversion. More over, software correctness plays a crucial role in the overall software quality in cases the latter take part in the automatic control of engines which may put in danger lives, like airplanes or subways.

Thus, a necessary phase in the program development is the validation of the program implementation, during which the producer checks if the written code conforms to her/his intentions.

A widely used approach is program testing. However, this technique is not capable to cover all the cases of the input space and thus, although the tests have been successful, the implementation may still contain errors. This, of course, is not desirable in particular in the case of critical program applications as is the case for applications which target the domains mentioned above.

Another possibility is to use formal verification methods. Contrary to program testing, formal program verification guarantees that if the verification procedure has been successful then the implementation respects the initial requirements for the software. The field of formal verification is vast and there exist different approaches as for instance model checking, type systems or program logic. In this thesis, we shall focus on the last approach. In Fig. 1.1, we give a general outline of the architecture for verifying source code programs.

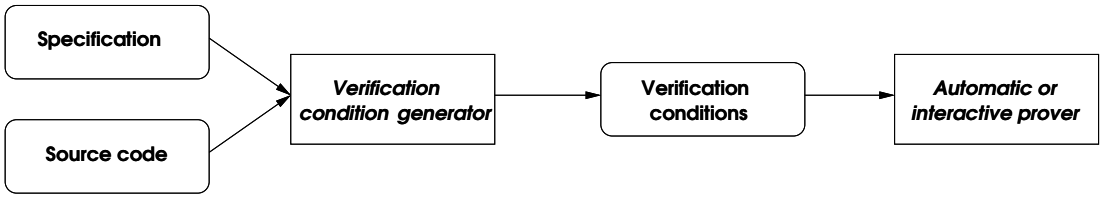

Figure 1.1: Source verification

As we can see from the figure, the architecture relies on the following components:

- specification of the implementation which expresses the intended behavior of the source program. The annotations are written in a formal specification language.

- an algorithm which generates logical (verification) conditions from the annotations and the program. Such an algorithm is usually called verification condition generator. The generated verification conditions are such that their truth guarantees that the implementation respects the program specification and thus, that the implementation respects its intended behavior.

- a system for showing the validity of the verification conditions. This is normally done either by an automatic decision procedure or interactive theorem prover. Automatic decision procedures do not need user interaction. However, automation cannot deal with difficult theorems because of the undecidability of the logic. Interactive prove assistant is an alternative which is more powerful and can more often "say" if something holds. However, even interactively it could be difficult to show that a theorem holds. Although interactive prove assistants systems are more reliable than automatic decision procedures they need a user interaction and expertise in theorem proving and logic. Thus, depending on the complexity of the verification conditions, an automatic decision procedure or an interactive system is used.

The field of formal verification is well studied and several tools exist for dealing with source verification using program logic. We should mention in the list of the verification tools tailored to source languages the Caveat[1] verification framework for C, Caduceus[2] for C, the extended static checker esc/java [72], the Loop tool [60], Krakatoa [76], Jack [28], etc. These tools are tailored to a source language and thus, are helpful for the development process. For instance, the developer may use such verification tools for discovering program bugs and errors. More generally, verification on source code allows the code producer to audit the quality of a software implementation.

Although program logics guarantees source program correctness, it does not say anything about the executable code resulting from the source compilation. Thus, source verification requires a trust in the compiler. However, such a compromise is not always admissible.

This is in particular the case for mobile code scenarios where a code client executes an unknown or untrusted application. Fig. 1.2 shows a schemata of mobile code scenarios. From the figure, we can see that in such scenarios there are two counter parts: a code producer and a code client. The producer generates the code and the client receives the generated code. However, the client normally does not have any evidence that the code respects his requirements.

Figure 1.2: Mobile code

Examples for a mobile code scenario can be the downloading of software components via internet and their execution on a user system which can be a personal computer, PDAs, embedded systems, etc. These software components could be from user specific applications to patches for the operating system.

Thus, it is more suitable to perform the verification directly on *the executable or interpreted code*. First, because the executable is normally not accompanied by the corresponding source code and second, even if it were this would require trust in the compiler. This would mean that the client should have a similar verification infrastructure as on source code as shown in Fig. 1.3.

[1] http://www-list.cea.fr/labos/fr/LSL/caveat/index.html
[2] http://caduceus.lri.fr

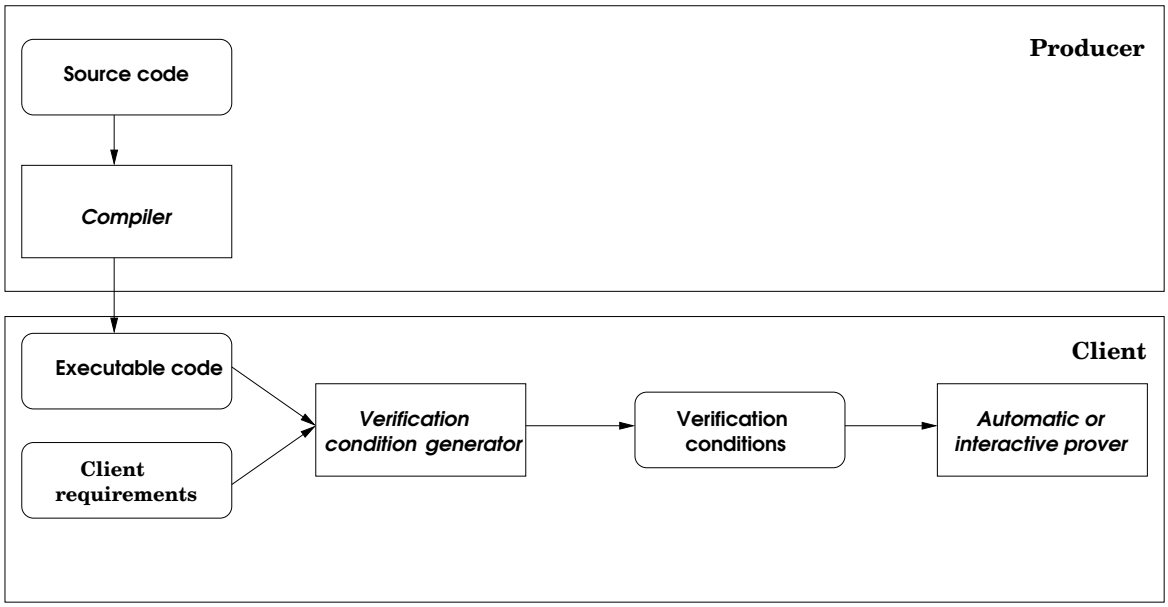

Figure 1.3: Mobile code verification

Providing mechanisms for verification over the executable or interpreted code gives the client the possibility to check if the application respects the requirements of the client system. However, the process of theorem proving is an expensive computational procedure and the client system may potentially not be willing to slow down its execution for verifying the unknown code. Especially, this is the case for small devices which rely on limited computational resources.

The Proof Carrying Code paradigm (PCC) and the certifying compiler proposed by G.Necula [86] gives a solution to this problem. In the traditional PCC architecture, untrusted code is accompanied by a proof which certifies that the code is in compliance with the client requirements. In Fig. 1.4, we show the basic components of such framework and the relation between them. The code producer uses the so called certifying compiler. The certifying compiler consists of a compiler and a certifier. The compiler not only compiles source programs into executable (or interpreted bytecode) but also generates annotations for the executable code. The certifier assembles the verification condition generator over the executable and invokes an automatic decision procedure which will generate a formal proof for the verification conditions. The producer sends the executable code with annotations and the proof to the client. The client then does not have to prove the program. Rather, he generates the verification conditions over the received code and his requirements and then checks if the proof accompanying the code is a proof for the generated verification conditions. Checking a proof w.r.t. a logical statement is much easier then finding its proof. For this, type checking algorithms are used as the verification conditions are interpreted as types and the proofs are interpreted as expressions. The complexity of type checking algorithms is small and thus, do not compromise the performance of the client system.

However, because traditional PCC is completely automated, it can only deal with safety properties, like well typedness or safe memory read and write access. These properties are very important as they guarantee that the execution of the whole system will not be violated by the untrusted application. However, restricting the verification mechanism only to safety properties is not satisfactory, because there are potentially situations in which the client security policy may be more complex. For instance, the client may have functional requirements over the unknown code. An example is when an updated version of an interface implementation is downloaded on the client computer system and patches an old implementation of the interface. In this cases, the client system needs guarantees that the unknown software component respects the interface specification. In such situations, traditional PCC is not appropriate as it will fail to generate the proof automatically.

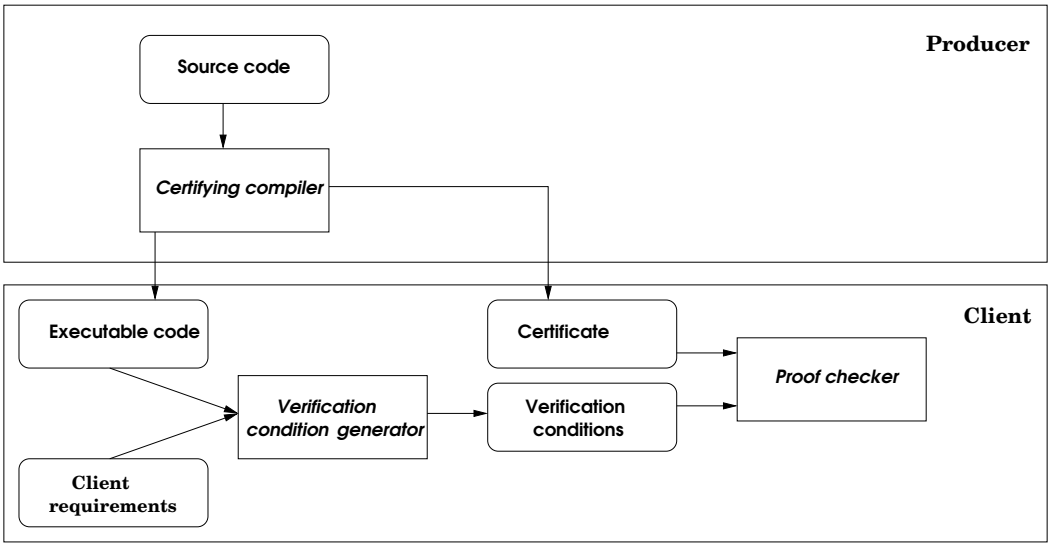

Figure 1.4: Proof carrying code architecture for mobile code

A solution to this problem is to give up automation in the certificate generation and thus, make the process of the proof generation on the producer site interactive as shown in Fig. 1.5. As you can see in the figure, we propose that the producer site *generates the certificate interactively over the source code*. The advantage of such an approach is that the interactive proof will allow to identify bugs in the source code and correct them until the source becomes conform with the client requirements. Moreover, interactive verification over source code allows to identify points in the source program which must be enforced with annotations (typically, loop invariants) that will allow the verification conditions to get provable. However, there are several points which need to be clarified. The first one is how the client may use a source code certificates for the verification of its executable counterpart. This is actually possible as verification conditions over source and low level code are equivalent if the low level code has been produced by a non optimizing compiler. However, this equivalence is not sufficient to make things work. In particular, sending the executable code and its certificate is not sufficient for the client to check correctly the program. As we said above, the generation of the certificate on the producer site, requires additional annotations. Recall that the validation of the executable program on the client side involves the generation of the verification conditions again. If the client does not have the annotation enforcement for the program, he will not succeed to type check the verification conditions that he generated against the received certificate, even if the program is correct. For this, we propose a compiler from source annotations to annotations on the level of the executable (interpreted) code.

In the following, we present a framework which follows the schemata from Fig. 1.5 which allows to verify statically bytecode programs against potentially non trivial functional and security properties. The framework is tailored to Java bytecode.

The Java platform of Sun Microsystems has gained a lot of popularity in industry for the last two decades. The reasons for its success is that it allows applications to be developed in a high-level language without committing to any specific hardware and since it features security mechanisms which guarantee the innocuousness of downloaded applications. It guarantees type safety via the bytecode verifier and restricts the access of untrusted applications to systems resources trough the sandbox mechanism. The JVM provides an automatic memory management through a garbage collector.

For instance, smart card applications are relying on JavaCard, a dialect of Java tailored to small devices. Such devices, however, impose security restrictions to the software components they run for which platforms like Java do not provide sufficient mechanisms for their guarantee, e.g. preserving of confidentiality or limited use of computational resources. Moreover, the new generation of smart card platforms where installation of software components after the card has

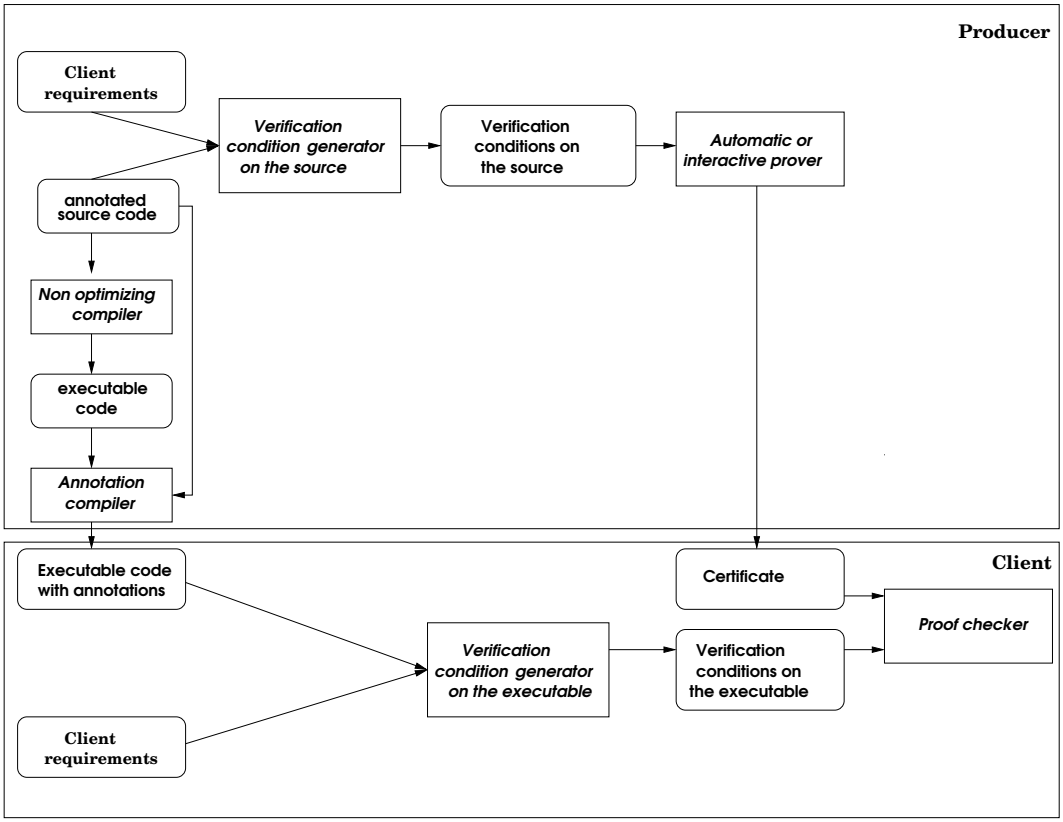

Figure 1.5: Proof preserving compilation for mobile code

been issued is possible, opens new security problems. Another example is the J2ME (which stands for Java 2 Micro Edition) which is widely used in mobile phone software industry. Mobile phone users benefit today from the possibility to install new software components on the phone system by the wireless net, however no mechanisms are provided by the Java system to check that they are not malicious, buggy or incompatible with other installed code.

1.1 Contributions

We propose a framework which allows the verification of bytecode programs against complex functional and safety properties, which has the following components.

Bytecode Modeling Language We define a specification language for bytecode called BML (short for Bytecode Modeling Language). BML is the bytecode encoding of an important subset of JML (short for Java Modeling Language). The latter is a rich specification language tailored to Java source programs and allows to specify complex functional properties over Java source programs. Thus, BML inherits the expressiveness of a subset JML and allows to encode potentially complex functional and security policies over Java bytecode programs. We define an encoding of BML in the class file which is in conformance with the Java Virtual Machine specification [75]. To our knowledge, BML does not have predecessors that are tailored to Java bytecode.

Verification condition generator for Java bytecode We propose a verification condition generator (VcGen for short) for Java bytecode which is completely independent from the source code. The verification condition generator covers a large subset of Java and deals with arithmetic operations, object creation and manipulation, method invocations, exception throwing and handling, stack manipulation etc. The verification condition relies on a weakest predicate

transformer function adapted for Java bytecode and we have proved its soundness relative to a Java operational semantics. We have an implementation which is integrated in Jack (short for Java Applet Correctness Kit) [28] which is a user friendly plugin for the eclipse ide [3].

Compiler from source to bytecode annotations We define a compiler from JML to BML which allows Java bytecode benefit from the JML source specification. The compiler does not depend on a particular Java compiler but requires it to be non optimizing The JML compilation results in enriching the class file with the BML encoding of the specification.

Equivalence between source and bytecode proof obligations Such an equivalence is useful when programs and requirements over them are complex. In this case, an interactive verification procedure over the source code could be helpful. (e.g. proving the verification conditions in an interactive theorem prover). First, interactive procedure is suitable where automatic decision procedures will not cope with difficult theorems which is potentially the case for sophisticated security policies or functional requirements. Second, using verification on source code is useful as program bugs and errors can be easily identified and corrected. Because of the relative equivalence between source and bytecode proof obligations, once the verification conditions over the source and the program requirements expressed as specifications are proved the bytecode and the certificate (the proof of the verification conditions) produced over the source can be shipped to the client.

We have shown the usefulness of bytecode verification condition generator in two cases. The first one is the verification of constrained memory consumption policies. Indeed, it is an important issue for devices with limited resources as smart cards to have a guarantee that a newly installed application will not corrupt the whole system because of malevolent resource usage. In such critical situations, bytecode verification is suitable as it does not compromise the compiler. This work was published in [20].

We have also shown how this methodology can be applied to bytecode to native compiler optimizations. The speed and small memory footprint size of executable code is important especially for constrained devices. In order to speed up the execution Java-to-native compilation can be done. But this comes on the price of a large memory footprint. We apply our verification framework for decreasing the size of the native code size by identifying check sides in the native code which can be removed. This gave issue to the publication [33].

1.2 Plan of the thesis

The present thesis is organized as follows. In the next Chapter 2, we give an overview of the basic ideas in Java source verification. We present there the JML language, a Java like source language and verification condition generator for the source language. We also discuss different approaches in program verification using program logic. Chapter 3 presents the bytecode language and its operational semantics which will be used all along the thesis for the definition of the verification condition generator and the proof of its soundness. Chapter 4 presents the syntax and semantics of the Bytecode Modeling Language (BML) and the compiler from JML to BML. In Chapter 5, we turn to the definition of the verification condition generator and Chapter 6 presents the proof of its soundness. In Chapter 7, we shall focus on the relation between verification conditions on source and bytecode level and we shall see under what conditions they are the same modulo names and basic types. Chapter 8 presents an application of the bytecode verification condition generator and BML to the verification of constraint memory consumption policies. Chapter 9 shows how we can build a Java-to-Native compiler using our verification framework.

[3] http://www.eclipse.org/

Chapter 2

Java verification overview

The purpose of this chapter is to remind the reader the basic principles in the formal verification of Java programs. Because the concepts on Java source and bytecode verification are similar, we have decided to give here an overview of them on Java source. We hope that this will be a gentle introduction to the rest of the thesis especially for those which are not acquainted with Java bytecode.

As we already stated in the introductory chapter, a formal program verification relies on three elements: a specification language in which to express the requirements that a program must satisfy, a verification procedure for generating verification conditions whose validity implies that the program respects its specification and finally, a system to decide the validity of the verification conditions. In this chapter and in the rest of thesis, we shall focus on the first two components. In particular,in Section 2.1, we shall present a Java like programming language supporting the most important features of Java. We also present JML, the de facto Java specification language tailored to Java. This will be done in Section 2.2. Section 2.3 presents a discussion about the different approaches for program verification using program logic. In Section 2.4, we shall define over it a verification condition calculus.

2.1 Java like statements and expressions

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) and constructors."*. Fields represent the state of an object of the class where those fields are declared. Fields have a name and a type. Thus, a field declaration in a class states the name and type of the field. Methods are the constructs in a class which execute and allow to change the state of the objects at runtime. Class constructors are special methods which initialize a new object of this class. Constructors have the same name as the class to which they belong. A method declaration includes the method name, the list of arguments which specifies their names and types, the method return type as well as the method body. A method body in Java represents a Java statement. Statements are program constructs whose role is to determine the control flow of the program during execution. Statements manipulate program constructs which represent values. Those constructs are the expressions of the language. Expressions do not determine the control flow but they evaluate to values. Values in our language are either references or integers. For an illustration, consider the class declaration:

```
1  class A {
2    int f;
3
4    A(int c) {
5     f = c;
6    }
7
8    int eqField ( A a ) {
```

```
 9      if (a != null) {
10        return 0;
11      }
12      if ( a.f == this.f ) {
13        return 1;
14      } else {
15        return 0;
16      }
17    }
18  }
```

The example shows the declaration of class A. The class first declares a field named `f` of type integer (line 2). Then follows the declaration of the class constructor which initializes the field `f` to the value of the parameter c (lines 4-6). The last component of the class is the method `eqField` which tests if the parameter a of type A is equal to the receiver of the method call by returning 0 or 1 (lines 8-18). Thus, the body of `eqField` first checks if a is not **null** and if it is returns 0. Otherwise, if the field `f` in the current object and in the object passed as the parameter a are equal then the method returns 1 and if not returns 0.

In the following, we shall concentrate in more detail on the statements and expressions typical for object oriented languages e.g. object manipulation and creation, method invocation, throwing and handling exceptions and subroutines. Fig. 2.1 gives the formal grammar.

$$
\begin{array}{lll}
E & ::= & \mathit{IntConst} \\
 & | & \textbf{null} \\
 & | & \textbf{this} \\
 & | & E\ op\ E \\
 & | & E.\texttt{f} \\
 & | & \mathit{Var} \\
 & | & (\texttt{Class})\ E \\
 & | & E.\texttt{m}() \\
 & | & \textbf{new}\ \texttt{Class}() \\
 & & \\
 & & op \in \{+, -, *, \} \\
 & & \\
E^{cond} & ::= & E\ cond\ E \\
 & & \\
 & & cond \in \{\leq, <, \geq, >, =, \neq\} \\
 & & \\
S & ::= & S; S \\
 & | & \texttt{if}\ (E^{cond})\ \texttt{then}\ \{S\}\ \texttt{else}\ \{S\} \\
 & | & \texttt{try}\ \{S\}\ \texttt{catch}\ (\texttt{Exc}\ )\ \{S\} \\
 & | & \texttt{try}\ \{S\}\ \texttt{finally}\ \{S\} \\
 & | & \texttt{throw}\ E \\
 & | & \texttt{while}\ (E^{cond})[\texttt{INV}, \texttt{modif}]\ \ \{S\} \\
 & | & \texttt{return}\ E \\
 & | & E = E
\end{array}
$$

Figure 2.1: Source language

The nonterminal S introduces the grammar for statements. The statements that are supported are standard control flow constructs like compositional statement, conditional statement, assignment statement, return statement etc. They have the usual semantics of programs with exceptional

and normal termination. For instance, the meaning of the compositional statements $S; S$ is that if the first statement terminates execution normally then the second statement is executed. Moreover, if the first or second statement terminates on exception then the whole statement terminates on exception

The construct `try` $\{S\}$ `catch` (`Exc`) $\{S\}$ allows for handling exceptions. Its meaning is that if the statement following the `try` keyword throws an exception of type `Exc` then the exception will be caught by the statement following the `catch` keyword. The language also supports try finally statements, a construct which is particular to the Java language. The meaning of the construct is that no matter how the statement following the keyword `try` terminates (on an exception or normally), the statement introduced by the keyword `finally` must execute after it. If the try block terminates normally then the whole try statement will terminate as the finally block. If the try block terminates on exception Exc and if the finally block terminates normally, the whole try finally statement terminates on the exception Exc. If the try block terminates on exception Exc and if the finally block terminates on exception Exc' then the whole try finally terminates on exception Exc'. Loop statements are also supported. Note that their syntax includes a formula `INV` which must hold whenever the loop entry is reached as well as a list of expressions `modif` which lists the locations that may not have the same value at the beginning and at the end of a loop iteration. Note that a variable should not be in the list `modif` even if during a loop iteration its value is changed as far as at the end of the iteration the value it had in the beginning of the iteration is restored.

Let us now turn to the expression grammar. As we can see from the figure, the language supports integer constants *IntConst*, the null constant **null** denoting the empty reference, a construct **this** for referring to the current object, arithmetic expressions E *op* E where $op \in \{+, -, div, rem, *\}$. The value stored in a field named `f` for the object reference E is denoted with the construct E.`f`, cast expressions with (`Class`)E and method local variables and parameters with identifiers taken from the set *Var*. The language also has constructs for expressing method invocation. Thus, the first expression in the syntax of the method invocation E.`m`() stands for the receiver object of the method call and `m` is the name of the invoked method. For the sake of clarity we consider only non void instance methods which does not receive parameters. Moreover, we assume that methods always return a value. The language supports also instance creation construct. Note that constructors like methods do not take arguments. The semantics of instance creation expression **new** `Class` is that it creates a new reference of type `Class` in the heap which is initialized by the class constructor `Class`. For class constructors, we also assume that they do not take parameters for the same reasons as above. A relation between expressions is denoted with E *cond* E where $cond \in \{\leq, <, \geq, >, =, \neq\}$. We could have considered a larger set of boolean expressions like for instance logical connectors $\wedge, \vee \ldots$ but we limit our language only to relational expressions for the sake of simplicity without losing any particular feature of the language.

2.2 Overview of JML

JML [67] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications which follows the design-by-contract approach (see [23]).

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. [25]). 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 [32] created by Yoonsik Cheon and G.T. Leavens. 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 [72] 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

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

Nijmegen, the Jack tool developed at Gemplus, the Krakatoa tool developed by the ProVal group at Inria, France. The tool Daikon [41] 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 [27]. In the following, we shall proceed with the description of the basic features of JML which correspond basically to the JML Level0 subset of JML.

2.2.1 JML expressions

Expressions manipulated in JML 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. 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, **typeof(expression)** which stands for the dynamic type of **expression**, etc.

2.2.2 Method contracts and method intra specification

Method contracts consists of the conditions upon which relies and the conditions that the method guarantees. The conditions upon which a method relies are expressed in specifications as the method precondition, i.e. the method requires them to hold in its pre state. The conditions which a method guarantees are expressed in specifications as the method postcondition, i.e. they express what the method guarantees to the methods that may invoke it. Method contracts are visible by the other methods in the program as they inform the other methods what they must establish when calling them current method and what the current method guarantees them. Moreover, JML provides a third component in the method contract, the so called frame condition which introduces the locations that can be modified by the method. Frame conditions are actually desugared as part of the postcondition as we shall see later in subsection 2.2.5. For an insight information on method contracts, the reader may refer to [23].

Specifying methods may involve writing specification which is not visible by "the outside world" (the other methods in the program) but which are useful for the program verification process. An example for such a specification are the loop invariants, which express a property which holds at the borders of a loop iteration. Moreover, here we shall allow that a loop is also accompanied by a frame condition which as in the case of method frame conditions specifies which locations may be modified in a loop. Such an extension of JML is used in the tool Jack [28]. We provide a more insight discussion about them later in subsection 2.2.5.

For introducing method preconditions, postconditions and method frame conditions in JML the keywords **requires**, **ensures** and **modifies** keywords are used respectively. Also for loop invariants and loop frame conditions, the keywords **loop_invariant** and **loop_modifies** are used. Fig. 2.2 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 occurrence 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 && list[k] == \old(list[k]));
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 2.2: CLASS ListArray WITH JML ANNOTATIONS

2.2.3 Ghost variables

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.

We illustrate the utility of such variables through an example. Let us have a class `Transaction` which manages transactions in the program as shown in Fig. 2.3. The class is provided with a method for opening transactions (`beginTransaction`) and a method for closing transactions (`commitTransaction`). Suppose that we want to guarantee that in an application there are never nested transactions. A possibility is to declare a method `getTransactionDepth` which counts the number of open transactions and use dynamic checks for guaranteeing that there are no nested transactions. But this would require to modify the code which is not desirable. Another approach would be to use the method `getTransactionDepth` as far as its execution does not affect the program state (i.e. it is pure) in the specification e.g. the precondition of method `beginTransaction` will be `//@ requires getTransactionDepth() == 0`. However, the treatment of pure methods is complicated and its semantics is still not well established. Ghost variables can be used for specifying the no nested transaction property in a simple way. The specification of the methods `beginTransaction` and `commitTransaction` models the property for no nested transactions via the ghost variable `TRANS` (declared on line 3) which keeps track if there is a running transaction or not. In particular, 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. 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 is a simple way for specifying this property without modifying the code.

```
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 }
```

Figure 2.3: specifying No Nested Transaction property with ghost variable

2.2.4 Light and heavy weight specification. Multiple specification cases

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. 2.2. 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 detailed method specifications which allow to describe what are the different cases on which a method might be called and what is the behavior of the method in every case. This specification style is called a *heavy* weight specification. Heavy weight specification are 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 [66]).

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.

An example for a *heavy* weight specification is given in Fig. 2.4. 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 omitted 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.

In a light weight specification style the specifier might omit some of the specification. For instance, in Fig. 2.4, the constructor C is provided only with a precondition. On verification time, the missing part of an incomplete light weight specification is set to its default values. The default value for omitted precondition and postcondition in a light weight specification is **true** and for frame condition is **nothing**[2].

A *heavy* weight specification is a syntactic sugar which is suitable for the specification process and makes easier the understanding of the specified code. The verification of a heavy weight spec-

[2]the default value for frame conditions may depend on our choice. For instance, in Jack the developer can set the default value for frame conditions to **nothing** or **everything**

```
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 2.4: An example for a method with a heavy weight specification in JML

ification involves appropriate desugaring of such specification corresponding to its semantics. The meaning of **normal_behavior** 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**. Similarly, an **exceptional_behavior** clause guarantees that if its precondition holds then the method terminates on exception and the specified exceptional postcondition will hold, i.e. its **ensures** clause is always **false**. In case a method has several specification cases, then one of the preconditions of its specification cases must hold when the method starts execution. Moreover, if the precondition of a particular method specification case holds in the method prestate then the postcondition of this specification case must hold.

Fig. 2.5 shows the desugared version of the method specification from Fig. 2.4. As we can see, the specification of the constructor C contains all the specification clauses, where the postcondition is set to the default postcondition **true**. The specification of the method divide does not contain the syntactic sugar **normal_behavior** and **exceptional_behavior** but explicitly specifies the behavior of the method in the two cases. Particularly, it contains now two specification cases surrounded by the tags {| |}. The first one corresponds to the **normal_behavior** case, i.e. the exceptional postcondition is set to **false** and the second specification case corresponds to the **exceptional_behavior**, i.e. the normal postcondition is set to **false**.

2.2.5 Frame conditions

As we can see in Fig. 2.4 JML allows for specifying a modifies clause for methods. The **modifies** clause or also the frame condition declares which are the locations that a method may modify. The example in Fig. 2.2 shows that we also allow frame condition in the loop introduced by the keyword **loop_modifies**. The semantics of the method and loop frame conditions is almost the same except for the fact that a loop frame condition may mention method parameters or local variables. Also, in the case for method frame condition their semantics is part of the method postcondition while the semantics of loop frame conditions can be encoded as part of the corresponding loop invariant.

```
1  public class C {
2      int a;
3
4      //@ public instance invariant a > 0 ;
5
6      //@ requires val > 0 ;
7      //@ modifies \nothing;
8      //@ ensures  true;
9      public C(int val){
10        a = val ;
11     }
12
13     //@ requires   b != 0 || b ==0;
14     //@ {|
15     //@ requires b != 0;
16     //@ modifies a;
17     //@ ensures  a == \old(a) / b;
18     //@ exsures (Exception) false
19     //@|}
20     //@ also
21     //@ {|
22     //@ requires b == 0;
23     //@ modifies \nothing;
24     //@ ensures false;
25     //@ exsures (ArithmeticException) a == \old(a);
26     //@ |}
27     public void divide(int b) {
28         a = a / b;
29     }
30 }
```

Figure 2.5: Desugared specification of the example from Fig. 2.4

In the following, let us look what is the meaning of the loop frame condition. The semantics of a loop modifies list can be encoded as part of the invariant as a condition which states that all the locations not mentioned in the loop modifies list must have the same value at the beginning and at the end state of a loop iteration. We illustrate this by an example. If a loop is specified with an invariant `INV` and the list of modified locations contains the expression a.f:

```
1  //@ loop_modifies a.f;
2  //@ loop_invariant I;
3   while (e) {
4  ...
5  }
```

then the augmented invariant resulting from the desugaring of the modifies clause is :

```
1  //@ loop_invariant   I  &&
2  //@  \forall ref; ref !=  a ==> ref.f == ref.f' &&
3  //@  \forall g \forall ref; g != f ==>  ref.g == ref.g'
4  while (e) {
5  ...
6  }
```

The first conjunct of the new invariant is the specified invariant I. The second conjunct expresses the fact the value in the field f for every object which is different from the reference a must have

the same value in the beginning (ref.f)and in the end of an iteration (ref.f'). Note that if an expression has the same value at every beginning and end of a loop iteration then it has the same value at the beginning and end of the loop. The third conjunct expresses the fact that the value of any other field g different from f for any object ref changes its value in the beginning and end of a loop iteration. This third part makes the verification of modifies frame conditions difficult as one has to enumerate explicitly all the fields of all the classes which are reachable. We do not discuss further the desugaring of the method modifies clauses in the method's postcondition as it is the same as for loop modifies clauses.

A very similar treatment of the method and loop modifies clauses is done in Jack both on the Java source and Java bytecode verification condition generators. For method modifies, the difference from the above formula is that the third conjunct is missing as we already said it is difficult to enumerate the whole heap. For loop modifies they are not verified but only assumed for the same reason of keeping the number of verification conditions reasonable.

In [77], the authors report for a change in the formalization of the heap in the verification tool Krakatoa which allows to treat also in a similar way as Jack the modifies clauses for methods. In this article, the authors also describe how to calculate read write effects of methods which allows to construct a reasonable part of the third conjunct above.

May be here we may point out that not only the assertion of modifies clauses play a role in a verification condition but also their assumption. Assuming modifies clauses on method invocation and over the verification conditions for a loop allow to an appropriate verification condition generator to initialize properly the variables or locations that are not modified by the loop or method. Consider the following example for a method m which calculates the sum of the k natural numbers in a loop where k is a method parameter:

```
1
2  class A {
3    int f;
4    //@ requires k >= 0;
5    //@ modifies this.f;
6    //@ ensures \result == this.f *(this.f+1)/2;
7    public int m(int k) {
8      this.f = k;
9      int sum = 0;
10
11     //@ loop_modifies sum, i;
12     //@ loop_invariant sum == (i*(i+1))/2 && i<=k;
13     for (int i = 0; i < k; i++) {
14       sum = sum + i;
15     }
16     return sum;
17   }
18 }
```

Before calculating the sum the instance variable a.f is set to the value of the parameter k. Assuming then the modifies clause of the loop allows to initialize properly the value of a.f and to establish that the program is correct w.r.t. the specification. Of course, the verification condition condition generator must propagate the verification conditions up to the entry of the method body. Such a verification scheme based on a weakest precondition predicate transformers allows for writing smaller specifications. Of course, here we could have used a stronger invariant (add in the loop invariant the fact that a.f == k) but in real programs the number of variables is usually larger than in this small example and the specification burden of the programmer will be inadmissible. Actually, both verification condition generators on Java source and bytecode in Jack propagate the verification conditions related to method correctness up to the method body entry point.

2.2.6 Class specification

JML can be used to specify not only methods but also properties of a class or interface. 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). A visible state of an object is basically one of these cases:

1. at the poststate of a constructor where this object is a receiver
2. at the prestate of a finalizer of that object
3. at the prestate and poststate of every method where this object is a receiver
4. in a state when no method, constructor or finalizer of this object is in progress

For a detailed discussion on visible states, the reader may refer to the JML reference manual [67]. An invariant may be either static (i.e. talks only about static fields) or instance (talks about instance fields). The class `C` in Fig.2.4 has also an instance invariant which states that the instance variable `a` is always greater than `0`. A Class history constraints is similar to a class invariant but it does not describe one visible state but rather relates a visible state and any visible state that occurs later in the execution. It can be seen as a transitive relation between the visible states of an object.

Let us now see how instance class invariants can be expressed as method pre and postconditions (the treatment of static class invariants and history constraints has the same features thus, we do not discuss them further but the reader may refer to [67]). From the first case of visible states described above, an instance invariant must be established by the constructor of this class which means that it must be part of the constructor's postcondition. From the second case it follows that the invariant of an object must be part of the precondition of the finalizers of this object. From the third case it follows that every method (not a constructor neither a finalizer) which has as a receiver an object must have as part of its pre and postcondition the object's invariant. The fourth case of a visible state implies that every method which does not have as a receiver the object must preserve this object's invariant. This actually means that every constructor must preserve (has as pre and postcondition) the invariant of all objects except for the receiver object's invariant which must be only established in the poststate. Similarly, a finalizer of an object must preserve the invariants of all the other objects and include in its precondition the object's invariant. Also every method must preserve the invariants of the objects of all classes. This actually makes the verification of class invariants a difficult task as the invariants of all objects in the heap must be mentioned in every method's pre and postcondition (except for the particular cases of constructors and finalizers). When applications are large, the number of verification conditions may be practically unmanageable and thus the verification process may be very hard. Another problem is that the verification is non modular, i.e. extending an application with a new class requires to verify the whole application once again in order to establish that the application does not violate the invariant of the objects of the newly added class. That is why verification tools may make some compromises. In JACK, for instance, constructors in a class `C` preserve the invariants of all objects of class `C` and establish the invariant of the receiver object of the constructor. Non constructor and non finalizer methods declared in class `C` preserve the invariant of all objects of class `C`.

A recent approach which relies on alias control type systems [38, 85] facilitates the verification of invariants and makes it modular. This technique relies on a hierarchy relation of "owner and owned" objects in the heap where only owners of an object may modify its contents. The alias control is guaranteed through ownership type systems which check that only an owner of a reference can modify its contents. Such technique can be used also for the sound treatment of layer object structures [83].

2.3 Program verification using program logic

Now that we have defined the source language and its specification language, let us see what are the possible approaches for verifying that a program respects its specification.

A first possibility is to formalize the semantics of programs and then to perform the verification directly over this formalization. For instance, this was done in a first version of the Loop tool [60] where a denotational semantics for programs was formalized in the theorem prover PVS. However, such a verification needs a lot of user interaction which makes difficult that this approach scale up.

Another approach then consists in using an axiomatic semantics of the programming language for the verification process. Axiomatic program semantics has been first introduced by C.A.R. Hoare [53] and is usually known under the name of Hoare logic. A later version of the Loop tool [60] and the interactive Java verification system Jive [79] are examples for Java program verification tools which use a Hoare style reasoning. Hoare logic consists in a system of inference rules defined inductively over the statement structure. The inference rules manipulate Hoare logic triples. A Hoare triple consists of a predicate Pre, a statement S and a predicate $Post$, such that if Pre holds in the initial state of S, then $Post$ holds in the final state of S. For denoting such a Hoare triple we use the notation $\{Pre\}S\{Post\}$. Correctness of program S w.r.t. a precondition Pre and postcondition $Post$ is then established if a derivation tree can be built by applying the inference rule of the Hoare logic over the triple $\{Pre\}S\{Post\}$ and the leaves of the tree are either axioms or valid formulas. Validity of formulas in the leaves of a Hoare logic tree is established by a decision procedure or a theorem prover. The inference rules in a Hoare logic are such that their premise states what should hold for the substatements of a statement S in order that the specification for S holds. Because Hoare logic is syntax directed, reasoning about programs using a Hoare logic is easier than working directly on the semantics of the programming language. However, Hoare style verification needs still a considerable amount of user interaction. For instance, we may consider the Hoare logic rule for the compositional statement given below:

$$\frac{\{Pre\}S_1\{Post'\} \quad \{Post'\}S_2\{Post\}}{\{Pre\}S_1; S_2\{Post\}}$$

The rule states that if there is a predicate $Post'$ such that the triples $\{Pre\}S_1\{Post'\}$ and $\{Post'\}S_2\{Post\}$ hold then the triple $\{Pre\}S_1; S_2\{Post\}$ holds. Although the inference rule tells us what must be proved for the substatements for establishing the correctness of $\{Pre\}S_1; S_2\{Post\}$, it does not tell which is the predicate $Post'$. A user interaction is needed to determine $Post'$. This situation occurs often also with the other rules in the logic. Actually, even for small programs, a lot of human interaction is necessary when using Hoare style reasoning.

Another possibility is to apply a more calculation approach. It consists in an algorithm wp which from a given postcondition $Post$ of statement S calculates the weakest predicate $wp(S, Post)$ that must hold in the prestate of S. Then, for establishing that S respects a precondition Pre and the postcondition $Post$ it is sufficient to show that Pre implies $wp(S, Post)$, i.e. that the verification condition $Pre \Rightarrow wp(S, Post)$ holds. The calculation of the weakest predicate can also be a difficult task because the weakest precondition for a while statement of $wp(\texttt{while}(c)S\}, Post) = X$ is the weakest solution of the following recurrence:

$$\begin{array}{l}(\neg c \wedge X) \Rightarrow Post \wedge \\ X \Rightarrow c \Rightarrow wp(S, X)\end{array}$$

Actually, the predicate X is the loop invariant but it is a fact there is no automatic procedure for finding an invariant for any loop. This means that the weakest precondition function can neither be completely automated. However, there do exist techniques for an approximative calculation of invariants: induction -iteration [103] based on iteration of the weakest precondition function over the loop body, predicate abstraction methods [45] using abstract interpretation or dynamic invariant detection (used in the Daikon tool [40]). For instance, the extended static checker for Java ESC/java uses the Daikon tool for inferring loop invariants [87]. Unfortunately, all of these approaches have shortcomings either because the calculation is very intensive and may loop forever (in the first case), because of an under-approximation (as in the second case) or not always returning a result (as in the third case). We can actually "help" the wp by supplying the loop invariants for the loop statements. In that case, the rule of wp in the case for loop would be :

$$\begin{array}{l}wp(\texttt{while}(c)[\texttt{INV}]\ S\}, Post) = \\ \texttt{INV} \wedge \\ (\texttt{INV} \wedge \neg c) \Rightarrow Post \wedge \\ (\texttt{INV} \wedge c) \Rightarrow wp(S, \texttt{INV})\end{array}$$

which makes the *wp* automated. In this case, although we do not calculate the weakest precondition but a weak precondition it is actually sufficient for our purposes. Note that in the following, we will continue to call the predicate transformer which uses annotations a weakest precondition predicate transformer function.

Note that there also exists a strongest postcondition predicate transformer function which works in a forward direction (contrary to the weakest precondition). However, the strongest postcondition predicate transformer did not gain popularity because of the way it treats assignments. Particularly, the strongest postcondition calculus quantifies existentially over the old value of the assigned variable while in a weakest predicate transformer assignments are treated with substitutions. For more detailed information on strongest postcondition calculus the reader may refer to [39]. Using weakest predicate transformers in program verification thus, avoids to deal with Hoare logic inference rules. This means that the user interacts (if needed) only in proving the verification conditions. This approach underlines the extended static checker for ESC/java [72], the Jack tool [28] and Krakatoa [76]. These tools scale up and has been used in verification of real case studies [57, 16, 62].

Another aspect of program verification using logical methods is also whether the verification should be performed directly over the original program or an intermediate language should be used. Let us see what are intermediate languages and how they can be used in the verification scheme. For instance, the simple guarded command language can be used as an intermediate language in the verification procedure. The guarded command language supports only few program constructs but which are sufficiently expressive to encode a rich programming language. If a guarded command language is used in the verification procedure this would mean a stage where bytecode programs are transformed in guarded command language programs. Using guarded command language as an intermediate representation of programs is useful because the verification condition generator can interface and can be extended to interface easily several programming languages. Moreover, if a change in the logic must be made, the change will impact only the constructs of the intermediate language which in the case of a guarded command language are few. Such an intermediate representation is used in the extended static checker ESC/java ([72]) and the Java verification system Krakatoa [76] which uses the input language of the verification tool Why[3]. However, we consider that an intermediate language opens a gap between the verified code and the source code as we verify a program in the intermediate language and not the original source program. Establishing the correctness of a verification systems which uses an intermediate language may be not trivial as the stages of the transformation of the original source program to its encoding in the intermediate language must be also proved correct. In particular, the verification ESC/java tool which uses an intermediate language does not have a proof of correctness [44].

A point that we would like also to discuss here is the use of loop frame conditions. For instance, as we saw from the examples in JML and our source language, we accompany loops not only with loop invariants but with loop frame conditions, i.e. list of locations that may be modified by a loop. Although this complicates slightly the verification condition generator, the loop frame condition is useful as it can work on weaker specification than a verification condition generator which does not support it. This in particular, implies that the burden of writing specification becomes definitely lighter, as the person which specifies an application must identify not *the strongest loop invariant* but a *loop invariant*. Of course, he should also identify the locations which are modified by the loop (locations that may have different values at the end and at the beginning of a loop iteration). However, modified locations in a loop is easier than identifying the variables not modified in the loop which must be done if we decide not use loop frame conditions.

To conclude, we consider that using predicate transformers for the generation of verification conditions makes the verification condition generator automatic. Moreover, performing the verification directly on the program code is more reliable as it guarantees that the proved program corresponds to the program that we want to prove. Last, we opt for specification which includes frame conditions as this leverages the specification burden.

[3] http://why.lri.fr/

2.4 Weakest precondition predicate transformer for Java like source language

We proceed here with the presentation of the weakest predicate transformer function. We would like first to remark that our predicate transformer deals both with normal and exceptional termination of programs. Among the first works on exceptional termination in weakest precondition predicate transformers is the work of R.Leino [71]. We adopt here a similar approach as the above cited work. Thus, the predicate transformer takes as arguments a statement S, a normal postcondition nPost^{src} and an exceptional postcondition function $\mathsf{excPost}^{src}$.

The intended meaning of the predicate WP that should be calculated by a weakest predicate transformer function against S, nPost^{src} and $\mathsf{excPost}^{src}$. The predicate WP must be such that if it holds in the pre state of S and if S terminates normally then nPost^{src} holds in the poststate and if S terminates on exception Exc then $\mathsf{excPost}^{src}(\mathrm{Exc})$ holds. The function $\mathsf{excPost}^{src}$ returns the predicate $\mathsf{excPost}^{src}(\mathtt{Exc})$ that must hold in a particular program point if at this point an exception of type `Exc` is thrown.

In the following, we will use function updates for the exceptional postcondition of the form $\mathsf{excPost}^{src}(\oplus \mathrm{Exc}' \rightarrow P)$ over $\mathsf{excPost}^{src}$ which are defined in the usual way:

$$\mathsf{excPost}^{src}(\oplus \mathrm{Exc}' \rightarrow P)(\mathrm{Exc}) = \begin{cases} P & if\ \mathrm{Exc} <: \mathrm{Exc}' \\ \mathsf{excPost}^{src}(\mathrm{Exc}) & else \end{cases}$$

Note that allowing for exceptional termination in the programming language, makes the definition of the *wp* more complicated than standard definitions of the weakest precondition predicate. Usually, those assume that programs terminate normally. Consider, for instance the standard rule of the weakest precondition predicate transformer over the conditional statement:

$$wp((E^{cond})\mathtt{then}\{S_1\}\mathtt{else}\ \{S_2\}, \mathsf{nPost}^{src}) =$$
$$E^{cond} \Rightarrow wp(S_1, \mathsf{nPost}^{src}) \wedge \neg E^{cond} \Rightarrow wp(S_2, \mathsf{nPost}^{src})$$

This rule is correct as far as the evaluation of the expression E^{cond} evaluates normally. But in our settings, this is not always the case. For instance, if E^{cond} is the relational expression $\mathtt{a.f} > 0$ and `a` evaluates to **null** the above rule does not capture the semantics of the exceptional termination. As we shall see in the following, the definition presented here will allow us to manage in appropriate way exceptional termination[4]. In particular, we extend the *wp* function also over expressions which makes the predicate calculus sensible to the exceptional and normal termination of the expression evaluation. A similar weakest precondition predicate transformer underlines the verification condition of the Jack verification framework.

In the following, we shall limit ourselves to a simpler specification than described in the previous Section 2.2. Particularly, we shall assume that methods are provided only with one specification case for the sake of clarity. Extending the definitions here to multiple specification cases should not present a major difficulty.

We shall also assume here that frame conditions are correct, i.e. we shall not desugar them into the appropriate specification (postconditions in the case of method frame conditions and loop invariants in the case of loop frame conditions) as described in subsection 2.2.5.

In the following, we give a brief discussion on the substitution over expressions and afterwords, we concentrate on the definition of a weakest precondition predicate transformer function over the source expressions and statements.

2.4.1 Substitution

Substitution is defined inductively in a standard way over the expression and formula structure. Still, we extend substitution to deal with field and array updates as follows:

[4]although here, we rule out these cases, such a definition also allows for dealing with side effects of expression evaluation like change the value of a variable or location during the evaluation as for instance is the case for Java expressions like i++

$$E[\mathtt{f}\backslash\mathtt{f}(\oplus E \rightarrow E)]$$

This substitution affects only field access expressions. For instance, the following substitution does not change the variable `a`:

$$\mathtt{a}[\mathtt{f}\backslash\mathtt{f}(\oplus E \rightarrow E)] = \mathtt{a}$$

Field substitution affects only field objects as in the following example:

$$\mathtt{b.f}[\mathtt{f}\backslash\mathtt{f}(\oplus a \rightarrow 3)] = \mathtt{b.f}(\oplus a \rightarrow 3)$$

Here, the substitution affects the field expression by making an update over the field `f`: $\mathtt{b.f}(\oplus a \rightarrow 3)$. The semantics of the resulting expression is that if $\mathtt{b} = \mathtt{a}$ then the whole expression $\mathtt{b.f}(\oplus a \rightarrow 3)$ simplifies to 3 otherwise simplifies to `b.f`.

Formally, we define substitution over field objects as follows:

$$E.\mathtt{f}_1[\mathtt{f}_2\backslash\mathtt{f}_2[\oplus E_1 \longrightarrow E_2]] =$$

$$\begin{cases} E.\mathtt{f}_1 & if\ \mathtt{f}_1 \neq \mathtt{f}_2 \\ E.\mathtt{f}_2[\oplus E_1 \longrightarrow E_2] & else \end{cases}$$

$$E.\mathtt{f}_1(\oplus E_1 \rightarrow E_2)[\mathtt{f}_2\backslash\mathtt{f}_2(\oplus E_3 \rightarrow E_4)] =$$

$$\begin{cases} \mathtt{f}_1(\oplus E_1[\mathtt{f}_2\backslash\mathtt{f}_2(\oplus E_3 \rightarrow E_4)] \rightarrow E_2[\mathtt{f}_2\backslash\mathtt{f}_2(\oplus E_3 \rightarrow E_4)]) & if\ \mathtt{f}_1 \neq \mathtt{f}_2 \\ \mathtt{f}_1\ \begin{array}{l}[\oplus E_1[\mathtt{f}_2\backslash\mathtt{f}_2(\oplus E_3 \rightarrow E_4)] \longrightarrow E_2[\mathtt{f}_2\backslash\mathtt{f}_2(\oplus E_3 \rightarrow E_4)]] \\ [\oplus E_3 \longrightarrow E_4]\end{array} & else \end{cases}$$

2.4.2 Weakest precondition predicate transformer for expressions

In the following, we give a definition of a weakest precondition predicate transformer function for source expressions.

For calculating the wp^{src} predicate of an expression E declared in method `m`, the function wp^{src} takes as arguments E, a postcondition nPost^{src}, an exceptional postcondition function $\mathsf{excPost}^{src}$ and returns the formula $wp^{src}(E, \psi, \mathsf{excPost}^{src}, \mathtt{m})_v$ which is the wp precondition of expression E if its evaluation is stored in the special logical variable v.

In Fig. 2.6 we can see the wp^{src} rules for most of the expressions of the source language (except for method invocation and instance creation). As we may notice, for some of the expressions the definition of the weakest precondition function is the identity function as their evaluation always terminates normally. For instance, the rule for constant expressions does not change the state and thus, if a predicate nPost^{src} holds after its execution this means that it held in the prestate of the expression. However, this is not the case for expressions that might throw an exception. The predicate returned by the weakest precondition predicate transformer for expressions which may throw an exception will basically accumulate the hypothesis under which the evaluation of the expression terminates normally and the conditions under which it terminates exceptionally. For instance, the rule for a cast expression (`Class`) E shows that the evaluation of the expression does not change the program state in case the E is of subtype of of class `Class`. In case the latter is not true the rule reflects the case that the a `CastExc` exception is thrown. Note that in the exceptional case, we specify that the thrown exception object is any fresh exception object w.r.t. the heap in the previous step via the predicate **instances** whose semantics is that an object belongs to the heap in the previous state. Similarly, the rule for the field access expression takes into account the two possible outcomes of its evaluation. If the evaluation v of the expression E is different from **null** then the evaluation terminates normally, otherwise the exceptional postcondition for `NullExc` must hold.

In Fig.2.7, we give the rule of instance creation. Recall that the semantics of this construct is that a new instance of the corresponding class `Class` is created and the class constructor which

$$wp^{src}(const, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v = \mathsf{nPost}^{src}[v\backslash const]$$
$$const \in \{IntConst, \mathbf{null}, \mathbf{this}, Var\}$$

$$wp^{src}(E_1\ op\ E_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v =$$
$$wp^{src}(E_1, wp^{src}(E_2, \begin{array}{l} v_2 \neq \mathbf{null} \Rightarrow \mathsf{nPost}^{src}[v\backslash v_1 op v_2] \\ \wedge \\ v_2 = \mathbf{null} \Rightarrow \left(\begin{array}{c} \forall \mathbf{bv}, (\neg \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null}) \Rightarrow \\ \mathsf{excPost}^{src}(\mathtt{ArithExc})[\mathbf{EXC}\backslash\mathbf{bv}] \end{array} \right) \end{array}, \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$
$$where\ op \in \{\mathrm{div}, \mathrm{rem}\}$$

$$wp^{src}(E.\mathtt{f}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v =$$
$$wp^{src}(E, v_1 \neq \mathbf{null} \Rightarrow \mathsf{nPost}^{src}[v\backslash v_1.\mathtt{f}] \wedge v_1 = \mathbf{null} \Rightarrow \left(\begin{array}{c} \forall \mathbf{bv}, (\neg \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null}) \Rightarrow \\ \mathsf{excPost}^{src}(\mathtt{NullExc})[\mathbf{EXC}\backslash\mathbf{bv}] \end{array} \right), \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$

$$wp^{src}((\mathtt{Class})\ E, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v =$$
$$wp^{src}(E, \mathbf{typeof}(v_1) <: \mathtt{Class} \vee v_1 = \mathbf{null} \Rightarrow \mathsf{nPost}^{src}[v\backslash v_1] \wedge$$
$$\neg\ \mathbf{typeof}(v_1) <: \mathtt{Class} \Rightarrow \forall \mathbf{bv}, (\neg \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null}) \Rightarrow \mathsf{excPost}^{src}(\ \mathtt{CastExc})[\mathbf{EXC}\backslash\mathbf{bv}],$$
$$\mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$

$$wp^{src}(E_1\ cond\ E_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v =$$
$$wp^{src}(E_1, wp^{src}(E_2, \mathsf{nPost}^{src}[v\backslash v_1 cond v_2], \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$

Figure 2.6: WP FOR SOURCE EXPRESSIONS

has the same name initializes the new reference. Class constructors in Java are also methods and thus, those are supplied with precondition $\mathtt{Class}.\mathsf{Pre}^{src}$ and a postcondition $\mathtt{Class}.\mathsf{nPost}^{src}$. Thus, the rule for instance creation states that in the state where the constructor of `Class` is invoked its precondition $\mathtt{Class}.\mathsf{Pre}^{src}$ holds and in the state in which `Class` terminates execution its postcondition $\mathtt{Class}.\mathsf{nPost}^{src}$ implies the postcondition nPost^{src} of the instance creation expression. Note that the implication is quantified over the newly created instance which did not exist in the prestate of the instance creation expression and which is different from **null**. The fact that the instance is new for the heap w.r.t. the previous execution states is expressed via the predicate ($\neg$ **instances**(**bv**)). Moreover, the fields of the new instance, i.e. fields that are declared in any superclass of `Class` or in `Class`($\mathsf{subtype}(\mathtt{f}.\mathsf{declaredIn}, \mathtt{Class})$) are set to their default values in the precondition of the constructor `Class`. This correspond to the Java semantics of creating new instances in the heap, i.e. first the reference is created and its fields are set to their default values and then, the constructor is invoked. The rule for method invocation is given in Fig.2.8 and is similar to the instance creation expression rule. In the case for method invocation, we may remark that a quantification is done over the result of the method invocation.

$$wp^{src}(\mathbf{new}\ \mathtt{Class}(), \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v =$$
$$\forall \mathbf{bv},$$
$$\neg\ \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null} \wedge \mathbf{typeof(bv)} = \mathtt{Class} \Rightarrow$$
$$\left(\begin{array}{l} \mathtt{Class}.\mathsf{Pre}^{src}[\mathbf{this}\backslash\mathbf{bv}][\mathtt{f}\backslash\mathtt{f}(\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(\mathtt{f}.\mathsf{Type}))]_{\mathsf{subtype}(\mathtt{f}.\mathsf{declaredIn}, \mathtt{Class})} \wedge \\ \left(\begin{array}{l} \forall\ x \in \mathtt{Class}.\quad \mathrm{mod}\ , \\ (\mathtt{Class}.\mathsf{nPost}^{src} \Rightarrow \mathsf{nPost}^{src})[v\backslash\mathbf{bv}] \end{array} \right) \\ \wedge \\ \left(\begin{array}{l} \forall \mathsf{Exc} \in \mathtt{Class}.\mathsf{exceptions}^{src}, \forall\ x \in \mathtt{Class}.\quad \mathrm{mod}\ , \forall \mathbf{bv}_{exc}, \\ (\mathbf{bv} \neq \mathbf{null} \wedge \mathbf{typeof}(\mathbf{bv}_{exc}) <: \mathsf{Exc} \wedge \mathtt{Class}.\mathsf{excPostSpec}^{src}(\mathsf{Exc})) \Rightarrow \mathsf{excPost}^{src}(\mathsf{Exc}) \end{array} \right) \quad [\mathbf{EXC}\backslash\mathbf{bv}_{exc}][v\backslash\mathbf{bv}] \end{array} \right)$$

Figure 2.7: WEAKEST PRECONDITION FOR INSTANCE CREATION

2.4.3 Weakest precondition predicate transformer for statements

In the following, we discuss the weakest precondition predicate transformer for control statements. We will not give an exhaustive overview of the *wp* rules for statements but we will rather concentrate

$$
\begin{array}{l}
wp^{src}(E.m(), \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v = \\
\quad m.\mathsf{Pre}^{src}[\mathbf{this}\backslash v_1] \\
\quad \wedge \\
wp^{src}(E, \left(\begin{array}{l} \forall\, x \in m.\quad \mathrm{mod}\,, \forall \mathbf{bv}, \\ m.\mathsf{nPost}^{src}[\mathbf{this}\backslash v_1][\mathbf{result}\backslash \mathbf{bv}] \Rightarrow \mathsf{nPost}^{src}[v\backslash \mathbf{bv}] \end{array}\right) \qquad\qquad , \mathsf{excPost}^{src}, \mathtt{m})_{v_1} \\
\quad \wedge \\
\quad \left(\begin{array}{l} \forall \mathtt{Exc} \in m.\mathsf{exceptions}^{src}, \forall\, x \in m.\quad \mathrm{mod}\,, \forall \mathbf{bv}_{exc}, \\ (\mathbf{typeof(bv}_{exc}\mathbf{)} <: \mathtt{Exc} \wedge \mathtt{Class.excPostSpec}^{src}(\mathtt{Exc})) \Rightarrow \mathsf{excPost}^{src}(\mathtt{Exc}) \end{array}\right) [\mathbf{EXC}\backslash \mathbf{bv}_{exc}]
\end{array}
$$

Figure 2.8: Weakest precondition for method invocation

on few which we consider illustrative.

Fig. 2.9 gives the rule for statements. The rule for field assignment shows that the statement may terminate normally or on an exception. In particular, if the dereferenced object reference E_1 does not evaluate to **null** then the postcondition nPost^{src} must hold where the value of the field `f` for the evaluation v_1 of E_1 is changed to the value v_2 of E_2. The aliasing is treated by an update of the field with the new value for the corresponding reference. If the dereferenced object reference E_1 evaluates to **null** then the postcondition $\mathsf{excPost}^{src}(\mathtt{NullExc})$ must hold.

The rule while which is slightly different from standard rules for *wp* for loops as it uses the list `modif` which stands for the locations that may be modified in a loop iteration.

Actually, if we want to verify it, we can express it as part of the loop invariant as described in subsection 2.2.5. Recall that a loop invariant is a predicate which must hold whenever the loop entry is reached. Thus, the first conjunct of the *wp* asserts that the invariant `INV` holds when the loop starts execution. The second conjunct is actually the *wp* of the loop condition. We calculate the precondition of the conditional expression upon a postcondition which expresses first that if it evaluates to true then the invariant must imply the *wp* of the loop body which terminates execution in a state where the invariant holds and second, that upon the termination of the loop, i.e. the condition evaluates to false the invariant implies the statement's postcondition normalPost. These two conditions are quantified over the elements of the `modif` list. This quantification allows to initialize correctly the variables that keep their values unchanged at the loop borders.

The control statements related to the exception handling and throwing as well as the finally statements have a more particular definition. Let us look at the rule for try catch statements. The weakest predicate of a try catch statement `try` $\{S_1\}$ `catch`(`Exc` c) $\{S_2\}$ w.r.t. a normal postcondition nPost^{src} and exceptional postcondition function $\mathsf{excPost}^{src}$ is the weakest predicate of the try statement S_1 w.r.t. the same normal postcondition nPost^{src} and the updated exceptional function $\mathsf{excPost}^{src}(\oplus \mathtt{Exc} \rightarrow wp^{src}(S_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}))$. We can see the rule for the try catch statement as dual to the rule of the compositional statement where in the latter we rather change the normal postcondition.

The try finally statement calculates the precondition of the try statement S_1 where it takes as normal postcondition the precondition of the finally statement S_2 calculated upon the initial normal postcondition nPost^{src} and exceptional postcondition function $\mathsf{excPost}^{src}$. This expresses the fact that after the try statement terminates normally the finally statement must be executed and the whole statement will terminate as the finally statement. On the other hand, the exceptional postcondition function passed to the try statement S_1 is basically the initial exceptional postcondition function $\mathsf{excPost}^{src}$ but updated for the exception type `Exception`. It is updated with the precondition of the finally statement S_2 calculated of the wp^{src} which takes as normal postcondition a predicate which we explain in the following. The postcondition which must hold in the poststate of S_2 states that $\mathsf{excPost}^{src}(\mathtt{Exception})$ holds in the normal poststate of S_2 if the variable **EXC** which stands for the thrown exception object is not **null**. It also states that the if **EXC** is not **null** then the exceptional postcondition $\mathsf{excPost}^{src}(\mathtt{NullExc})$ in case of a thrown `NullExc` is thrown.

The exception type `Exception` is the super class of all exception types and thus, an update of the exceptional postcondition function means that for any exception thrown by S_1 the exceptional postcondition is actually the precondition of the finally statement S_2. This also corresponds to the

$$wp^{src}(S_1; S_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(S_1, wp^{src}(S_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}), \mathsf{excPost}^{src}, \mathtt{m})$$

$$wp^{src}(Var = E_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(E_2, \mathsf{nPost}^{src}[Var \backslash v], \mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(E_1.\mathtt{f} = E_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(E_1, wp^{src}(E_2, \begin{array}{l} v_1 \neq \mathbf{null} \Rightarrow \mathsf{nPost}^{src}[\mathtt{f} \backslash \mathtt{f}(\oplus v_1 \rightarrow v_2)] \\ \wedge \\ v_1 = \mathbf{null} \Rightarrow (\forall \mathbf{bv}, (\neg \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null}) \Rightarrow \\ \quad \mathsf{excPost}^{src}(\mathtt{NullExc})[\mathbf{EXC} \backslash \mathbf{bv}]) \end{array}, \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$

$$wp^{src}(\ \mathtt{if}\ (E^{cond})\mathtt{then}\{S_1\}\mathtt{else}\ \{S_2\}\ , \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{if}}$$
$$wp^{src}(E^{cond}, v \Rightarrow wp^{src}(S_1, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) \wedge \neg v \Rightarrow wp^{src}(S_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}), \mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathtt{while}\ (E^{cond})\ [\mathtt{INV}, \mathtt{modif}]\ \ \{S\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$\mathtt{INV}$$
$$\wedge$$
$$\forall\ mod, mod \in \mathtt{modif},$$
$$\mathtt{INV} \Rightarrow wp^{src}(E^{cond}, v \Rightarrow\ wp^{src}(S, \mathtt{INV}, \mathsf{excPost}^{src}, \mathtt{m}) \wedge \neg v \Rightarrow \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathtt{return}\ E, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{return}}$$
$$wp^{src}(E, \mathsf{nPost}^{src}[\mathbf{result} \backslash v], \mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathtt{throw}\ E, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(E, \begin{array}{l} v = \mathbf{null} \Rightarrow (\forall \mathbf{bv}, (\neg \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null}) \Rightarrow \mathsf{excPost}^{src}(\mathtt{NullExc})[\mathbf{EXC} \backslash \mathbf{bv}]) \\ \wedge \\ v \neq \mathbf{null} \Rightarrow \left(\begin{array}{l} \forall \mathtt{Exc}\ , \\ \mathbf{typeof(}v\mathbf{)} <: \mathtt{Exc}\ \ \Rightarrow \end{array} \quad \mathtt{m}.\mathsf{excPost}^{src}(\mathtt{Exc}\ )[\mathbf{EXC} \backslash v]\ \right) \end{array}, \mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathtt{try}\ \{S_1\}\ \mathtt{catch(Exc}\ c)\ \{S_2\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(S_1, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}(\oplus \mathtt{Exc} \rightarrow wp^{src}(S_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})[c \backslash \mathbf{EXC}]), \mathtt{m})$$

$$wp^{src}(\mathtt{try}\ \{S_1\}\ \mathtt{finally}\ \{S_2\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(S_1, wp^{src}(S_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}),$$
$$\mathsf{excPost}^{src}(\oplus \mathtt{Exception} \rightarrow wp^{src}(S_2, \left(\begin{array}{l} \mathbf{EXC} \neq \mathbf{null} \Rightarrow \mathsf{excPost}^{src}(\mathtt{Exception}) \wedge \\ \mathbf{EXC} = \mathbf{null} \Rightarrow \\ (\forall \mathbf{bv}, (\neg \mathbf{instances(bv)} \wedge \mathbf{bv} \neq \mathbf{null}) \Rightarrow \\ \quad \mathsf{excPost}^{src}(\mathtt{NullExc})[\mathbf{EXC} \backslash \mathbf{bv}]) \end{array} \right), \mathsf{excPost}^{src}, \mathtt{m})), \mathtt{m})$$

Figure 2.9: Weakest precondition for control statements

semantics of try finally described earlier. In particular, it says that if the try statement terminates on exception E the finally statement must be executed. If the finally statement terminates normally, then the whole statement terminates on exception E and if the finally statement terminates on exception E' then the whole statement terminates on exception E'.

Chapter 3

Java bytecode language and its operational semantics

In this chapter, we shall turn our attention to a bytecode language and and its operational semantics. The bytecode language will be used all along the thesis and more particularly later in Chapter 5 for the definition of the verification condition generator. We define the operational semantics of the bytecode language as a relation between the initial and final states of instruction execution. The operational semantics will be used for establishing the soundness of the verification procedure. As our verification procedure is tailored to Java bytecode the bytecode language introduced hereafter is close to the Java Virtual Machine language [75](JVM for short).

In particular, **the features supported** by our bytecode language are

arithmetic operations like multiplication, division, addition and subtraction.

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 invocation In the following, we consider only non void methods. We restrict our modeling for the sake of simplicity without losing any specific feature of the Java language.

object manipulation and creation Java supports all the facilities for object creation and manipulation. This is an important aspect of the Java language as it is completely object oriented. Thus, we support field access and update as well as object creation. Note that we also support arrays with the corresponding facilities for creation, access and update.

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. Programmatic exceptions are forced by the programmer with a special instruction in the program text.

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 hierarchy.

integer type The unique basic type which is supported is the integer type. This is not so unrealistic as the JVM treats the other integral types e.g. byte and short like the integer type. JVM actually supports only few instructions for dealing in a special way with the array of byte and short.

Our bytecode language omits some of the features of Java. Let us see which ones exactly and why.

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. However, in our implementation we treat void methods.

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. Once again, static methods are supported in the implementation of the verification condition generator.

static initialization This part of the JVM is discarded as the JVM specification is ambiguous on it and thus, its formal understanding is difficult and complex. However, research exists on the subject as is the work by Leino and al. [69] which deals with static initialization and static invariants.

garbage collection and finalization Both in the formalization presented in the following as well as in the implementation of the verification condition generator, we do not support garbage collection, i.e. unused memory is never freed and assume an infinite memory. Garbage collection is the general name of techniques for freeing unused memory for later reallocation. Finalizer methods are closely related to garbage collection. A program developer may write a finalizer method in a class to specify what are the final operations to be done before an object of this class is reallocated. Finalizers are executed by the garbage collector. Reasoning and modeling a garbage collected heap should take into account the notion of reachability of an object in the heap which can be not trivial. In particular, most of the existing formalizations of Java discard the garbage collection. Also, standard program logic and semantics of program logic assertions do not take into account garbage collection, as for instance is the standard Hoare logic. An interesting work which proposes a new semantics of assertions in a language with garbage collection is [30]. Although the treatment of garbage collection is omitted here, it is an important part of the JVM and future extensions of the formalization here should take it into account.

subroutines Subroutines in Java is a piece of code inside a method which must be executed after another, no matter how the first terminate execution. They correspond to the construction `try{ } finally { }` in the Java language. Subroutines are a controversy point in the JVM because on one hand they complicate a lot the bytecode verification algorithm in Java and second, slow the JVM execution because of the way they are implemented. While the standard compilation of `try{ } finally { }` is with special instructions, the most recent Java compilers inline the subroutine code which results more efficient for the bytecode verifier as well as for the code execution. In the implementation of our verification calculus we also inline subroutines and in this way we omit the special bytecode constructs for subroutines in our bytecode language.

errors The exception mechanism in Java is provided with two kind of exceptions: exceptions from which a reasonable application may recover. i.e. handle the exception and exceptions which cannot be recovered. This last group of exceptions in Java are called JVM errors. JVM errors are thrown typically when the proper functioning of the JVM cannot continue. The cause might be an error on loading, linking, initialization time of a class or on execution time because of deficiency in the JVM resource, e.g. stack, memory overflow. For instance, during the resolution of a field name may terminate on `NoSuchFieldError` if such a field is not found. Another example is the `MemoryOverflowError` thrown when no more memory is available. Note that such errors is not always clear how to express in our program logic presented later in the thesis. This is because, the logic is more related with the functional properties of the program while the JVM errors are related more to the physical state of the JVM. Thus, we omit here this aspect of the JVM.

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 behavior. 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.

arithmetic overflow The arithmetic in Java is bounded. This means that if the result of an arithmetic operation exceeds (is below) the largest integer (the smallest) the operation will result in overflow. The arithmetic proposed here is infinite. Note that depending on what are the objectives of a formalization of a programming language semantics, one might support or not bounded arithmetic. For instance, ESC/java does not support bounded arithmetic for the sake of efficiency [44]. However, if the application domain of a verification tool targets programs which are sensitive to arithmetic errors it is certainly of interest to support arithmetic overflow. We could have designed the arithmetic operations such that they take into account the arithmetic overflow as we consider that for the purposes of the present thesis this is not necessary and will complicate the presentation without bringing any particular feature of the bytecode.

64 bit arithmetic We do not consider long values as their treatment is similar to the integer arithmetic. However, it is true that the formalization and manipulation of the long type can be more complicated as long values are stored in two adjacent registers but it is feasible to extend the current formalization to deal with long values.

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 [70]. Although larger and more complicated than integral data, formalization of floating point arithmetic is possible. For example, the specification of IEEE [36] 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 the current document.

Now that we have seen the general outlines of our language, in the rest of this chapter we shall proceed with a more detailed description.

The rest of this chapter is organized as follows: subsection 3.1 gives some particular notations that will be used from now on along the thesis, subsection 3.2 introduces the structures classes, fields and methods used in the virtual machine, subsection 3.3 gives the type system which is supported by the bytecode language, subsection 3.4 introduces the notion of state configuration, subsection 3.4.1 gives the modeling of the memory heap, subsection 3.7 is a discussion about our choice for operational semantics, subsection 3.8 gives the operational semantics of our language and finally we conclude with subsection 3.10 which is an overview of existing formalizations of the JVM semantics. .

3.1 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 $f : A_1 * ... * A_n \to B$. 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 :

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

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 $l.length - 1$.

3.2 Program, classes, fields and methods

We turn our attention to modeling the structure of a Java class. In the following, classes are encoded with the data structure **Class**, fields with **Field** and methods are encoded with **Method** data structure. In the following, we use the domain **ClassName** for class names, **FieldName** for field names and **MethodName** for method names 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 $\bot$. 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:

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

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 encodes in this way array length 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 types(argsType), 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. Finally, the structure also contains information about the class where the method is declared(declaredIn).

$$\textbf{Method} = \left\{ \begin{array}{ll} \mathsf{Name} & : \textbf{MethodName} \\ \mathsf{retType} & : \mathit{JType} \\ \mathsf{argsType} & : \mathit{list}\ \mathit{JType} \\ \mathsf{nArgs} & : \mathit{nat} \\ \mathsf{body} & : \mathit{list}\ \mathrm{I} \\ \mathsf{excHndlS} & : \mathit{list}\ \textbf{ExcHandler} \\ \mathsf{exceptions} & : \mathit{list}\ \textbf{Class}_{exc} \\ \mathsf{declaredIn} & : \textbf{Class} \end{array} \right\}$$

We assume that for every method m the entry point 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.

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

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:

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

3.3 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.

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

Our language supports two kinds of values : values of the basic type **int** and reference values *RefVal*. *RefVal* may be references to class objects, references to array objects or the special null value which denotes the reference pointing nowhere. The set of references of class objects is denoted with *RefValCl*, the set of references to array objects is represented with *RefValArr* and the null reference value is denoted with **null**. The following definition gives the formal grammar for values:

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

Every reference has an associated type which is determined by the function TypeOf which maps references to their dynamic type:

$$\mathsf{TypeOf}\ : \mathit{RefVal} \rightarrow \mathit{RefType}$$

Every type has an associated default value which can be accessed via the function defVal. Particularly, for reference types (*RefType*) the default value is **null** and the default value of **int** type is 0. Thus, the definition of the function defVal is as follows:

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

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

We define also a subtyping relation as follows:

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

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

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

Note that the subtyping relation that we use here is a subset of the Java subtyping relation. However, differently from the system of program types in Java, here we do not consider interface and abstract class types.

3.4 State configuration

State configurations model 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.
- St is 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

We will denote with $< \mathrm{H}, Final >^{final}$ 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 modeling and the operand stack.

3.4.1 Modeling the object heap

An important issue for the modeling 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 modeling, 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 3.2) into partial functions from references (*RefType*) into values (*Values*).
- a component Arr which is a partial function from the components of arrays into their values
- a component Loc which stands for the list of references allocated in the heap

Formally, the data type HeapType has the following structure:

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

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 for which the components H.Fld and H.Arr are functions defined only for references from the proper type, i.e. well-typed and which are in the list of references of the heap H.Loc and are not **null**, i.e. well-defined. We give the following formal definition.

Definition 3.4.1. *We say that the heap* H *is well-formed if the following holds:*

$$\begin{array}{l} \forall \mathtt{f} : \mathbf{Field}, \\ \quad \mathsf{Dom}(\mathrm{H}.\mathsf{Fld}(\mathtt{f})) = \{\mathtt{ref} \mid \begin{array}{l} \mathtt{ref} \in \mathrm{H}.\mathsf{Loc} \wedge \\ \mathtt{ref} \neq \mathbf{null} \\ \mathsf{subtype}(\mathsf{TypeOf}\ (\mathtt{ref}), \mathtt{f}.\mathsf{declaredIn})\} \end{array} \\ \textit{and} \\ \quad \mathsf{Dom}(\mathrm{H}.\mathsf{Arr}) = \{\mathtt{ref} \mid \begin{array}{l} \mathtt{ref} \in \mathrm{H}.\mathsf{Loc} \wedge \\ \mathtt{ref} \neq \mathbf{null} \\ 0 \leq i < \mathrm{H}.\mathsf{Fld}(\mathrm{arrLength})(\mathtt{ref})\} \end{array} \end{array}$$

If a new object of class C is created in the memory, a fresh reference value `ref` different from **null** 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:

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

The formalization of the resulting heap and the new reference is given by the following definition.

Definition 3.4.2. *Operator for instance allocation*

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

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:

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

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

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

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

Definition 3.4.3. *Operator for array allocation*

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

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 : $\mathrm{H.Fld}(\mathtt{f})$ is replaced with $\mathrm{H}(\mathtt{f})$ for the sake of simplicity.

The same abbreviation 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:

$$\mathrm{H.Fld}(\oplus \mathtt{f} \rightarrow \mathrm{H.Fld}(\mathtt{f})(\oplus \mathtt{ref} \rightarrow \mathit{val}))$$

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

$$\mathrm{H}(\oplus \mathtt{f} \rightarrow \mathtt{f}(\oplus \mathtt{ref} \rightarrow \mathit{val}))$$

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 in an update of the component H.Arr:

$$\mathrm{H.Arr}(\oplus(\texttt{ref}, i) \rightarrow val)$$

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

$$\mathrm{H}(\oplus(\texttt{ref}, i) \rightarrow val)$$

3.4.2 Registers

State configurations have an array of registers which is denoted with Reg. Registers are addressed by indexing and the index of the first register is 0. Thus, Reg(0) stands for the first register in the state configuration. An integer is be considered to be an index into the register array if and only if that integer is between zero and one less than the size of the register array. Registers are used to pass parameters on method invocation. On static method invocation, any parameters are passed in consecutive registers starting from register Reg(0). The register 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). The other parameters are subsequently passed in consecutive registers starting at register at index 1.

3.4.3 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 vice versa. 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.

3.4.4 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.

3.5 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:

- `NullExc` thrown if a null pointer is dereferenced
- `NegArrSizeExc` thrown if there is an attempt to create an array with a negative size
- `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 instruction as we shall see later in the coming section.

The modeling of the exception handling mechanism involves several auxiliary functions. The function *getStateOnExcRT* deals with bytecode instructions that may throw runtime exceptions. This function applies only to instructions which may throw a `RuntimeExc` exception but which are not a method invocation neither the special instruction by which the program can throw explicitly an exception. The function returns the state configuration after the current instruction during the execution of `m` throws a runtime 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. Note also that the heap is changed as a new instance of the corresponding exceptional type is created:

$$getStateOnExcRT : S^{interm} * \texttt{ExcType} * \textbf{ExcHandler}[] \rightarrow S^{interm} \cup S^{exc}$$

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

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

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 exists such a handler the function returns the index of the instruction at which the exception handler starts, otherwise it returns $\bot$:

$$findExcHandler : \texttt{ExcType} * nat * \textbf{ExcHandler}[] \rightarrow \{nat \cup \bot\}$$

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

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

3.6 Method lookup

Java allows that a class declares a method `m` with the same signature as in one of its super types. In this case, we say that method `m` overrides the corresponding method in the super type or interface. When defining subtypes, this feature of Java allows to change the behavior of the subtype w.r.t. the superclass behavior.

This however complicates slightly the invocation mechanism. Let us see why. The problem arises because the types of variables on execution time differ from the type with which they are declared. In particular, in Java we talk about dynamic type and static type of expressions. Dynamic types can not be determined statically, they can only be determined on execution time. For instance, if we have a Java expression $a.\mathtt{m}()$ where the variable a is declared with type A and class A has a subclass B, in the general case it is not possible to determine statically if a contains a reference value of dynamic type B or dynamic type A. If a method `m` is invoked on object of dynamic type B accordingly to the semantics of Java, we intend to call the method declared in the class B if a method with such signature is declared in B.

Because this cannot be resolved statically, it is the virtual machine which on execution time determines which method to execute. In order, to simulate this behavior we define the function *lookUp*, which takes a method signature - its name, its argument types and its return value and a class `Class` and determines either the method is declared in `Class` via the function *findMethod* which searches in the list of the method of class `Class`. If this is the case, then the lookup procedure returns the method in `Class`, otherwise the procedure continues recursively to look for the method in the super type `Class.superClass` of `Class`. Finally, we give the function for the method look up:

$$lookUp(name, argTypes, retType, \mathtt{Class}) = \begin{cases} \mathtt{m} & findMethod(name, argTypes, retType, \mathtt{Class}) = \mathtt{m} \\ \\ \mathtt{m} & \begin{array}{l} findMethod(name, argTypes, retType, \mathtt{Class}) = \bot \wedge \\ lookUp(name, argTypes, retType, \mathtt{Class.superClass}) = \mathtt{m} \end{array} \end{cases}$$

3.7 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 and the array of registers of the method. 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. Thus, an operational semantics which follows closely the JVM would model the method frame stack and use a small step operational semantics.

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 program termination, i.e. the verification calculus assumes the correctness and the termination of the rest of the methods, we do not need a model for reasoning about the termination or the correctness of invoked methods. A big step operational semantics which is silent about the method frame stack provides a suitable level of abstraction.

3.8 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 invocation,

as well as exception throwing and handling. In fig. 3.1, we give the list of instructions that constitute our bytecode language[1].

$$
\begin{array}{rl}
\mathrm{I} ::= & \texttt{nop} \\
& \mid \texttt{if\_}\mathit{cond} \\
& \mid \texttt{goto} \\
& \mid \texttt{return} \\
& \mid \texttt{arith\_op} \\
& \mid \texttt{load} \\
& \mid \texttt{store} \\
& \mid \texttt{push} \\
& \mid \texttt{pop} \\
& \mid \texttt{dup} \\
& \mid \texttt{iinc} \\
& \mid \texttt{new} \\
& \mid \texttt{newarray} \\
& \mid \texttt{putfield} \\
& \mid \texttt{getfield} \\
& \mid \texttt{astore} \\
& \mid \texttt{aload} \\
& \mid \texttt{arraylength} \\
& \mid \texttt{instanceof} \\
& \mid \texttt{checkcast} \\
& \mid \texttt{athrow} \\
& \mid \texttt{invoke}
\end{array}
$$

Figure 3.1: BYTECODE LANGUAGE

We define the operational semantics of a single Java instruction as a relation between its initial and final state configurations as follows.

Definition 3.8.1 (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*

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

We also define the transitive closure of the single execution step with the following definition.

Definition 3.8.2 (Transitive closure of a method state transition relation). *If in the execution of the method* m *in state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *executes an instruction* I *and there exists a transitive state transition to the state* $< \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$ *we denote this with:*

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

The following definition characterizes the executions that terminate.

Definition 3.8.3 (Termination of method execution). *If the method* m *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *at the entry point instruction* $\texttt{m}.\mathsf{body}[0]$ *and there is a transitive state transition to the final state* $< \mathrm{H}', \mathit{Final} >^{final}$ *then we denote this with:*

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

[1]The instruction `arith_op` stands for any arithmetic instruction in the list `add`, `sub`, `mult`, `and`, `or`, `xor` , `ishr`, `ishl`, `div`, `rem`

We now give the operational semantics of a terminating method execution. A terminating execution of method `m` is the execution of its body up to reaching a final state configuration:

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

We now turn to the semantics of the individual instructions in our language. Fig. 3.2 gives the rules for control transfer instructions. The first rule refers to the instruction `if_`*cond*. 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 evaluates to 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. The rule for `goto` shows that the instruction transfers the control to the instruction at position n. The instruction `return` causes the normal termination of the execution of the current method `m`. The resulting poststate of the instruction is a final state where the return value is contained in the stack top element St(Cntr).

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

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

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

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

Figure 3.2: Operational semantics for the `nop` and control transfer instructions

Fig. 3.3 shows the semantics of arithmetic instructions and instructions for loading and storing on the operand stack. Arithmetic instruction pop the values which are on the stack top St(Cntr) and St(Cntr - 1) at the position below and apply the corresponding arithmetic operation 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). Note that our formalization does not take into consideration overflow of arithmetic instructions, i.e. we assume that we can manipulate any unbounded integers. We may remark that there are two rules for the arithmetic instructions `div` and `rem`. This is because these instructions may terminate on a runtime exception when the second argument is 0. Let us focus on their rules in more detail. From the rule for exceptional termination we can see that the exception handler table `m.excHndlS` of the current method will be searched for an exception handler protecting the current position Pc from `ArithExc` exceptions and depending whether such a handler was found or not the instruction execution will terminate exceptionally or not.

The instructions `load` i and `store` i load and store respectively the value of the local variable of the currently executing method, while the the instruction `iinc` increments the value of the local variable Reg(i). The instruction `push` i pushes on the stack top the integer value i. The instruction `pop` pops the stack top element. The instruction `dup` duplicates the stack top element St(Cntr).

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

$$\frac{\begin{array}{l}op = \{\mathrm{div}, \mathrm{rem}\}\\ \mathrm{St}(\mathrm{Cntr}) = 0\\ \mathit{getStateOnExcRT}(< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \texttt{ArithExc}, \mathtt{m}.\mathsf{excHndlS}) = S\end{array}}{\mathrm{m} \vdash \texttt{arith\_op}{:}<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}> \hookrightarrow S}$$

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

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

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

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

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} - 1\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \texttt{pop}{:}<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}> \hookrightarrow <\mathrm{H},\mathrm{Cntr}',\mathrm{St},\mathrm{Reg},\mathrm{Pc}'>}$$

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}(\oplus \mathrm{Cntr} + 1 \to \mathrm{St}(\mathrm{Cntr}))\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \texttt{dup}{:}<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}> \hookrightarrow <\mathrm{H},\mathrm{Cntr}',\mathrm{St}',\mathrm{Reg},\mathrm{Pc}'>}$$

Figure 3.3: Operational semantics for arithmetic and load store instructions

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

$$\frac{\mathrm{St}(\mathrm{Cntr}-1) = \textbf{null} \quad \mathit{getStateOnExcRT}(< \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc} >, \mathtt{NullExc}, \mathtt{m.excHndlS}) = S}{\mathtt{m} \vdash \mathtt{putfield\ f}{:}\langle \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\rangle \hookrightarrow S}$$

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

$$\frac{\mathrm{St}(\mathrm{Cntr}) = \textbf{null} \quad \mathit{getStateOnExcRT}(< \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc} >, \mathtt{NullExc}, \mathtt{m.excHndlS}) = S}{\mathtt{m} \vdash \mathtt{getfield\ f}{:}\langle \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\rangle \hookrightarrow S}$$

Figure 3.4: Operational semantics for field manipulation

Fig. 3.4, 3.5 and 3.6 give the semantics for instructions manipulating the program heap. Let us focus on Fig. 3.4 which shows how object fields are accessed and modified. We shall concentrate on the instruction for field update `putfield`, the instruction for `astore` and array length access `arraylength`, the others being similar. The instruction `putfield` pops the top value contained on the stack top St(Cntr) and the reference value contained in St(Cntr - 1). If the reference St(Cntr - 1) is not **null**, the function standing for the `f` is updated with the valueSt(Cntr) for the reference St(Cntr - 1) and the counter Cntr is decremented. If the reference in St(Cntr - 1) is **null** then a `NullExc` is thrown.

The instruction `astore` stores the value in St(Cntr) at index St(Cntr -1) in the array St(Cntr -2). 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 in the stack top element is stored in the component at index St(Cntr - 1) of the array in St(Cntr - 2). If the array reference St(Cntr - 2) is **null** a `NullExc` 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.

The instruction `arraylength` gets the length of an array. The stack top element is popped from the stack. It must be a reference that points to an array. If the stack top element is not **null** the length of the array arrLengthSt(Cntr) is fetched and pushed on the stack. If the stack top element is **null** then a `NullExc` is thrown. Here we can see how the array length is modeled via the special object field arrLength.

Let us now look at Fig. 3.6 to see how object creation is modeled.

For example, a new class instance is created by the instruction `new`. 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). It deserves to note that although the semantics of the instance creation described here is very close to the JVM semantics, we omit the so called VM errors, e.g. such the class from which an instance must be created is not found.

The instruction `newarray` creates 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 initialized 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.

Our language also supports instructions for checking if an object is of a given type. They are

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

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr}-2) = \textbf{null} \\ \mathit{getStateOnExcRT}(<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}>, \texttt{NullExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash\texttt{astore} :<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}>\hookrightarrow S}$$

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr}-2) \neq \textbf{null} \\ (\mathrm{St}(\mathrm{Cntr}-1) < 0 \vee \ \mathrm{St}(\mathrm{Cntr}-1) \geq \mathrm{arrLength}(\mathrm{St}(\mathrm{Cntr}-2))) \\ \mathit{getStateOnExcRT}(<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}>, \texttt{ArrIndBndExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash\texttt{astore} :<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}>\hookrightarrow S}$$

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

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr}-1) = \textbf{null} \\ \mathit{getStateOnExcRT}(<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}>, \texttt{NullExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash\texttt{aload} :<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}>\hookrightarrow S}$$

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

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

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

Figure 3.5: Operational semantics for array manipulation

given in Fig. 3.7. For instance, the instruction `instanceof` checks if the stack top element is of subtype C, then the 1 is pushed on the stack. If the object reference is **null**or not a subtype of C then 0 is pushed on the stack top. The `checkcast` instruction has a similar behavior, only that in case that the stack top element is not a subclass of C a `CastExc` is thrown.

The language presented here allows also to force exception throwing. This is done via the instruction `athrow` presented in Fig. 3.8. The stack top element must be a reference of an object of type `Throwable`. If the exception object on the stack top is not **null** then there are two possible execution of the instruction. Either there is not an exception handler that protects this bytecode

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

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

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

Figure 3.6: Operational semantics for object creation

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

$$\frac{\begin{array}{l}\neg(\mathsf{subtype}(\mathrm{H.TypeOf}\ (\mathrm{St}(\mathrm{Cntr})), C)) \vee \mathrm{St}(\mathrm{Cntr}) = \textbf{null}\\ \mathrm{St}' = \mathrm{St}(\oplus\mathrm{Cntr} \rightarrow 0)\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\texttt{m}\vdash\texttt{instanceof c}{:}\!<\!\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\!>\!\hookrightarrow\!<\!\mathrm{H},\mathrm{Cntr},\mathrm{St}',\mathrm{Reg},\mathrm{Pc}'\!>}$$

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

$$\frac{\begin{array}{l}\mathrm{St}(\mathrm{Cntr}) \in \mathrm{H.Loc}\\ \neg(\mathsf{subtype}(\mathrm{H.TypeOf}\ (\mathrm{St}(\mathrm{Cntr})), C))\\ \mathit{getStateOnExcRT}(< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \texttt{CastExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash\texttt{checkcast}\ C{:}\!<\!\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\!>\!\hookrightarrow S}$$

Figure 3.7: Operational semantics for type checking

instruction from the exception type and the current method `m` terminates exceptionally by throwing the exception object St(Cntr) or there is a handler that protects this bytecode instruction from the exception thrown and the control is transfered to the instruction at index Pc^{eH} at which the exception handler starts. If the object on the stack top is **null**, a `NullExc` is thrown and is handled as the function *getStateOnExcRT* prescribes.

Finally, in Fig. 3.9, we can see the semantics of method invocation. As the rule shows, first the proper method n to be executed will be looked for via the function *lookUp*. This is is done by taking from the data structure meth the method signature - the method name meth.Name, its argument types meth.argsType and its return type method.retType. The search starts from the dynamic type of the object reference on which the method must be called. Once the proper method method n is found it is invoked. Next, the first top n.nArgs elements in the operand stack St are

$$\frac{\mathrm{St(Cntr)} \neq \textbf{null} \quad \mathit{findExcHandler}(\mathrm{H.TypeOf\ (St(Cntr))}, \mathrm{Pc}, \mathtt{m.excHndlS}) = \bot}{\mathtt{m} \vdash \mathtt{athrow}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow \langle \mathrm{H,St(Cntr)} \rangle^{exc}}$$

$$\frac{\mathrm{St(Cntr)} \neq \textbf{null} \quad \mathit{findExcHandler}(\mathrm{H.TypeOf\ (St(Cntr))}, \mathrm{Pc}, \mathtt{m.excHndlS}) = \mathrm{Pc}^{eH} \quad \mathrm{St'} = \mathrm{St}(\oplus 0 \rightarrow \mathrm{St(Cntr)})}{\mathtt{m} \vdash \mathtt{athrow}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow \langle \mathrm{H,0,St',Reg,Pc}^{eH} \rangle}$$

$$\frac{\mathrm{St(Cntr)} = \textbf{null} \quad \mathit{getStateOnExcRT}(< \mathrm{H, Cntr, St, Reg, Pc} >, \mathtt{NullExc}, \mathtt{m.excHndlS}) = S}{\mathtt{m} \vdash \mathtt{athrow}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow S}$$

Figure 3.8: Operational semantics for programmatic exceptions

popped from the operand stack. If St(Cntr - n .nArgs) is not **null**, the method n is executed on the object St(Cntr - n.nArgs) and where the first n.nArgs+ 1 elements of the list of its of local variables is initialized with St(Cntr - n .nArgs) ... St(Cntr). In case that the execution of method n terminates normally, the return value Res of its execution is stored on the operand stack of the invoker. If the execution of of method n 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 - n.nArgs) on which the method n must be called is **null**, a `NullExc` is thrown. Note that in this last case we do not perform method lookup as it is sure that an exception can be thrown.

$$\frac{\mathit{lookUp}(\mathrm{meth.Name}, \mathrm{meth.argsType}, \mathrm{meth.retType}, \mathrm{TypeOf\ (St(Cntr - meth.nArgs))}) = \mathrm{n} \quad \mathrm{St(Cntr - n.nArgs)} \neq \textbf{null} \quad \mathrm{n} :< \mathrm{H}, 0, [\,], [\mathrm{St(Cntr - n.nArgs)}, \ldots, \mathrm{St(Cntr)}], 0 >\Downarrow< \mathrm{H'}, \mathrm{Res} >^{norm} \quad \mathrm{Cntr'} = \mathrm{Cntr} - \mathtt{m.nArgs} + 1 \quad \mathrm{St'} = \mathrm{St}(\oplus \mathrm{Cntr'} \rightarrow \mathrm{Res}) \quad \mathrm{Pc'} = \mathrm{Pc} + 1}{\mathtt{m} \vdash \mathtt{invoke}\ \mathrm{meth}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow \langle \mathrm{H',Cntr',St',Reg,Pc'} \rangle}$$

$$\frac{\mathit{lookUp}(\mathrm{meth.Name}, \mathrm{meth.argsType}, \mathrm{meth.retType}, \mathrm{TypeOf\ (St(Cntr - meth.nArgs))}) = \mathrm{n} \quad \mathrm{St(Cntr - n.nArgs)} \neq \textbf{null} \quad \mathrm{n} :< \mathrm{H}, 0, [\,], [\mathrm{St(Cntr - n.nArgs)}, \ldots, \mathrm{St(Cntr)}], 0 >\Downarrow< \mathrm{H'}, \mathrm{Exc} >^{exc} \quad \mathit{findExcHandler}(\mathrm{H'.TypeOf\ (Exc)}, \mathrm{Pc}, \mathtt{m.excHndlS}) = \bot}{\mathtt{m} \vdash \mathtt{invoke}\ \mathrm{meth}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow \langle \mathrm{H',Exc} \rangle^{exc}}$$

$$\frac{\mathit{lookUp}(\mathrm{meth.Name}, \mathrm{meth.argsType}, \mathrm{meth.retType}, \mathrm{TypeOf\ (St(Cntr - meth.nArgs))}) = \mathrm{n} \quad \mathrm{St(Cntr - meth.nArgs)} \neq \textbf{null} \quad \mathrm{n} :< \mathrm{H}, 0, [\,], [\mathrm{St(Cntr - n.nArgs)}, \ldots, \mathrm{St(Cntr)}], 0 >\Downarrow< \mathrm{H'}, \mathrm{Exc} >^{exc} \quad \mathit{findExcHandler}(\mathrm{H'.TypeOf\ (Exc)}, \mathrm{Pc}, \mathtt{m.excHndlS}) = \mathrm{Pc}^{eH} \quad \mathrm{St'} = \mathrm{St}(\oplus 0 \rightarrow \mathrm{Exc})}{\mathtt{m} \vdash \mathtt{invoke}\ \mathrm{meth}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow \langle \mathrm{H',0,St',Reg,Pc}^{eH} \rangle}$$

$$\frac{\mathrm{St(Cntr - meth.nArgs)} = \textbf{null} \quad \mathit{getStateOnExcRT}(< \mathrm{H, Cntr, St, Reg, Pc} >, \mathtt{NullExc}, \mathtt{m.excHndlS}) = S}{\mathtt{m} \vdash \mathtt{invoke}\ \mathrm{meth}{:}\langle \mathrm{H,Cntr,St,Reg,Pc} \rangle \hookrightarrow S}$$

Figure 3.9: Operational semantics for method invocation

3.9 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 3.2 that every method `m` has an array of bytecode instructions `m.body`. 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].

The array of bytecode instructions of a method `m` determine the control flow graph $G(V, \longrightarrow)$ of method `m` in which the vertexes are the instruction indexes of the method body:

$$V = \{k \mid 0 \leq k < \texttt{m.body}.length\}$$

Fig. 3.10 gives the definition of the execution relation between instructions. Note first that we rather use the infix notation $j \longrightarrow k$ instead of $(j, k) \in \longrightarrow$. Moreover, there is an edge between two vertices's j and k if they may execute immediately one after another. We say that j is a predecessor of k and that k is a successor of j. The definition states the `return` instruction does not have successors. If `m.body`[j] is the jump instruction `if_`*cond* k then its successors are the instruction at index k and the instruction at index j in `m.body`. 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 `NullExc` exception. The possible successors of the instruction `athrow` are the entry point of any exception handler in the method `m`. .

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 [9]. The next definition identifies backlogs 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 $\longrightarrow^l$ as follows:

Definition 3.9.1 (Backedge Definition). *Assume we have the method* `m` *with body* `m.body` *which determine the control flow graph* $G(V, \longrightarrow)$ *with entry point* `m.body`[0]. *In such a graph G, we say that loopEntry is a loop entry instruction and loopEnd is a loop end instruction of the same loop if the following conditions hold:*

- *for every path P in the control flow graph P from* `m.body`[0] *to loopEnd there exists a subpath subP which is a prefix of P and which terminates at loopEntry such that loopEnd does not belong to subP*
- *there is a path in the control flow graph in which loopEntry follows immediately after loopEnd (loopEnd* $\longrightarrow$ *loopEntry)*

We denote the execution relation between loopEnd and loopEntry with $loopEnd \longrightarrow^l loopEntry$ *and we say that* $\longrightarrow^l$ *is a loop backedge.*

In [9], reducibility is defined in terms of the dominator relation. Although not said explicitly, the first condition in the upper definition corresponds to the dominator relation.

We illustrate the above definition with the control flow graph of the example from Fig. 2.2.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 loop 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`). Note that the "back" in "backedge" stands for

$$\frac{\mathtt{m.body}[j]=\mathtt{nop}}{j\longrightarrow j+1}\qquad \frac{\mathtt{m.body}[j]=\mathtt{if\_}\mathit{cond}\ k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{goto}\ k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{putfield}\ \mathit{findExcHandler}(\mathtt{NullExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{putfield}\ \mathit{findExcHandler}(\mathtt{NullExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{getfield}\ \mathit{findExcHandler}(\mathtt{NullExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{astore}\ \mathit{findExcHandler}(\mathtt{NullExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{astore}\ \mathit{findExcHandler}(\mathtt{ArrIndBndExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{aload}\ \mathit{findExcHandler}(\mathtt{NullExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{aload}\ \mathit{findExcHandler}(\mathtt{ArrIndBndExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{invoke\ n}\ \mathit{findExcHandler}(\mathtt{NullExc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{invoke\ n}\quad \begin{array}{l}\forall \mathtt{Exc}, \exists s, \mathtt{n.exceptions}[s]=\mathtt{Exc}\wedge\\ \mathit{findExcHandler}(\mathtt{Exc},j,\mathtt{m.excHndlS})=k\end{array}}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]=\mathtt{athrow}\ \forall \mathtt{Exc},\mathit{findExcHandler}(\mathtt{Exc},j,\mathtt{m.excHndlS})=k}{j\longrightarrow k}$$

$$\frac{\mathtt{m.body}[j]\neq\mathtt{goto}\ \mathtt{m.body}[j]\neq\mathtt{return}\ k=j+1}{j\longrightarrow k}$$

Figure 3.10: Execution relation between bytecode instructions in a control flow graph

that the control flow goes back to an instruction through which the execution path has already passed which does not imply that the orientation of the edge is in a backwards direction in the graphical representation of the control flow graph.

3.10 Related Work

A considerable effort has been done on the formalization of the JVM semantics as a reply to the holes and ambiguities encountered in the specification of the Java bytecode verifier. Thus, most of the existing formalizations are used for reasoning over the Java bytecode well-typedness. Differently from the aforementioned formalizations, our formalization will serve us to prove the verification condition scheme introduced in Chapter 5.

We can start with the work of Stata and Abadi [102] in which they propose a semantics and typing rules for checking the correct behavior of subroutines in the presence of polymorphism. In [48] N.Freund and J.Mitchell extend the aforementioned work to a language which supports all the Java features e.g. object manipulation and instance initialization, exception handling, arrays. In [24], Boerger and all formalize the semantics of the JVM as well as a compiler from Java to Java

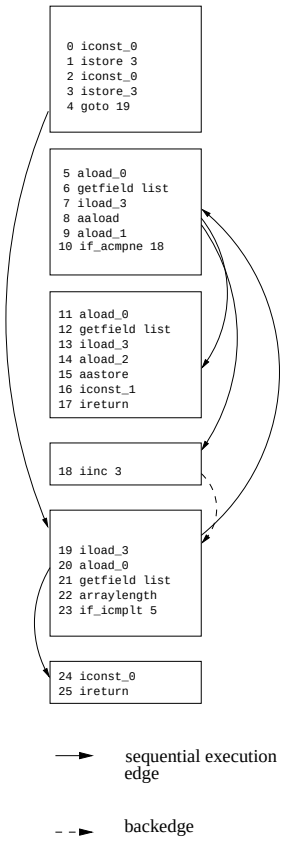

Figure 3.11: The control flow graph of the source program from Fig. 2.2

bytecode and exhibit conditions upon which a Java compiler can be proved to compile Java code correctly. In [94] Qian gives 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 [93] 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 [63], 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 preserves semantics and well-typedness. The small size and complexity of the JavaCard platform (the JVM version tailored to smart cards) simplifies the formalization of the system and thus, has attracted particularly the scientific interest. CertiCartes [19, 18] 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 [101] in terms of a small step operational semantics.

Chapter 4

Bytecode modeling language

This chapter presents the 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. The chapter is organized as follows. In Section 4.1, we start with discussing the basic design features of BML and its compiler. A detailed overview of the syntax and semantics of BML is given in Section 4.2. In Section 4.3, we discuss the issue of well typed BML specification. 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 4.4. We conclude the chapter with an overview of related works in Section 4.5.

4.1 Design features of BML

Before proceeding with the syntax and semantics of BML, we would like to discuss the design choices made in the encoding and the compiler of 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 independence Producing class files containing BML specification must not depend on any particular Java 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. 2.2 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.

BML corresponds to a partially desugared version of JML BML is designed to correspond to a partially desugared version of JML, namely BML supports a desugared version of the BML behavioral specification cases. We consider that such encoding makes the verification procedure more efficient. 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 limited resources.

We impose also few restrictions on the structure of the class file:

Line_Number_Table and Local_Variable_Table A requirement to the class file format is that it must contain the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [75], yet almost all standard non optimizing compilers can optionally 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. These attributes are usually used by debuggers as they describe the relation between source and bytecode. It is also necessary for the compiler from JML to BML, as we shall see later in Section 4.4.

Non-optimizing compilation The compilation process from JML to BML relies on finding the relation between source and its compilation into bytecode. As code optimization can make this relation to disappear, we require that the bytecode be produced by a non-optimizing compiler. Note that this is not a major restriction as most of the Java compilers do not support optimizations. Of course, this situation may change with the evolution of the Java platform which must be taken in future upgrades of BML. We provide a discussion on optimizing compilers in Chapter 10.

4.2 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 4.2.1 gives the notation conventions adopted here and Section 4.2.2 gives the formal grammar of BML as well as an informal description of its semantics.

4.2.1 Notation convention

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

4.2.2 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*

bv ::= **v_***intLiteral*

E ::= $constants_{bml}$
| **reg**(*digits*)
| *E*.*FieldConstRef*
| *ident*
| **arrAccess**(*E*,*E*)
| *E* *op* *E*
| **cntr**
| **st**(*E*)
| **old**(*E*)
| **EXC**
| **result**
| *bv*
| **typeof**(*E*)
| **type**(*ident*)
| **elemtype**(*E*)
| **TYPE**

FieldConstRef ::= *ident*

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

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

P ::= *E* $\mathcal{R}$ *E*
| *true*
| *false*
| not *P*
| $P \wedge P$
| $P \vee P$
| $P \Rightarrow P$
| $P \iff P$
| $\forall$ *bv*, *P*
| $\exists$ *bv*, *P*

classSpec ::= **invariant** *modifier* *P*
| **classConstraint** *P*
| **declare** **ghost** *ident* *ident*

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

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

assertion ::= *loopSpec*
| **assert** *P*
| **set** *E* *E*

loopSpec ::= **loop_invariant** P;
loop_modifies *modLocations*;
loop_decreases E;

methodSpec ::= **requires** P;
specCases;

specCases ::= *specCase*
| *specCase* **also** *specCases*

specCase ::= {|
requires P;
modifies *modLocations*;
ensures P;
exsuresList
|}

exsuresList ::= [ ] | **exsures** (*ident*) P; *exsuresList*

modLocations ::= [ ] | *modLocation*, *modLocations*

modLocation ::= E.*FieldConstRef*
| **reg**(i)
| *arrayModAt*(E, *specIndex*)
| **everything**
| **nothing**

specIndex ::= all| $E_1 \ldots E_2$ | E

4.2.3 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.

4.2.3.1 BML expressions

Among the common features of BML and JML are the following expressions: field access expressions E.*ident*, array access (**arrAccess**(E_1, E_2)), arithmetic expressions (E *op* E). Like JML, BML may talk about expression types. The BML expression **typeof(**E**)** denotes the dynamic type of the expression E, **type(***ident***)** is the class described at index *ident* in the constant pool of the corresponding class file. The construction **elemtype(**E**)** denotes the type of the elements of the array E, 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 (registers) 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.*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.

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.4.1 gives the bytecode as well as the BML specification of the code presented in Fig.2.4. 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. 2.4).

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.

```
1
2  Class instance invariant:
3     lv(0).#19 > 0;
4
5
6  Method specification:
7     requires lv(1) ≠ 0 ∨ lv(1) = 0;
8     {|
9      requires lv(1) ≠ 0;
10     modifies lv(0).#19;
11     ensures  lv(0).#19 = \old( lv(0).#19 ) / lv(1);
12     exsures  ( #25 ) false;
13    |}
14    also
15    {|
16     requires lv(1) == 0;
17     modifies \nothing;
18     ensures false;
19     exsures ( #26 ) lv(0).#19  = \old(lv(0).#19);
20    |}
21
22  public void divide(int lv(1))
23       0 load 0
24       1 dup
25       2 getfield #19 // instance field a
26       3 load 1
27       4 div
28       5 putfield #19 // instance field a
29       6 return
```

Figure 4.1: An example for a heavy weight specification in BML

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 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 characterize the states before and after an instruction execution. Stack expressions are represented in BML as follows:

- **cntr** represents the stack counter.
- **st**(E) stands for the element in the operand stack at position E. For instance, the element

below the stack top is represented with $\mathbf{st}(\mathbf{cntr} - 1)$ Note that those expressions may appear in predicates that refer to intermediate instructions in the bytecode.

4.2.3.2 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. 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 $\neq, =, \leq$ $, \leq \ldots$

4.2.3.3 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. 4.1 we can see the BML invariant resulting from the compilation of the JML specification in Fig. 2.4.

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.

4.2.3.4 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 *modLocation*. We look now what are the syntax constructs that may appear in the frame condition:

- $E.$*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
- $\mathbf{reg}(i)$ 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.
- $\mathit{arrayModAt}(E, \mathit{specIndex})$ states that the components at the indexes specified by *specIndex* in the array denoted by E may be modified. The indexes of the array components that may be modified *specIndex* have the following syntax:
  - i is the index of the component at index i. For instance, $\mathit{arrayModAt}(E, i)$ 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 $\mathit{arrayModAt}(E, \text{all})$ means that any element in the array may potentially be modified.
  - $E_1..E_2$ specifies the interval of array components between the index E_1 and E_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

4.2.3.5 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 [96] for a detailed specification of the JML desugaring process). A specification in BML may declare several method specification cases like in JML (*specCases*). The syntactic structure of a specification case is defined by the nonterminal *specCase*.

We illustrate this with an example in Fig. 4.1. In the figure, we remark that BML does not have the syntactic sugar for normal and exceptional behavior. On the contrary, the specification now states as precondition the disjunction of the preconditions of every specification case and the specification cases explicitly declare their behavior as described in subsection 2.2.4 from Chapter 2.2.

4.2.3.6 Intra — method specification

As we can see from the formal grammar in subsection 4.2.2, BML allows to specify a property that must hold at particular program point inside a method body. The nonterminal which describes the grammar of intra— method 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 (**atIndex**). BML supports specification constructs for loop entry points. The loop specification in BML given by the nonterminal *loopSpec* contains the loop invariant predicate (**loop_invariant**), the list of modified locations (**loop_modifies**) and the decreasing expression (**loop_decreases**). The **atIndex** attribute for loop specification contains the index of the loop entry point instruction. Moreover, the language allows for assertions at arbitrary program point in a method's bytecode. The syntax for this is **assert** P which specifies the predicate P that must hold at the corresponding position in the bytecode. BML has also constructs for setting the values of the specification ghost variables. The **set** E E is a special expression that allows to set the value of a specification ghost variable. This means that the first argument must denote a reference to a ghost variable, while the second expression is the new value that this ghost variable is assigned to.

We illustrate by in example in Fig. 4.2 how BML loop specification looks like. The example represents the bytecode and the BML specification of the source program in Fig. 2.2. 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 keyword **loop_invariant** respects the syntax for BML expressions and predicates that we described above.

4.3 Well formed BML specification

In the previous Section 4.2, we gave the formal grammar of BML. However, we are interested in a strict subset of the specifications that can be generated from this grammar. In 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 **arrAccess**(E_1, E_2) must be such that E_1 is of array type and E_2 is of integer type.
- the field access expression $E.ident$ is such that E is of subtype of the class where the field described by the constant pool element at index *ident* is declared

```
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 4.2: An example for a loop specification in BML

- For any expression $E_1 op E_2$, E_1 and E_2 must be of a numeric type.
- in the predicate $E_1 r E_2$ where $r = \leq, <, \geq, >$ the expressions E_1 and E_2 must be of integer type.
- in the predicate $E_1 <: E_2$, the expressions E_1 and E_2 must be of type **TYPE** (which is the same as `java.lang.Class`).
- the expression **elemtype(***E***)** must be such that E has an array type.

Examples for structural constraint are :

- All references to the constant pool must be to an entry of the appropriate type. For example: the field access expression *E.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 Java bytecode verifier may perform the checks over BML specification against such kind of structural and type constraints. However, we have not worked on this problem and is a good candidate for future work. For the curious reader, it will be certainly of interest to turn to the Java Virtual Machine description [75] which contains the official specification of the Java bytecode verifier or to the existing literature on bytecode verification (see the overview article [73]).

4.4 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. 2.3 is given in Fig.4.3 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.

3. Desugaring of the JML specification
   The phase consists in converting the JML method heavy-weight behaviors and the light - weight non complete specification into BML specification cases. It corresponds to part of the standard JML desugaring as described in [96]. For instance, the BML compiler will produce from the specification in Fig.2.4 the BML specification given in Fig.4.1

```
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. Possible values are public, protected and private
- **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 4.3: Compilation of ghost variable declaration

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. 2.4 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.4.1.

   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 the compiled specification refers to JML ghost fields.

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[1] 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. 2.2 the java compiler will produce two lines in the

[1] 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 [9]

Line_Number_Table

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

Figure 4.4: **Line_Number_Table** FOR THE METHOD `replace` IN FIG. 2.2

Line_Number_Table which correspond to the source line **17** as shown in Fig. 4.4. The problem is that none of the basic blocks determined by instructions **2** and **18** contain the loop entry instruction of the compilation of the loop at line **17** in Fig. 2.2. Actually, the loop entry instruction in the bytecode in Fig. 4.2 (remember that this is the compilation in bytecode of the Java source in Fig. 2.2) 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 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.2.2 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[2].

Finally, the compilation of the postcondition of method `replace` is given in Fig. 4.5. From the postcondition compilation, one can see that the expression **result** has integer type and the equality between the boolean expressions in the postcondition in Fig.2.2 is compiled into logical equivalence.

[2]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

$$\mathbf{result} = 1$$

$$\Longleftrightarrow$$

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

Figure 4.5: The compilation of the postcondition in Fig. 2.2

```
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. It is of length 2 bytes (**u2**)
- **modifies**[]: The array of locations that may be modified. It is of length 2 bytes (**u2**)
- **invariant** : The predicate that is the loop invariant. It is a compilation of the JML formula in the low level specification language
- **decreases**: The expression which decreases at every loop iteration

Figure 4.6: Structure of the Loop Attribute

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 is given in Fig. 4.6. 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**). Every kind of method specification (e.g. method specification cases, intra method assertions like **assert** and **set**) is encoded in a similar way in a corresponding user defined attribute. Those attributes are attached to the corresponding method attribute in the class file. Note that the data structures describing methods in the class are generated by the java compiler (i.e. in the first phase of the compilation of BML specification described above). The documentation of the encoding format of JML in Java class files is provided as an appendix in the end of the document.

4.5 Related work

The idea of introducing annotations in the bytecode is actually not so recent. For instance, the possibility to check a property at run-time, using the `assert` construct, has been long adopted in the C programming language and recently also in Java (Java 1.5, see [50, §14.10]). Checking runtime simple assertions, as for instance that a reference is not null is certainly very useful as it allows for detecting bugs early in the development process. But in the presence of mobile code, the need for more powerful specification mechanisms on the executable or interpreted code arises. Moreover, it represents an interest for software audit which does not trust the compiler. Another reason to consider a rich specification bytecode language is the verification of programs which are written directly in bytecode.

Although verification of bytecode programs has started to attract the scientific interest, not too much work has been done in the direction of bytecode specification. Several logics have been developed to reason about bytecode but none of them discusses the issue about a formalism for expressing program properties. For instance, in [11] Bannwart & Müller propose a general purpose Hoare style bytecode logic which is proved correct and complete. They also show how to compile Hoare style derivations from the source language into Hoare style derivations over bytecode in order to cope with complex properties in a PCC scenarios. However, they do not introduce a formalism into which to express those program properties. Within the MRG project [6], a resource aware logic is designed for Grail, a bytecode language which combines functional and object oriented features. However, there also the main focus is the development of a sound proof system. BML comes in reply to the need of writing and encoding understandable specifications for bytecode.

The development of BML is clearly inspired by the development of the JML specification language [67]. Both JML and BML follow the Design by Contract principle introduced first in Eiffel [78]. JML is a rich specification language which supports many specification constructs. Note that the semantics of few of the specification constructs in BML is still under discussion. Currently, BML supports only a subset of JML corresponding basically to the so called Level 0 of JML [67] but which has a well established semantics. JML is "the most popular formal specification" of Java and thus we consider that supporting a relation between a Java bytecode specification language and JML is important.

The Extended Virtual Platform project[3] also is very close to JML. This project aims at developing a framework that allows to compile JML annotations, to allow run-time checking [4]. However, in contrast to our work, they do not intend to do static verification of bytecode programs. Moreover, the Extended Virtual Platform takes JML-annotated source code files as starting point, so it is not possible to annotate bytecode applications directly.

The Spec# programming system [15] is probably the system which is closest to BML as introduces in similarly the Design by Contract principles into the C# programming language. However, we consider that the design goals of both Spec# and BML are not exactly the same. Let us give a brief description of the architecture of the Spec# system. It consists of the following components: a programming language Spec#, a compiler from Spec# to CLR,(the .NET intermediate language [3]), a runtime checker and a static verification scheme based on an intermediate language BoogiePL and the static verifier Boogie [13]. In particular, Spec# is a superset of the programming language C#. Spec# allows for expressing method (preconditions, postconditions and frameconditions) and class contracts as well as class contracts (class invariants). Moreover, the language extends the programming type system with non - null types, field initializers and expose blocks for an object (those constructs embrace a block which may break the object's invariant and at the end of its execution must reestablish it). These additional elements of the language are enforced with runtime checks emitted by the Spec# compiler. Spec# programs can be verified dynamically and statically. Both verification styles work on CLR as the Spec# compiler produces not only intermediate interpreted code but also metadata which contains the translation of the source specifications. As we stated above, the design of Spec# is such that it alters the underlying programming language C#. This is especially suitable for producing a high level quality software as the language obliges programmers to respect a good programming discipline. However, this does not fit completely in a mobile code scenario as a specified and statically verified CLR code using the Spec# system,

[3]See `http://www.cs.usm.maine.edu/~mroyer/xvp/`.

should be produced only if the code has been originally written in Spec#. But restricting mobile code producers to use a special programming system may be a hard restriction. This is a difference between BML and Spec#. BML targets standard Java source programs and specifications written in JML which is not part of the Java language. Moreover, it is compatible with any non optimizing Java compiler and thus does not provide a major restriction over the producer of the source code.

Chapter 5

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.

The vcGen is tailored to the bytecode language introduced in Section 3.8 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 5.1, we discuss the assertion language which the formalization of the verification condition generator manipulates. The assertion language consists of a subset of the BML language extended with few new constructs. In Section 5.2, we show the data structures which encode the specification. In Section 5.3, we focus on the verification calculus. As we stated earlier, our verification condition generator is based on a weakest precondition (wp) calculus. However, a logic tailored to stack based bytecode should take into account particular bytecode features as for example the operand stack. Another particularity of the verification condition calculus is the propagation of verification conditions up to the program point similar to the definition of the weakest precondition calculus for the Java-like source language in Chapter 2. To do this, we define the weakest precondition predicate transformer in terms of two mutually recursive functions. The first one calculates the precondition of instructions over an intermediate predicate which should hold in between the current instruction and its successor. A predicate which must hold between an instruction and its successor depends on the precondition of the successor and the execution relation between the two instructions, namely it depends on if the successor is a loop entry or not. Section 5.4 gives an example for how the verification condition generator works. Finally, section 5.5 is an overview of the existing work in the domain.

5.1 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 5.

The assertion language presented here will abstract from most of the BML specification clauses described in Section 4.2. Our interest will be focused only on method and loop specification. Also, the assertion language presented here discards class invariants, history constraints because they boil down to method pre and postconditions. We shall also restrict our attention to expressions that refer only to one state, i.e. we do not consider **old** expressions. This aspect needs a formalization of heaps in intermediate states especially in the case of method invocation. This aspect is well studied for structured object oriented languages. Such a semantics and proof of a logic which deals

expressions referring to the initial state of method execution is presented in the thesis of Cees Pierik [90].

The rest of this chapter is organized as follows. Section 5.1.1 presents what is exactly the BML fragment of interest and its extensions. Section 5.2 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. Finally, Section 5.1.2 gives formal semantics of the assertion language.

5.1.1 The assertion language

The assertion language in which we are interested corresponds to the BML expressions (nonterminal E) and predicates (nonterminal P) 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 $E.\texttt{f}(\oplus E \rightarrow E)$.
- update array access expression $\textbf{arrAccess}(\oplus(E, E) \rightarrow E)(E, E)$

The semantics of those syntactic constructs has been explained already in subsection 2.4.1 on 19.

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 $\textbf{instances}(\mathit{RefVal})$. Informally, the semantics of the predicate $\textbf{instances}(\texttt{ref})$ where $\texttt{ref} \in \mathit{RefVal}$ means that the reference `ref` is allocated in the current state.

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. The assertion language discussed here supports only method pre and postconditions. Note that as we discussed earlier in Section 2.2 class invariants and history constraints can be encodes as pre and postconditions.

5.1.2 Interpretation

In this section, we shall focus on the semantics of formulas and expressions w.r.t. a state configuration. A subtle point in giving the evaluation rules for expressions and the interpretation of formulas is the fact that the evaluation is actually a partial function. What we mean by partiality is the existence of functions like division or dereferencing of a field, array indexing etc. To get a precise idea of the problem we can consider the following logical statements:

$$\textit{(1)}\ E.\texttt{f} == 3$$
$$\textit{(2)}\ \textbf{arrAccess}(E_1, E_2) == 5$$

Under certain conditions, these formulas may not have a meaning. In case *(1)*, the statement does not make sense in a state where E is a reference which does not belong to the valid references in the heap or evaluates to **null**. In case *(2)* the statement does not make sense in a state where if either E_1 is not a valid reference in the current or is **null** or E_2 is not in the bounds of E_1.

Building a logic for partiality is not trivial. Different solutions exist. A naive three valued logic, where expressions may be evaluated to a value or undefined in a state and formulas might be *false*,

true or undefined in a state appear to loose certain nice properties which standard logic with equality have as for instance associativity of equality, the excluded middle [51] etc. Gries and Schnieder [51] give a solution which consists in function underspecification and thus avoid the problem of undefideness. More particularly, their approach considers all functions as total but for argument values for which the function is actually not defined the function may return whatever value. For instance, using the semantics of underspecification, the formula **null**.f == **null**.f will evaluate to true. This approach is adopted in JML [67]. However, seeing the expressions in specifications as totally defined functions may sometimes lead to unsoundness in program verification. Thus, not all tools supporting JML support the semantics of underspecification. An alternative is to check specifications for if they are well formed. For instance, ESC/java supports well-definedness checks of specifications.

In [26], L. Burdy proposes a three valued logic in which the above features of classical logic are preserved. He introduces a well-definedness operator $\Delta() : E \cup P \rightarrow P$ over expressions and formulas. Thus, formulas can be either *true* , *false* or not defined and expressions may evaluate to a value or be undefined. The operator $\Delta(E)$ $(\Delta(P))$ gives the necessary and sufficient conditions such that $E(P)$ is defined. Particularly, the application of the operator $\Delta(E)$ $(\Delta(P))$ over E (P) evaluates only either to *true* or *false*, i.e. it holds in a state only if the expression E has a value in this state, otherwise it is interpreted to *false*.

This in particular, means that for every formula and its subexpressions additional verification conditions for well-definedness must be generated. Thus a formula may hold in a state only under the condition that it is defined, i.e. all the expression contained in the formula should have a meaning in this state. Here, we shall not enter in details of the definition of the well definedness operator $\Delta()$ but discuss it informally through an example. Let us see the definition of the operator $\Delta()$ in case of an array access:

$$\Delta(\mathbf{arrAccess}(E_1, E_2)) = \begin{array}{l} \Delta(E_1) \wedge \\ \Delta(E_2) \wedge \\ E_1 \neq \mathbf{null} \wedge \\ 0 \leq E_2 < \mathtt{arraylength}(E_1) \end{array}$$

Thus for an array access to be well formed, we require that its two subexpressions are well formed, that E_1 denotes a reference in the heap in the current state, that the evaluation of E_2 is a valid index in the array E_1.

Although the evaluation of expression is partial, we shall consider only the validity of well defined formulas in a state, i.e. formulas which talk only about well defined expressions in a state. Thus we define interpretation of formulas in a two valued logic.

We ignore a lot of formulas, i.e. those which are not well defined, this will be sufficient for our proof of correctness of the verification condition generator later in Chapter 6 as we assume that programs are bytecode verified and that specifications are well defined. Under these assumptions and because the verification condition generator preserves well definedness of formulas, we can perform the correctness proof in Chapter 6.

We use the notation $s \vDash P$ to say that the well defined formula P is valid w.r.t. a current state s and an initial state s_0.

The interpretation $s \vDash P$ is defined in the standard way. For instance, the statement $s \vDash E_1 \mathcal{R} E_2$ by definition means that the evaluation $[\![E_1]\!]_s$ of E_1 and the evaluation $[\![E_2]\!]_s$ of E_2 are in relation $\mathcal{R}$ or written formally that the following holds: $[\![E_1]\!]_s \mathcal{R} [\![E_2]\!]_s$. The formal statement $s \vDash \mathbf{instances}(\mathtt{ref})$ by definition means that the reference `ref` is part of the heap locations in state s or in other words that the following holds $\mathtt{ref} \in s.\mathrm{H.Loc}$.

We define a function for expression evaluation which evaluates expressions in a state and which has the following signature:

$$[\![*]\!]_* : E \rightarrow S \rightarrow S \rightharpoonup \mathit{Values} \cup \mathit{JType}$$

The evaluation function takes an expression of the assertion language presented in the previous Section 5.1.1 and a state (see Section 3.4 for the definition of state) and returns a value or a class type (see Section 3.3 for the definition of values and types in the language).

Definition 5.1.1 (Evaluation of expressions). *The evaluation in a state* $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *or* $s =< \mathrm{H}, \mathit{Final} >^{final}$ *of an expression* E *w.r.t. an initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *is denoted with* $[\![E]\!]_s$ *and is defined in its domain inductively over the grammar of expressions* E *as follows:*

$$
\begin{array}{lcl}
[\![v]\!]_s & = & v \\
\textit{where } v \in \textbf{int} \ \vee \ v \in \mathit{RefVal} & & \\
[\![E.\mathtt{f})]\!]_s & = & \mathrm{H}(\mathtt{f})([\![E]\!]_s) \\
[\![E_3.\mathtt{f}(\oplus E_1 \rightarrow E_2)]\!]_s & = & \mathrm{H}(\oplus \mathtt{f} \rightarrow \mathtt{f}(\oplus [\![E_1]\!]_s \rightarrow [\![E_2]\!]_s))(\mathtt{f})([\![E_3]\!]_s) \\
[\![\textbf{arrAccess}(E_1, E_2)]\!]_s & = & \mathrm{H}([\![E_1]\!]_s, [\![E_2]\!]_s) \\
[\![\textbf{arrAccess}(\oplus(E_1, E_2) \rightarrow E_3)(E_4, E_5)]\!]_s & = & \mathrm{H}(\oplus([\![E_1]\!]_s, [\![E_2]\!]_s) \rightarrow [\![E_3]\!]_s)([\![E_4]\!]_s, [\![E_5]\!]_s) \\
[\![\textbf{reg}(i)]\!]_s & = & \mathrm{Reg}(i) \\
[\![E_1 \ op \ E_2]\!]_s & = & [\![E_1]\!]_s \, op \, [\![E_2]\!]_s \\
[\![\textbf{typeof}(E)]\!]_s & = & \begin{cases} \textbf{int} & [\![E]\!]_s \in \textbf{int} \\ \mathsf{TypeOf}\ ([\![E]\!]_s) & else \end{cases} \\
[\![\textbf{elemtype}(E)]\!]_s & = & \mathtt{T} \quad where\ \mathsf{TypeOf}\ ([\![E]\!]_s) = \mathtt{T[\,]} \\
[\![\textbf{TYPE}]\!]_s & = & \mathtt{java.lang.Class}
\end{array}
$$

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

$$
\begin{array}{lcl}
[\![\textbf{cntr}]\!]_s & = & \mathrm{Cntr} \\
[\![\textbf{st}(E)]\!]_s & = & \mathrm{St}([\![E]\!]_s)
\end{array}
$$

The evaluation of the following expressions can be done only in a final state:

$$
\begin{array}{ll}
[\![\textbf{result}]\!]_s = \mathrm{Res} & where\ s =< \mathrm{H}, \mathrm{Res} >^{norm} \\
[\![\textbf{EXC}]\!]_s = \mathrm{Exc} & where\ s =< \mathrm{H}, \mathrm{Exc} >^{exc}
\end{array}
$$

The next definition introduces the notion of an assertion formula logically valid in every program state. We call such formulas valid formulas.

Definition 5.1.2 (Valid formulas). *We say that the formula* P *a valid formula if forall states* s $s \vDash Ps$*. We note this with* $\vDash P$

5.2 Extending method declarations with specification

In the following, we propose an extension of the method formalization given in Section 3.2. The extension takes into account the method specification. The extended method structure is given below:

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

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 locations 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 4, section 4.2. 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 result of the function m.excPostSpec for SExc is m.excPostSpec(SExc) = **reg**(1) == **null**. If for an exception Exc there is not specified explicitly an exsures clause then the function excPostSpec returns the default exceptional postcondition predicate **false**, i.e. m.excPostSpec(Exc) = **false**
- m.loopSpecS is a list of **LoopSpec** data structures which give the specification information for a particular loop in the bytecode

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

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

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

- the field m.loopSpecS[k].pos is a valid index in the body of m: $0 \leq$ m.loopSpecS[k].pos $<$ m.body.$length$ and is a loop entry instruction in the sense of Def.3.9.1
- m.loopSpecS[k].invariant is the predicate that must hold whenever the instruction m.body[m.loopSpecS[k].pos] is reached in the execution of the method m
- m.loopSpecS[k].modif are the locations such that for any two states $state_1$, $state_2$ in which the instruction m.body[m.loopSpecS[k].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 $state_1 =^{\mathsf{modif}} state_2$

5.3 Weakest precondition calculus

Now that we have introduced the assertion language of the verification condition generator as well as the encoding of the method specification in the method data structure, we can turn to the definition of the weakest predicate transformer function which underlines the verification condition generator.

Thus, 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:

$$wp : int \longrightarrow \textbf{Method} \longrightarrow P$$

The function wp takes two arguments : the second argument is the method m to which the instruction belongs and the first argument is a point in the body of m.

Let us first see what is the desired meaning of wp. Particularly, we would like that the function wp returns a predicate $wp(i\ ,\texttt{m})$ 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.excPostSpec(Exc) holds where the function excPostSpec was introduced in Section 5.2. Thus, the wp function takes

into account both normal and exceptional program termination. The truthfulness of the predicate returned by the *wp* function may only guarantee that the postcondition holds under the assumption that the program terminates.

In the following, we will give an intuition to the way in which we have defined our verification condition generator. Consider the example in Fig. 5.1 which shows both the source code and the bytecode of a method [1] which calculates the sum of all the natural numbers smaller or equal to the parameter k. The source and bytecode are annotated, the first one in JML and the latter in BML. However, the bytecode annotations are actually stored separately from the bytecode instructions as we have described in Section 5.2 but we have put them explicitly in the bytecode at the point where they must hold for the sake of clarity. We have also marked the instructions which are identified as loop start and end according to Def.3.10 in Chapter 3.8, Section 3.9.

```
1  //@requires reg(1)>=0
2  0 const 0
3  1 store 2
4  2 const 0
5  3 store 3
6  4 goto 10
7  5 load 2
8  6 load 3
9  7 add
10 8 store 2
11 9 iinc 3 //LOOP END
12 //@loop_modifies    reg(2) , reg(3)
13 //@loop_invariant   I :  reg(3) ≥ 0 ∧ reg(3) ≤ reg(1) ∧ reg(2) == reg(3) ∗ (reg(3) − 1)/2
14 10 load 3 //LOOP ENTRY
15 11 load 1
16 12 if_icmplt 5
17 13 return
18 //@ensures  result== reg(1) *( reg(1) + 1)/2
```

```
1  //@requires k >= 0 ;
2  //@ensures \result == k∗(k+1)/2;
3  public void m(int k){
4    int sum = 0;
5    //@loop_modifies  sum, i;
6    //@loop_invariant  i≥0 ∧ i≤k ∧ sum == i∗(i−1)/2;
7    for (int i = 0;i < k;i++){
8      sum = sum + i;
9    }
10 }
```

Figure 5.1: BYTECODE OF METHOD SUM AND ITS SPECIFICATION

It is worth first to note that because the bytecode is not structured we cannot define the weakest precondition in the same way in which a predicate transformer for structured languages is defined. We will rather define the predicate transformer for instructions that may have one possible successor to depend on this successor:

$$wp(j) = S_k(wp(k)), where\ j \longrightarrow k$$

where S_k stands for a function which might be the identity function or a function which applies some substitution over its argument. We would proceed in a similar way with instructions that may branch - instructions which may jump (`goto` and `if_`*cond*) as well as instructions which may throw an exception (e.g. `putfield`, `astore`), but this time the predicate transformer for them depends on all of its successors:

$$wp(j) = \bigwedge_k C_k \Rightarrow S_k(wp(k)), where\ j \longrightarrow k$$

The predicates C_k stand for some condition to be filled in order that after the execution of instruction at index j the instruction at index k is executed. Note that here, in this example, we are using

[1] Here, we prefer to use almost the same syntax for source and bytecode specifications. For instance, alhough JML syntax encodes conjunction as && we shall rather use the mathematical symbol ∧ as is done in BML

a loose notation for *wp* function, as we only depends here on one parameter, namely the index of the instruction for which we calculate the weakest precondition.

Returning back to our example in Fig. 5.1, the weakest precondition for the instruction at index 12 (in the bytecode version) which is a conditional jump of the program will be:

$$\begin{aligned} wp(12) = \quad & \mathbf{st}(\mathbf{cntr}-1) < \mathbf{st}(\mathbf{cntr}) \Rightarrow wp(13)[\mathbf{cntr}\backslash\mathbf{cntr}-2] \\ & \wedge \\ & \mathbf{st}(\mathbf{cntr}-1) >= \mathbf{st}(\mathbf{cntr}) \Rightarrow wp(5)[\mathbf{cntr}\backslash\mathbf{cntr}-2], \\ & \textit{where } 12 \longrightarrow 13, 12 \longrightarrow 5 \end{aligned}$$

Let us see now what we would expect about the result of the function *wp* when applied to the instructions that have as successor the loop entry instruction at index 10. For instance, we can look at the instruction at index 9 which is marked in the figure as the end of the loop. As we said earlier we have inlined annotations in the bytecode at the places where they must hold. Thus after the execution of the instruction at index 9 the loop invariant must hold. It follows then that for a loop end instruction we will rather require that the *wp* function takes into account the corresponding loop invariant:

$$\begin{aligned} wp(9) = \quad & I[\mathbf{reg}(3)\backslash\mathbf{reg}(3)+1], \\ & \textit{where } 9 \longrightarrow^{l} 10 \end{aligned}$$

The situation is similar for the instruction at index 4 which jumps to the loop entry instruction at index 10. The semantics of the invariant requires that in the state after the execution of instruction at index 4 and before the execution of the instruction at index 10 the loop invariant must hold and second, whatever are the values of the program variables that might be modified by the loop, the invariant should imply the precondition of the loop entry instruction at 10. Thus we would like that the function *wp* gives us something like:

$$\begin{aligned} wp(4) = \quad & I \\ & \wedge \\ & \forall \mathbf{reg}(2), \mathbf{reg}(3), I \Rightarrow wp(10), \\ & \textit{where } 4 \longrightarrow 10 \ \wedge 10 \textit{ is a loop entry} \end{aligned}$$

The example shows that the function *wp* depends on the semantics of the instruction for which it calculates a precondition and also on the execution relation it has with its successors. In order to define the function *wp* we will use an intermediate function which shall decide what is the postcondition of an instruction upon the execution relation with its successors. This function is introduced in the next subsection 5.3.1. We will also see how the weakest precondition is defined in the presence of exceptions in subsection 5.3.2.

Note also that the calculation of the *wp* predicate may be done in a forward or backwards direction. By backwards direction, we mean that the calculation starts from the "ends" of the control flow graph, i.e. from the `return` and `athrow` instructions and goes in a backwards directions to the predecessors up to reaching the program entry point. A forward calculation starts from the entry point instruction and goes in a forward direction to the successors up to reaching the return instructions or athrow instructions of the control flow graph.

5.3.1 Intermediate predicates

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

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

The predicate $inter(j, k, \mathtt{m})$ will be used for determining the weakest predicate that must hold in the poststate of the instruction j in the execution path where `j` is followed by the instruction k. This predicate depends on the execution relation between the two instructions j and k. The function *inter* allows to generate correct verification conditions for loops. For this, *inter* does a

case analysis over the relation $\longrightarrow$ (introduced in Chapter 3.8, Section 3.9) between the current instruction and its successor. If the relation is a $\longrightarrow^l$, i.e. the next instruction is a loop entry instruction and the current is a loop end (as defined in Def.3.9.1) then the predicate that must hold in between the current instruction and the next one is the loop invariant associated to the loop entry. The rest of the cases are two. In case that the next instruction is a loop entry then two conditions must hold in between it and the current instruction. First, the invariant must hold there and second, the invariant must imply the weakest predicate of the loop entry instruction. For the last case, when the next instruction is not a loop entry, we get that the predicate that must hold between the current instruction and the successor instruction is the weakest precondition of the successor.

Definition 5.3.1.1 (Intermediate predicate between two instructions). *Assume that instructions j and k may execute one after another, i.e. $j \longrightarrow k$. The predicate $inter(j, k, \mathtt{m})$ must hold after the execution of j and before the execution of k and is defined as follows:*

- *if k is a loop entry and j is a loop end, i.e. $j \longrightarrow^l k$ then there exists an index s in the loop specification table* $\mathtt{m.loopSpecS}$ *such that* $\mathtt{m.loopSpecS}[s].\mathsf{pos} = k$ *then the corresponding loop invariant must hold:*
$$inter(j, k, \mathtt{m}) = \mathtt{m.loopSpecS}[s].\mathsf{invariant}$$
- *else if k is a loop entry then there exists an index s in the loop specification table* $\mathtt{m.loopSpecS}$ *such that* $\mathtt{m.loopSpecS}[s].\mathsf{pos} = k$ *and the corresponding loop invariant* $\mathtt{m.loopSpecS}[s].\mathsf{invariant}$ *must hold before k is executed, i.e. after the execution of j. We also require that* $\mathtt{m.loopSpecS}[s].\mathsf{invariant}$ *implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations* $\mathtt{m.loopSpecS}[s].\mathsf{modif}$ *that may be modified in the loop body:*

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

- *else*
$$inter(j, k, \mathtt{m}) = wp(k\ , \mathtt{m})$$

5.3.2 Weakest precondition in the presence of exceptions

Our weakest precondition calculus deals with exceptional termination and thus, we need a way for calculating the postcondition of an instruction in case it terminates on an exception. In particular, the postcondition should depend on if there is an exception handler or not. In the first case, the execution continues at the exception handler entry point and thus the postcondition of the exceptionally terminating instruction will be the precondition of the instruction from which the exception handler starts. In the case where there is no exception handler, this means that the current method also terminates on exception and thus, the specified exceptional exceptional postcondition of the method for this exception should hold.

We define the function excPostIns with signature :

$$\mathsf{excPostIns} : int \longrightarrow \mathtt{ExcType} \longrightarrow P$$

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

Definition 5.3.2.1 (Postcondition in case of a thrown exception).

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

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

$$\texttt{m.excPostRTE} : int \longrightarrow \texttt{ExcType} \longrightarrow P$$

m.excPostRTE(i, Exc) returns the predicate that must hold in the prestate of the instruction at index i which may throw a runtime exception of type Exc. Note that the function m.excPostRTE does not deal with programmatic exceptions thrown by the instruction athrow, neither exception caused by a method invocation (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 the next subsection.

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

Definition 5.3.2.2 (Auxuliary function for instructions throwing runtime exceptions).

$$\begin{array}{l} i \neq \texttt{athrow} \wedge i \neq \texttt{invoke} \Rightarrow \\ \texttt{m.excPostRTE}(\ i,\ \texttt{Exc}) = \\ \forall \texttt{ref}, \\ \quad (\neg\ \textbf{instances}(\texttt{ref}) \wedge \\ \quad \texttt{ref} \neq \textbf{null} \\ \quad \textbf{typeof}(\texttt{ref}) =\ \texttt{Exc}) \Rightarrow \\ \quad \Big(\texttt{m.excPostIns}(\ i,\ \texttt{Exc})[\textbf{cntr}\backslash 0][\textbf{st}(0)\backslash \texttt{ref}][\texttt{f}\backslash \texttt{f}(\oplus \texttt{ref} \rightarrow \texttt{defVal}(\texttt{f.Type}))]^{\texttt{subtype}(\texttt{f.declaredIn},\ \texttt{Exc})}_{\texttt{f}}\Big) \end{array}$$

The function m.excPostRTE will return a predicate which states that for every newly created exception reference the predicate returned by the function excPostIns for the exception type Exc and program point i must hold.

5.3.3 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, $i =$ goto n

     $$wp(i\ , \texttt{m}) =\ inter(i, n, \texttt{m})$$

     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, $i =$ if_*cond* n

     $$\begin{array}{l} wp(i\ , \texttt{m}) = \\ \textbf{st}(\textbf{cntr})\ \mathit{cond}\ \textbf{st}(\textbf{cntr} - 1) \Rightarrow\ inter(i, n, \texttt{m})[\textbf{cntr}\backslash \textbf{cntr} - 2] \\ \wedge \\ \neg(\textbf{st}(\textbf{cntr})\ \mathit{cond}\ \textbf{st}(\textbf{cntr} - 1)) \Rightarrow\ inter(i, i+1, \texttt{m})[\textbf{cntr}\backslash \textbf{cntr} - 2] \end{array}$$

     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 $inter(i, n, \texttt{m})[\textbf{cntr}\backslash \textbf{cntr} - 2]$ 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 $inter(i, i+1, \texttt{m})[\textbf{cntr}\backslash \textbf{cntr} - 2]$.

  3. return, $i =$ return

     $$wp(\texttt{m}\ , i) = \texttt{m.normalPost}[\textbf{result}\backslash \textbf{st}(\textbf{cntr})]$$

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

- the skip instruction, $i = \texttt{nop}$

$$wp(i\ ,\texttt{m}) = \ inter(i, i+1, \texttt{m})$$

- load and store instructions

1. load a local variable on the operand stack, $i = \texttt{load j}$

$$wp(i\ ,\texttt{m}) = \\ inter(i, i+1, \texttt{m})[\mathbf{cntr}\backslash\mathbf{cntr}+1][\mathbf{st}(\mathbf{cntr}+1)\backslash\mathbf{reg}(j)]$$

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 **reg**(j). For instance, if we have that the predicate $inter(i, i+1, \texttt{m})$ is equal to $\mathbf{st}(counter) == 3$ then we get that the precondition of instruction is $\mathbf{reg}(\text{j}) == 3$:

$$\{\mathbf{reg}(\text{j}) == 3\} \\ i : \texttt{load j} \\ \{\mathbf{st}(\mathbf{cntr}) == 3\} \\ i+1 : \ldots$$

2. store the stack top element in a local variable $i = \texttt{store j}$

$$wp(i\ ,\texttt{m}) = \\ inter(i, i+1, \texttt{m})[\mathbf{cntr}\backslash\mathbf{cntr}-1][\mathbf{reg}(\text{j})\backslash\mathbf{st}(\mathbf{cntr})]$$

Contrary to the previous instruction, the instruction `store j` will take the stack top element and will store its contents in the local variable **reg**(j).

3. push an integer constant on the operand stack $i = \texttt{push j}$

$$wp(i\ ,\texttt{m}) = \\ inter(i, i+1, \texttt{m})[\mathbf{cntr}\backslash\mathbf{cntr}+1][\mathbf{st}(\mathbf{cntr}+1)\backslash\ \text{j}\ ]$$

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 $i = \texttt{iinc j}$

$$wp(\texttt{m}\ ,i) = \\ inter(i, i+1, \texttt{m})[\mathbf{reg}(j)\backslash\mathbf{reg}(\ \text{j})+1]$$

- arithmetic instructions

1. instructions that cannot cause exception throwing $i \in \{\texttt{add}, \texttt{sub}, \texttt{mult}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{ishr}, \texttt{ishl}\}$

$$wp(i\ ,\texttt{m}) = \\ inter(i, i+1, \texttt{m})[\mathbf{cntr}\backslash\mathbf{cntr}-1][\mathbf{st}(\mathbf{cntr}-1)\backslash\mathbf{st}(\mathbf{cntr})\texttt{op}\ \mathbf{st}(\mathbf{cntr}-1)]$$

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

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

2. instructions that may throw exceptions $i = \{\texttt{rem}, \texttt{div}\}$

$$
\begin{array}{l}
wp(i\ , \texttt{m}) = \\
\ \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad inter(i, i+1, \texttt{m})[\mathbf{cntr} \backslash \mathbf{cntr} - 1][\mathbf{st}(\mathbf{cntr} - 1) \backslash \mathbf{st}(\mathbf{cntr})\ \texttt{op}\ \mathbf{st}(\mathbf{cntr} - 1)] \\
\ \wedge \\
\ \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i, \texttt{ArithExc})
\end{array}
$$

- object creation and manipulation

1. create a new object, $i = \texttt{new Class}$

$$
\begin{array}{l}
wp(i\ , \texttt{m}) = \\
\forall \mathbf{bv}, \\
\left(\begin{array}{ll}
\neg\ \mathbf{instances}(\mathbf{bv}) \wedge \mathbf{typeof(bv)} = C \wedge \mathbf{bv} \neq \mathbf{null} \Rightarrow & \\
\qquad [\mathbf{cntr} \backslash \mathbf{cntr} + 1] & \\
\quad inter(i, i+1, \texttt{m})\ [\mathbf{st}(\mathbf{cntr} + 1) \backslash \mathbf{bv}] & \\
\qquad [\texttt{f} \backslash \texttt{f}(\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(\texttt{f.Type}))]_{\texttt{f}}^{\mathsf{subtype}(\texttt{f.declaredIn}, \texttt{Class})} &
\end{array} \right)
\end{array}
$$

The postcondition of the instruction `new` is the intermediate predicate $inter(i, i+1, \texttt{m})$. The weakest precondition of the instruction says that for any reference **bv** if **bv** is not an instance reference in the state before the execution of the instruction and whose type is `Class` then the precondition is the same predicate but in which the stack counter is incremented and **bv** is pushed on the stack top. The fields for the **bv** have the default value of their type which is expressed through series of substitutions $\mathsf{subtype}(\texttt{f.declaredIn}, \texttt{Class})$.

2. array creation, $i = \texttt{newarray T}$

$$
\begin{array}{l}
wp(i\ , \texttt{m}) = \\
\forall \texttt{ref}, \\
\left(\begin{array}{ll}
\mathit{not}\ \mathbf{instances}(\texttt{ref}) \wedge & \\
\texttt{ref} \neq \mathbf{null} \wedge \mathbf{typeof}(\texttt{ref}) = \mathbf{type}(\texttt{T}[]) \wedge \mathbf{st}(\mathbf{cntr}) \geq 0 \Rightarrow & \\
\qquad inter(i, i+1, \texttt{m}) & \\
\qquad [\mathbf{st}(\mathbf{cntr}) \backslash \texttt{ref}] & \\
\qquad [\mathbf{arrAccess} \backslash \mathbf{arrAccess}(\oplus(\texttt{ref}, j) \rightarrow \mathsf{defVal}(\texttt{T}))]_{\forall j, 0 \leq j < \mathbf{st}(\mathbf{cntr})} & \\
\qquad [\mathrm{arrLength} \backslash \mathrm{arrLength}(\oplus \texttt{ref} \rightarrow \mathbf{st}(\mathbf{cntr}))] & \\
\wedge & \\
\mathbf{st}(\mathbf{cntr}) < 0 \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i, \texttt{NegArrSizeExc}) &
\end{array} \right)
\end{array}
$$

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

3. field access $i = \texttt{getfield f}$

$$
\begin{array}{l}
wp(i\ , \texttt{m}) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow\ inter(i, i+1, \texttt{m})[\mathbf{st}(\mathbf{cntr}) \backslash \texttt{f}(\mathbf{st}(\mathbf{cntr}))] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i, \texttt{NullExc})
\end{array}
$$

The instruction for accessing a field value takes as postcondition the predicate that must hold between it and its next instruction $inter(i, i+1, \texttt{m})$. 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 $\texttt{f}(\mathbf{st}(\mathbf{cntr}))$. If the stack top element is **null**, then the instruction will terminate on a `NullExc` exception. In this case the precondition of the instruction is the predicate returned by the function $\texttt{m}.\mathsf{excPostRTE}$ for position i in the bytecode and exception `NullExc`

4. field update $i = \texttt{putfield f}$

$$wp(i\ ,\texttt{m}) =$$
$$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \ inter(i, i+1, \texttt{m})[\mathbf{cntr}\backslash\mathbf{cntr}-2][\texttt{f}\backslash\texttt{f}(\oplus\mathbf{st}(\mathbf{cntr}-1) \rightarrow \mathbf{st}(\mathbf{cntr}))]$$
$$\wedge$$
$$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i, \texttt{NullExc})$$

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 $inter(i, i{+}1, \texttt{m})$ must hold where the stack counter is decremented with two elements and the fobject is substituted with an updated version $\texttt{f}(\oplus\mathbf{st}(\mathbf{cntr}-2) \rightarrow \mathbf{st}(\mathbf{cntr}-1))$.

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

$$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow (\texttt{f}(\mathbf{reg}(1)) \neq \mathbf{null})[\mathbf{cntr}\backslash\mathbf{cntr}-2][\texttt{f}\backslash\texttt{f}(\oplus\mathbf{st}(\mathbf{cntr}-1) \rightarrow \mathbf{st}(\mathbf{cntr}))]$$
$$\wedge$$
$$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathbf{false}$$

After applying the substitution following the rules described in Section 2.4.1, we obtain that the precondition is

$$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \texttt{f}(\oplus\mathbf{st}(\mathbf{cntr}-1) \rightarrow \mathbf{st}(\mathbf{cntr}))(\mathbf{reg}(1)) \neq \mathbf{null}$$
$$\wedge$$
$$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathbf{false}$$

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

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

5. access the length of an array $i = \texttt{arraylength}$

$$wp(i\ ,\texttt{m}) =$$
$$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \ inter(i, i+1, \texttt{m})[\mathbf{st}(\mathbf{cntr})\backslash \mathrm{arrLength}(\mathbf{st}(\mathbf{cntr}))]$$
$$\wedge$$
$$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i, \texttt{NullExc})$$

The semantics of arraylengthis 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 $inter(i, i+1, \texttt{m})$ must hold where the stack top element is substituted with its length. The case when a NullExcis thrown is similar to the previous cases with exceptional termination

6. checkcast $i = \texttt{checkcast}\ C$

$$wp(i\ ,\texttt{m}) =$$
$$\mathbf{typeof(st(cntr))} <: C \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \ inter(i, i+1, \texttt{m})$$
$$\wedge$$
$$\neg(\mathbf{typeof(st(cntr))} <: C) \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i, \texttt{CastExc})$$

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.excPostRTE for the position i and exception `CastExc` must hold

7. instanceof $i =$ `instanceof` C

$$
\begin{array}{l}
wp(i\ ,\mathtt{m}) = \\
\mathbf{typeof(st(cntr))} <: C \Rightarrow\ inter(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr})\backslash 1] \\
\wedge \\
\neg(\mathbf{typeof(st(cntr))} <: C) \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow\ inter(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr})\backslash 0]
\end{array}
$$

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). $i =$ `invoke n`

$$
\begin{array}{l}
wp(i\ ,\mathtt{m}) = \\
\mathtt{n}\ .\mathsf{pre}[\mathbf{reg}(s)\backslash\mathbf{st}(\mathbf{cntr}+s-\mathtt{m}.\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \\
\\
\wedge \\
\\
\forall res, m(m \in \mathtt{n}\ .\mathsf{modif}) \\
\left(\begin{array}{l}
\mathtt{n}\ .\mathsf{normalPost}[\mathbf{result}\backslash res][\mathbf{reg}(s)\backslash\mathbf{st}(\mathbf{cntr}+s-\mathtt{n}\ .\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \Rightarrow \\
inter(i, i+1, \mathtt{m})[\mathbf{cntr}\backslash\mathbf{cntr}-\mathtt{n}\ .\mathsf{nArgs}][\mathbf{st}(\mathbf{cntr}-\mathtt{n}\ .\mathsf{nArgs})\backslash res]
\end{array}\right) \\
\\
\wedge_{j=0}^{\mathtt{n}\ .\mathsf{exceptions}.length-1} \\
\\
\wedge findExcHandler(\mathtt{n}\ .\mathsf{exceptions}[j], i, \mathtt{m}.\mathsf{excHndlS}) = \bot \Rightarrow \\
\forall e, m(m \in \mathtt{n}\ .\mathsf{modif}), \\
(\ \mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j])[\mathbf{EXC}\backslash e]\mathtt{m}.\mathsf{excPostIns}(i, \mathtt{m}.\mathsf{exceptions}[j])[\mathbf{EXC}\backslash e]\ ) \\
\wedge \\
(findExcHandler(\mathtt{m}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j]), i, \mathtt{m}.\mathsf{excHndlS}) = k \Rightarrow \\
\forall e, m(m \in \mathtt{n}\ .\mathsf{modif}), \\
\left(\begin{array}{l}
\mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j])[\mathbf{EXC}\backslash e] \Rightarrow \\
inter(i, k, \mathtt{m})[\mathbf{cntr}\backslash 0][\mathbf{st}(0)\backslash e][\mathtt{n}\ .\mathsf{modif}[i]\backslash bv_i]_{i=0}^{\mathtt{n}\ .\mathsf{modif}.length}
\end{array}\right)
\end{array}
$$

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

Second, we get a logical statement which guarantees the correctness of the method invokation in case of normal termination. On the other hand, its postcondition n.normalPost is assumed to hold and thus, we want to establish that under the assumption that m.normalPost holds with **result** substituted with a fresh bound variable res 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. We denote the quantification with $m(m \in \mathtt{n}\ .\mathsf{modif})$ to say that we quantify over the locations which are in the modifies list. 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 and before k must hold once again for any value of thrown exception.

- throw exception instruction, i = athrow

$$
\begin{array}{l}
wp(i\ ,\mathtt{m}) = \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostRTE}(i, \mathtt{NullExc}) \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \ \ \forall \mathtt{Exc}, (\mathbf{typeof(st(cntr))} <: \mathtt{Exc} \Rightarrow \mathtt{m}.\mathsf{excPostIns}(i, \mathtt{Exc})[\mathbf{EXC} \backslash \mathbf{st}(\mathbf{cntr})])
\end{array}
$$

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 terminates on an exception NullExc where the predicate returned by the function m.excPostRTE(i, NullExc) 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

5.3.4 Verification conditions

Let us see now how with the help of the *wp* we can express formulas whose validity implies the correctness of a program in our language. We shall express program correctness in terms of correctness of methods declared in the classes of a program. In particular, the verification conditions for a program is the set of the verification conditions for all the classes in P. The verification conditions for a class are the verification conditions for all methods declared in a class. Method correctness is understood as the compliance between the specified method contract (pre and postconditions) and method implementation. Moreover, if a method in a class overrides a method declared in a super type the correctness of the overriding method requires that it behaves as the method it overrides.

Thus, in the following, we take a closer look at the formulas that concern method correctness.

5.3.4.1 Method implementation respects method contract

Supposing the execution of a method always terminates, the verification condition which expresses the fact that a method m respects its specification states that the specified method precondition m.pre implies the predicate $wp(0\ ,\mathtt{m})$ calculated over the body of m. Moreover, we have few more assumptions concerning the register (local) variables which store the receiver object and the method parameters. Those assumptions actually reflect properly the semantics of our programming language. Thus the verification condition will be as shown in the following:

Definition 5.3.1. *Let us have a method* m *with a precondition* m.pre*, postcondition* m.normalPost *and exceptional postcondition function* m.excPostSpec*. The verification condition which expresses that the method respects its specification is the following:*

$$\mathtt{m}.\mathsf{pre} \wedge \mathbf{reg}(0) \wedge \mathbf{instances}(\mathbf{reg}(0)) \wedge \neq \mathbf{null} \wedge \mathsf{locVarWD}(\mathtt{m}) \wedge \mathsf{locVarWT}(\mathtt{m}) \Rightarrow wp(0\ ,\mathtt{m})$$

As we can see, the verification condition assumes not only the specified method precondition but also that the local variable $\mathbf{reg}(0)$ which stores the current object is not **null** and that it contains a reference to an object in the heap. Actually, this is a natural condition as because of the

semantics of our language method may be executed only on a non null reference which belongs to the current heap. Next, we also take as a hypothesis the formula locVarWD which is parameterized by the method m and which stands stands for:

$$\mathsf{locVarWD}(\mathtt{m}) = \forall i.1 \leq i \leq \mathtt{m.nArgs}, \mathtt{m.argsType}[i] <: \mathtt{Object} \Rightarrow \mathbf{instances}(\mathbf{reg}(i)) \vee \mathbf{reg}(i) = \mathbf{null}$$

Thus we constrain the values of the local variables which are of reference type(i.e. are of type which is a subtype of the root class `Object` of all reference types) to be either a reference to an object allocated in the heap or **null**. This is also in correspondence with the semantics of the language, which manipulates either references which are in the domain of the heap or are **null**.

The next formula in the assumption of the verification condition is locVarWD also parameterized with m which stands for the following:

$$\mathsf{locVarWT}(\mathtt{m}) = \forall i.0 \leq i \leq \mathtt{m.nArgs}, \mathbf{typeof}(\mathbf{reg}(i)) <: \mathtt{m.argsType}[i]$$

Thus we assume that any local variable $\mathbf{reg}(i), 0 \leq i \leq \mathtt{m.nArgs}$ (including the local variable $\mathbf{reg}(0)$ which stores reference to the current object) is assumed to be with the expected type $\mathbf{reg}(i) <: \mathtt{m.argsType}[i]$ with which it is declared.

Consider for instance, the simple method which assigns the value 3 to a field `f` of the receiver object stored in the method register $\mathbf{reg}(1)$.

```
1  A {
2   int  f;
3
4  //@ requires  true
5  //@ ensures   reg(0).f = 3
6  //@ exsures (Exception)  false
7   m (  )
8     load 0
9     const 3
10    putfield  f
11    return
12 }
```

This method respects its specification as in its body it assigns the value 3 to the field `f` of the current object. Its verification condition is:

$$\begin{array}{l}(\mathbf{true} \wedge \mathbf{reg}(0) \neq \mathbf{null} \wedge \mathsf{locVarWD} \wedge \mathsf{locVarWT}) \Rightarrow \\ \mathbf{reg}(0) \neq \mathbf{null} \Rightarrow \mathbf{reg}(0).\mathtt{f}(\oplus \mathbf{reg}(0) \rightarrow 3) = 3 \\ \wedge \\ \mathbf{reg}(0) = \mathbf{null} \Rightarrow \mathbf{false}\end{array}$$

We can see that the verification condition also holds. The first conjunct in it corresponds to a normal termination of the method and holds because the left hand side of the equality simplifies to 3. The second conjunct concerns the exceptional termination of the method (in case the dereferenced object by the instruction `putfield` is **null**). This case also holds because of the contradiction of in the hypothesis where $\mathbf{reg}(0)$ is both equal and different from **null**.

5.3.4.2 Behavioral subtyping

As we said above, in an object oriented language with subclassing and method overriding, the notion of method correctness must also include the fact that the behavior of an overriding method conforms with the semantics of the method it overrides.

We illustrate the importance of this issue by the following example:

```
1  A {
2  //@ requires  Pre1;
3  //@ ensures   Post1;
```

```
4  int m ( ){...}
5  }
6
7  B extends A{
8  //@ requires Pre1;
9  //@ ensures  Post1;
10  m ( ){...}
11 }
12
13 C {
14  n(A a) {
15  int a.m();
16  }
17 }
```

In the example, the class B extends the class A and overrides the method m. The last part of the example shows a method n declared in class C which makes a call to m over the method parameter a declared with static type the class A. If we had to verify this method call, as we saw in the previous section we would use the specification of method m declared in A, i.e. the precondition Pre1 and the postcondition Post1. Unfortunately, this is not sufficient to establish the correctness of the method n as the dynamic type of the parameter a might be B and we do not have any guarantee that the overriden method and its specification makes the verification condition for method n valid.

One way to cope with this situation is to generate verification conditions at every method call site for all the methods and their specifications that override the called method (methods with the same signature declared in the subclasses of the static type of the object over which the method call is done) respect. But this solution is not modular as it requires to reverify the whole program every time a new subclass extension is made in the program. Such approach is taken in [91].

We adopt here an alternative solution which consists in the following. If a method `m` declared in a class B overrides method A from the super class `B` of `A`, the specification of method `m` must conform with the specification of method `n`: This is expressed by the two conditions (contravariant and covariant) over their pre and postconditions:

- the precondition of the overriden method `n` must imply the precondition of the overriding method `m`. Intuitively, this means that where the overriden method is called the overriding method can be also called.

- the postcondition of the overriding method `m` must imply the postcondition of the overriden method `n`. This must be true for normal postcondition and every exceptional postcondition case. This means that the overriding method guarantees stronger properties than the overriden method

In the next, we show the conditions for establishing the correctness of a method `m` if method `m` overrides method `n`:

Definition 5.3.2. *Verification conditions for correct subtyping The verification condition which express that a method* `m` *which overrides method* `n` *is a behavioral subtype is given in the following:*

$$\forall bv_0, \ldots, bv_{\mathtt{n.nArgs}},$$
$$\left(\begin{array}{l} \left(\begin{array}{l} \mathtt{n.pre} \wedge \mathbf{reg}(0) \neq \mathbf{null} \wedge \mathbf{instances}(\mathbf{reg}(0)) \wedge \\ \mathsf{locVarWD}(\mathtt{n}) \wedge \mathsf{locVarWT}(\mathtt{n}) \end{array} \right) \Rightarrow \mathtt{m.pre} \end{array} \right) [\mathbf{reg}(i) \backslash bv_i]_{i=0}^{\mathtt{n.nArgs}}$$

$$\forall bv, bv_0, \ldots, bv_{\mathtt{n.nArgs}},$$
$$\left(\begin{array}{l} \left(\begin{array}{l} \mathtt{m.normalPost}[\mathbf{result} \backslash bv] \wedge bv <: \mathtt{n.retType} \wedge \\ \mathbf{reg}(0) \neq \mathbf{null} \wedge \mathbf{instances}(\mathbf{reg}(0)) \wedge \\ \mathsf{locVarWD}(\mathtt{n}) \wedge \mathsf{locVarWT}(\mathtt{n}) \wedge \\ \mathbf{instances}(\mathbf{reg}(0)) \end{array} \right) \Rightarrow \mathtt{n.normalPost}[\mathbf{result} \backslash bv] \end{array} \right) [\mathbf{reg}(i) \backslash bv_i]_{i=0}^{\mathtt{n.nArgs}}$$

$$\forall bv, bv_0, \ldots, bv_{\mathtt{n.nArgs}},$$
$$\left(\begin{array}{l} \left(\begin{array}{l} \mathtt{m.excPostSpec(Exc)}[\mathbf{EXC} \backslash bv] \wedge \\ \mathbf{instances}(bv) \wedge \\ bv <: \mathtt{Exc} \wedge \\ \mathbf{reg}(0) \neq \mathbf{null} \wedge \\ \mathbf{instances}(\mathbf{reg}(0)) \wedge \\ \mathsf{locVarWD} \wedge \mathsf{locVarWT} \end{array} \right) \Rightarrow \mathtt{n.excPostSpec(Exc)}[\mathbf{EXC} \backslash bv] \end{array} \right) [\mathbf{reg}(i) \backslash bv_i]_{i=0}^{\mathtt{n.nArgs}}$$

where $\mathtt{Exc} \in \mathtt{n.exceptions}$

As in the verification conditions conditions concerning only method specification above, they all use additional assumption concerning the types and values of the local variables. As explained above, the first condition expresses the fact that the precondition of the overriding method is stronger than the precondition it overrides. This is the so called contravariant rule. Note that the implication must hold for any valid value of the local variables (i.e. for those of reference type they must be valid references in the heap and moreover, the receiver of the call must be not **null**)

The second condition expresses the fact that the normal postcondition of the overriding method is stronger than than the postcondition specified in the method it overrides. We quantify over the postcondition result as well as the values of the local variables including the receiver of the method call **reg**(0) . The third formula represents a series of formulas which show that the postcondition for any exceptional outcome of the overriding method on any exception type `Exc` declared in `n.exceptions`must respect the condition of the overriden method. Similarly, we want that the implication holds for any values of the local variables and of the exception object which of the expected exception type `Exc`.

Note that here, because we do not treat **old** expressions, the verification condition is simple w.r.t. verification conditions which might take into account the initial state. The latter involves a modeling of the heap which expresses the relation between the initial and final states of a method execution. A detailed explanation for how to construct such verification conditions may refer to [90].

5.4 Example

In the following, we shall see what are the resulting preconditions that the *wp* will calculate for the instructions in the bytecode from the program in Fig. 5.1.

Fig.5.2 shows the weakest preconditions for some of the instructions in the bytecode of the method sum. In the figure, the line before every instruction gives the calculated weakest precondition of the instruction. Thus, the weakest precondition of the instruction `return` at line 74 states that before the instruction is executed the stack top element **st**(**cntr**) must contain the sum of the natural numbers smaller than the local variable **reg**(1) . This precondition is calculated from the method postcondition which is given in curly brackets at line 75.

The instruction preceding the `return` instruction is a conditional branch which may jump to instruction at line 44 (or at position 5 in the bytecode array). This instruction has as precondition a predicate which reflects the two possible choices after it: if the element below the stack top **st**(**cntr**-1) is smaller than the stack top element **st**(**cntr**) then the precondition P5 of the instruction at

line 44 must hold, otherwise the precondition Pre13 of the instruction at line 74 holds. For every instruction which does not targets a loop entry instruction the precondition is calculated from the precondition of its successor instructions. The special cases are the instructions at lines 37 and 56 which point to the loop entry instruction at line 61. As described earlier we can see that the resulting precondition of the instruction at line 56 is calculated upon the loop invariant. The precondition of the instruction at line 37 is calculated also upon the loop invariant but also confirms that the invariant implies the precondition of the loop entry instruction.

Finally, we can remark that the verification condition for the method $Pre \Rightarrow Pre0$ is valid.

5.5 Related work

Floyd is among the first to work on program verification using logic methods for program languages (see [98]). Following the Floyd's approach, T.Hoare gives a formal logic for program verification in [53] known today under the name Hoare logic. Dijkstra and Scholten [39] proposes then an efficient way for applying Hoare logic in program verification, in particular they propose two predicate transformer calculus and give their formal semantics.

In the following, we review briefly the existing work related to bytecode verification and more particularly program verification tailored to Java bytecode programs. 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 [95] 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 [22] gives a typed logic for a bytecode language with stacks and jumps. The technique that he proposes checks that both types and specifications are respected. The language is simple and supports basically stack and arithmetic operations. A proof of correctness w.r.t. an operational semantics is given. Differently from this work, here we assume that program are well typed. This is a safe assumption as the JVM is supplied with a bytecode verifier [73]. We consider that the separation of the concerns for well typedness and functional correctness between the bytecode verifier and a verification condition generator is a good design decision.

Following the work of Nick Benton, Bannwart and Muller [11] 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 is well-typed. In particular, they define a Hoare-style bytecode logic which consists in building a derivation tree and the leaves of the derivation tree must be proved in a classical logic. This is different from our solution where we generate directly verification conditions using a weakest precondition calculus which are then proved in a logic. A main inconvenient of using Hoare logic triples for proving program correctness is that this is a complex process and needs even in simple cases a high level of user interaction and competence. Of course, our approach also requires user interaction as far as the generated verification conditions are hard to prove but automation is possible as far as the verification conditions are simple.

In [106], M. Wildmoser and T. Nipkow describe a framework for verifying Jinja (a Java bytecode subset) which features object manipulation, exceptions, method invocations. The verification framework is based on a verification condition generator which uses weakest preconditions. The framework is developped in the interactive theorem prover Isabelle/HOL and proved sound and complete. They show how the safety policy against an arithmetic overflow can be checked. As in our case, they also assume that the program is provided with annotations (e.g. loop invariants).

The Spec# [15] 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 verification procedure [14] includes several processing stages of 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. These transformations of course, facilitate the verification procedure.

```
1  {Pre := reg(1) ≥ 0  }
2  {Pre0  :=        0≥  0  ∧  0≤ reg(1) ∧ 0 == 0 ∗ (0 − 1)/2∧
3  ∀reg(3), reg(2),
4  reg(3) ≥ 0 ∧ reg(3) ≤ reg(1) ∧ reg(2) == reg(3) ∗ (reg(3) − 1)/2
5  ⇒ reg(3) < reg(1) ⇒ Pre5∧
6  reg(3) ≥ reg(1) ⇒ Pre13
7    }
8  0  const  0
9
10 {0 ≥ 0 ∧ 0 ≤ reg(1) ∧ st(cntr) == 0 ∗ (0 − 1)/2
11   ∧
12 ∀reg(3), reg(2)
13 reg(3) >= 0 ∧ reg(3) ≤ reg(1) ∧ reg(2) == reg(3) ∗ (reg(3) − 1)/2
14   ⇒ reg(3) < reg(1) ⇒ Pre5
15       ∧
16   reg(3) >= reg(1) ⇒ Pre13       }
17 1  store  2
18
19 { 0 ≥ 0 ∧ 0 <= reg(1) ∧ reg(2) == 0 ∗ (0 − 1)/2
20   ∧∀reg(3), reg(2)
21     reg(3) ≥ 0 ∧ reg(3) <= reg(1) ∧ reg(2) == reg(3) ∗ (reg(3) − 1)/2
22        ⇒ reg(3) < reg(1) ⇒ Pre5
23        ∧
24         reg(3) ≥ reg(1) ⇒ Pre13    }
25 2  const  0
26
27 { st(cntr) ≥ 0 ∧ st(cntr) <= reg(1) ∧ reg(2) == st(cntr) ∗ (st(cntr) + 1)/2
28  ∧
29  ∀reg(3), reg(2),
30  reg(3) ≥ 0 ∧ reg(3) <= reg(1) ∧ reg(2) == reg(3) ∗ (reg(3) − 1)/2 ⇒
31  reg(3) < reg(1) ⇒ Pre5 ∧ reg(3) ≥ reg(1) ⇒ Pre13    }
32 3  store  3
33
34 {I ∧ ∀reg(2), reg(3)(I ⇒ Pre10)    }
35 4  goto  10
36
37 { Pre5 := reg(3) + 1 ≥ 0 ∧ reg(3) + 1 <= reg(1) ∧ reg(2) + reg(3) == (reg(3) + 1) ∗ (reg(3))/2      }
38 5  load  2
39
40 { reg(3) + 1 ≥ 0 ∧ reg(3) + 1 <= reg(1) ∧ st(cntr) + reg(3) == (reg(3) + 1) ∗ (reg(3))/2      }
41 6  load  3
42
43 { reg(3) + 1 ≥ 0 ∧ reg(3) + 1 <= reg(1)∧
44  st(cntr − 1) + st(cntr) == (reg(3) + 1) ∗ (reg(3))/2    }
45 7  add
46
47 { reg(3) + 1 ≥ 0 ∧ reg(3) + 1 <= reg(1)∧
48  st(cntr) == (reg(3) + 1) ∗ (reg(3))/2    }
49 8  store  2
50
51 { reg(3) + 1 ≥ 0 ∧ reg(3) + 1 <= reg(1)∧
52  reg(2) == (reg(3) + 1) ∗ (reg(3))/2 }
53 9  iinc  3  //LOOP END
54
55 { Pre10 := reg(3) < reg(1) ⇒ Pre5∧
56        reg(3) ≥ reg(1) ⇒ Pre13 }
57 10  load  3  //LOOP ENTRY
58
59 {   st(cntr) < reg(1) ⇒ Pre5∧
60   st(cntr) ≥ reg(1) ⇒ Pre13      }
61 11  load  1
62
63 {   st(cntr − 1) < st(cntr) ⇒ Pre5∧
64   st(cntr − 1) ≥ st(cntr) ⇒ Pre13 }
65 12  if_icmplt  5
66
67 {   Pre13 := st(cntr) == reg(1) ∗ (reg(1) + 1)/2    }
68 13  return
69 {   Post := result == reg(1) ∗ (reg(1) + 1)/2    }
```

Figure 5.2: WEAKEST PRECONDITION PREDICATES FOR THE INSTRUCTIONS OF THE BYTECODE OF METHOD SUM

Transforming the bytecode into an acyclic control flow graph (or simply identifying the loop entries) allows for an easy treatment of loop invariants. As we said earlier, using an intermediate language allows to treat a smaller language and thus the changes in the verification condition generator can be easily applied. Moreover, supporting an intermediate language allows for using the same verification framework for different programming languages. Passification avoids duplication of formulas which can be exponential in the number of the conditional branches in a program [68]. This method consists basically in converting a program into a single assignment form. Despite that in our implementation we also transform the control graph into an acyclic program, we consider that in a mobile code scenario one should limit the number of program transformations for the following reasons. A design of a verification condition generator should be as simple as possible especially when it is tailored to be installed on the client site of a mobile code scenario. Bur a verification framework which relies on several transformation layers can be relatively complex. Second, a verification condition generator must be proved correct. In the case of several transformations this may be not trivial.

Chapter 6

Correctness of the verification condition generator

In the previous chapter, we defined a verification condition generator for a Java bytecode like language. In this section, we will show 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 method's body and its specification for establishing that the method 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 3.8.

In the following, in Section 6.1 we shall formulate the correctness statement of the verification condition generator. In Section 6.2, we describe the steps which must be overtaken in the proof. The second Section 6.3 establishes the relation between syntactic substitution and semantic evaluation. The latter will play a role in the correctness proof of the verification condition generator in Section 6.4.

6.1 Formulation of the correctness statement

In order to define what it means for the verification calculus to be correct we will need to precise what does it mean that a method respects its specification. As we have stated before the intuition behind this notion is that if the method starts execution in a state where its precondition holds then if it terminates execution then its postcondition holds.

We shall concentrate on the correctness of method pre and postconditions. We shall assume that the modified locations are correct.

As we want also to treat recursive calls, we use a technique known in the literature [88] which consists in annotating the execution relation in the operational semantics with levels. A level of execution stands for the maximal call depth in the execution of a method. Thus, the operational semantics of a method invokation in case of normal termination[1] would be :

$$\frac{\begin{array}{l} \mathit{lookUp}(\text{meth.}\mathsf{Name}, \text{meth.}\mathsf{argsType}, \text{meth.}\mathsf{retType}, \mathsf{TypeOf}\ (\mathrm{St}(\mathrm{Cntr} - \text{meth.}\mathsf{nArgs}))) = \mathrm{n} \\ \mathrm{St}(\mathrm{Cntr} - \mathrm{n}.\mathsf{nArgs}) \neq \textbf{null} \\ \mathrm{n} :< \mathrm{H}, 0, [\,], [\mathrm{St}(\mathrm{Cntr} - \mathrm{n}.\mathsf{nArgs}), \ldots, \mathrm{St}(\mathrm{Cntr})], 0 >\Downarrow_{\mathtt{k}}< \mathrm{H}', \mathrm{Res} >^{norm} \\ \mathrm{Cntr}' = \mathrm{Cntr} - \mathtt{m}.\mathsf{nArgs} + 1 \\ \mathrm{St}' = \mathrm{St}(\oplus \mathrm{Cntr}' \rightarrow \mathrm{Res}) \\ \mathrm{Pc}' = \mathrm{Pc} + 1 \end{array}}{\mathtt{m} \vdash \mathtt{invoke}\ \text{meth}:<\mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}> \hookrightarrow_{\mathtt{k + 1}} <\mathrm{H}',\mathrm{Cntr}',\mathrm{St}',\mathrm{Reg},\mathrm{Pc}'>}$$

If a method invocation is at level `k` this means that the maximal call depth of the method is at most `k-1`. If a method invocation is at level `1` this means that the method execution does not

[1] we are not exhaustive here about all the possible rules of the operational semantics for method invokation, but the other cases are similar

perform any method calls. Thus, a method invocation labeled with 0 does not exist. Moreover, if the invocation of m is at level k+1 any method meth called in its body will have a level k. For the other instructions the weight of the maximal call depth is just passed. The rules for the instructions different from method invocation have this general form:

$$\frac{i \neq \texttt{invoke m} \vdash i{:}s_1 \hookrightarrow_{\texttt{k}} s_2 \ \texttt{k}>0}{\texttt{m} \vdash s_1 \hookrightarrow_{\texttt{k}} s_2}$$

A method respects its specification if every terminating execution is such that if in its initial state the method precondition holds then in its final state the method postcondition holds. We also have to define the method correctness at execution at level k which follows.

Definition 6.1.1 (method respects its specification at level k). *For every method* m *with precondition* m.pre*, normal postcondition* m.normalPost *and the exceptional postcondition function* m.excPostSpec*, we say that* m *respects its specification at level* k *if for every two states* s_0 *and* s_1 *such that :*

- $\texttt{m} : s_0 \Downarrow_{\texttt{k}} s_1$
- $s_0 \vDash \texttt{m.pre}$

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

Next, we give a definition for program correctness at level k.

Definition 6.1.2 (program is correct at level k). *A program is correct if for all classes* Class *in the program, every method* m *in* Class *respects its specification at level* k*.*

We generalize the definition for program correctness w.r.t. the level k.

Definition 6.1.3 (program is correct). *A program is correct if for all levels* $\texttt{k} \geq 0$ *the program is correct at level* k*.*

Let us now see informally the correctness condition for the verification calculus. We would like to establish that if the verification conditions generated for all methods in a program are valid then we can conclude that the program is correct. We formulate this as a the theorem:

Theorem 6.1.1 (Verification condition generator is correct). *If the verification conditions for all methods in a program* P *(see subsection 5.3.4) are valid then* P *is correct w.r.t. Def. 6.1.3*

The main purpose of the current chapter is to establish this theorem. Before entering in technical part of the proof we shall give an outline of the steps to be taken for establishing it.

6.2 Proof outline

We give now an informal description of the steps to be overtaken for the proof of Theorem 6.1.1. We will describe our reasoning in the direction opposite to its formalization in the later sections. We start with our main objective and then start to "zoom" in the steps that must be made in order that it be achieved.

Thus, in order to establish our main theorem, we have to see under what conditions a single method is correct at a level k. The greater part of the proof is concentrated on this. Remind that a method correctness at level k means if the precondition calculated by the *wp* for the entry instruction of the method holds in the initial state of the method then the postcondition of the method will hold in the final state of the method provided that the method terminates. This is the statement of Lemma 6.4.5.

For justifying Lemma 6.4.5, we first establish that if the precondition calculated by the *wp* for the entry instruction of the method holds in the initial state of the method then the precondition calculated by *wp* for every instruction reachable from the method entry instruction also holds in

the prestate of that instruction. This is done in Lemma 6.4.4. We use here an induction over the number of execution steps made in the execution path. The induction case uses Lemma 6.4.3.

Lemma 6.4.3 shows that if in an execution path the preconditions calculated by *wp* for the instructions in the path are such that the respective precondition holds in the prestate of the respective instruction then either the respective normal or exceptional method postcondition holds or if another execution step can be made then the weakest precondition of the next instruction holds. This is also known as a subject reduction property. The latter lemma is may be the most complicated one as it uses the argument of the reducibility of the execution graph (Section 3.9, Def. 3.9.1). The lemma has three cases:

- the case where the next instruction is not a loop entry instruction. In this case the proof is standard and uses the single step soundness of *wp*.
- the case when the next instruction is a loop entry instruction and the current instruction is not a loop end. In this case, we use the single step soundness of *wp* as well the special form of the precondition of the current instruction.
- the case where the next instruction is a loop entry and the current is a loop end instruction (see Def. 3.9.1). In this case, we use the reducibility of the control flow graph which gives us that the execution path has a prefix subpath which passes through the loop entry instruction but not through the current loop end instruction. This fact allows us to conclude that also in that case the Lemma 6.4.3 holds

What we mean by single step soundness of the *wp* function is that if the predicate calculated by *wp* for an instruction holds in its prestate then the postcondition upon which the precondition is calculated holds in its poststate (Lemma 6.4.2). The argument for the single step soundness uses the relation between syntactic substitution and semantic evaluation which looks like:

$$[\![E_1[E_2 \backslash E_3]]\!]_{s_1} = [\![E_1]\!]_{s_1(\oplus [\![E_2]\!]_{s_1} \rightarrow [\![E_3]\!]_{s_1})}$$

This equivalence is standardly used for establishing the soundness of predicate transformer function w.r.t. a program semantics and means that it does not matter if we use a syntactic substitution over the expression and evaluate the resulting expression or update the state and evaluate the original expression. The next section is dedicated to the relation between substitution and evaluation.

Note that we do the proof under several assumptions. The first one is that the bytecode has passed the bytecode verification. This guarantees that when the bytecode is executed every instructions gets from the operand stack values of the expected type. This assumption liberates of us from the obligation to make type checks in the verification condition generator. Next, we assume that the control flow graph of a method is reducible, or in other words there are no cycles in the graph with two entries. This means that if program have cycles then they should conform to Def. 3.9.1 from Section 3.9. As we shall in the following, control flow graph reducibility plays an important role in the proof of soundness of the verification condition generator. Note that this restriction (as we said earlier) is realistic as every non-optimizing Java compiler produces reducible control flow graphs and even hand written code is usually reducible.

6.3 Relation between syntactic substitution and semantic evaluation

In this section, we will show what is the relation between the syntactic notion of substitution and the semantic notion of evaluation. Particularly, we shall see that they commute. As an intermediate execution state $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ is composed from several elements, a heap H, the stack counter Cntr , the operand stack St and the array of registers Reg, we shall state for each component a separate lemma. In the following, we shall sketch the proof only of those lemmas, that we consider representative the others having a similar proof.

Let us now look at the next formal statement. It refers to the fact that if we substitute in an expression E_1 the expression **reg**(i) which represents the local variable at index i with another

expression E_2 and evaluate the resulting expression in a state s_1 we will get the same value if we evaluate E_1 in the state $s_1(\oplus \mathrm{Reg}(i) \rightarrow [\![E_2]\!]_{s_1})$.

Lemma 6.3.1 (Update a local variable). *For any expressions E_1, E_2 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$, $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}(\oplus i \rightarrow [\![E_2]\!]_{s_1}), \mathrm{Pc} >$ and i is a valid index in the array of method register* Reg *then the following holds:*

1. $[\![E_1[\mathbf{reg}(i) \backslash E_2]]\!]_{s_1} = [\![E_1]\!]_{s_2}$
2. $s_1 \vDash \psi[\mathbf{reg}(i) \backslash E_2] \iff s_2 \vDash \psi$

Proof

1. we look at the first part of the lemma concerning expression evaluation. It is by structural induction on the structure of E_1. We look only at the simple case when $E_1 = \mathbf{reg}(i)$. The other cases proceed in a similar way.

   $[\![\mathbf{reg}(i)[\mathbf{reg}(i) \backslash E_2]]\!]_{s_1} =$
   $\{$ *apply substitution* $\}$
   $[\![E_2]\!]_{s_1} =$
   $\{$ *evaluation of local variables and by the initial hypothesis for s_2* $\}$
   $[\![\mathbf{reg}(i)]\!]_{s_2}$

2. second case of the lemma. It is by induction on the structure of the formula ψ. We sketch the case when $\psi = E_1 \ \mathcal{R} \ E_2$

   $s_1 \vDash (E_1 \ \mathcal{R} \ E_2)[\mathbf{reg}(i) \backslash E_2] =$
   $\{$ *apply substitution* $\}$
   $s_1 \vDash E_1[\mathbf{reg}(i) \backslash E_2] \ \mathcal{R} \ E_2[\mathbf{reg}(i) \backslash E_2] =$
   $\{$ *interpretation of formulas* $\}$
   $[\![E_1[\mathbf{reg}(i) \backslash E_2]]\!]_{s_1} \ \mathcal{R} \ [\![E_2[\mathbf{reg}(i) \backslash E_2]]\!]_{s_1} =$
   $\{$ *from the first part of the lemma and the initial hypothesis for s_2 we get* $\}$
   $[\![E_1]\!]_{s_2} \ \mathcal{R} \ [\![E_2]\!]_{s_2} =$
   $\{$ *from definition of formula interpretation in a state* $\}$
   $s_2 \vDash E_1 \ \mathcal{R} \ E_2$

Lemma 6.3.2 (Update of the heap). *For any expressions E_1, E_2, E_3 and any field* `f` *if we have that the states s_1 and s_2 are such that the evaluation of E_1 $[\![E_2]\!]_{s_1}$ is different from* **null** *and the evaluation of E_3 is either an instance in the heap or of type* **int**, *$s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$,and $s_2 =< \mathrm{H}(\oplus \mathtt{f} \rightarrow \mathtt{f}(\oplus [\![E_2]\!]_{s_1} \rightarrow [\![E_3]\!]_{s_1})), \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ the following holds*

1. $[\![E_1[\mathtt{f} \backslash \mathtt{f}(\oplus E_2 \rightarrow E_3)]]\!]_{s_1} = [\![E_1]\!]_{s_2}$
2. $s_1 \vDash \psi[\mathtt{f} \backslash \mathtt{f}(\oplus E_2 \rightarrow E_3)] \iff s_2 \vDash \psi$

Proof
We sketch only the first part of the lemma, the second part is by structural induction as in the previous lemma, second case.

1. By structural induction on the structure of E_1. We look at the case when $E_1 = E_2.\mathtt{f}$

   $[\![E_2.\mathtt{f}[\mathtt{f} \backslash \mathtt{f}(\oplus E_2 \rightarrow E_3)]]\!]_{s_1} =$
   $\{$ *apply substitution over fields as described in subsection 2.4.1 page 19* $\}$
   $[\![E_2.\mathtt{f}(\oplus E_2 \rightarrow E_3)]\!]_{s_1} =$
   $\{$ *simplify the expression as in subsection 2.4.1 page 19* $\}$
   $[\![E_3]\!]_{s_1} =$
   $\{$ *evaluation of field access expression and by the initial hypothesis for s_2* $\}$
   $[\![E_2.\mathtt{f}]\!]_{s_2}$

Lemma 6.3.3 (Update of the heap with a newly allocated object). *For any expressions E_1 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}', \mathrm{Cntr}, \mathrm{St}(\oplus\mathrm{Cntr} \rightarrow [\![\mathtt{ref}]\!]_{s_1}), \mathrm{Reg}, \mathrm{Pc} >$ where $\mathsf{newRef}(\mathrm{H}, C) = (\mathrm{H}', \mathtt{ref})$ the following holds*

1. $$[\![E_1[\mathbf{st}(\mathbf{cntr})\backslash\mathtt{ref}][\mathtt{f}\backslash\mathtt{f}(\oplus\mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, C)}_{\mathtt{f}}]\!]_{s_1} = [\![E_1]\!]_{s_2}$$

2. $$s_1 \vDash \psi[\mathbf{st}(\mathbf{cntr})\backslash\mathtt{ref}][\mathtt{f}\backslash\mathtt{f}(\oplus\mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, C)}_{\mathtt{f}} \iff s_2 \vDash \psi$$

Proof
We sketch only the first part of the lemma, the second part is by structural induction as in the first lemma, second case.

1. By structural induction on the structure of E_1. We look at the case when $E_1 = \mathbf{st}(\mathbf{cntr}).\mathtt{g}$ where the field $\mathtt{g}$ is declared in class C.

   $[\![\mathbf{st}(\mathbf{cntr}).\mathtt{g}[\mathbf{st}(\mathbf{cntr})\backslash\mathtt{ref}][\mathtt{f}\backslash\mathtt{f}(\oplus\mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, C)}_{\mathtt{f}}]\!]_{s_1} =$
   $\{$ *applying the first substitution over* $\mathbf{st}(\mathbf{cntr}).\mathtt{g}$ $\}$
   $[\![\mathtt{ref.g}[\mathtt{f}\backslash\mathtt{f}(\oplus\mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, C)}_{\mathtt{f}}]\!]_{s_1} =$
   $\{$ *as by initial hypothesis* $\mathtt{g}$ *is declared in class* C *the series of substitutions over the fields declared in any subtype of* C *results in* $\}$
   $[\![\mathtt{ref.g}(\oplus\mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{g.Type}))]\!]_{s_1} =$
   $\{$ *simplify the field update expression* $\}$
   $[\![\mathsf{defVal}(\mathtt{g.Type})]\!]_{s_1} =$
   $\{$ *by initial hypothesis about the state* s_2 *and definition of the function* $\mathsf{newRef}(\mathrm{H}, C)$ *in section 3.4.1 we get* $\}$
   $[\![\mathbf{st}(\mathbf{cntr}).\mathtt{g}]\!]_{s_2}$

Lemma 6.3.4 (Update the stack). *For any expressions E_1, E_2, E_3 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}(\oplus[\![E_2]\!]_{s_1} \rightarrow [\![E_3]\!]_{s_1}), \mathrm{Reg}, \mathrm{Pc} >$ then the following holds:*

1. $[\![E_1[\mathbf{st}(E_2)\backslash E_3]]\!]_{s_1} = [\![E_1]\!]_{s_2}$
2. $s_1 \vDash \psi[\mathbf{st}(E_2)\backslash E_3] \iff s_2 \vDash \psi$

Lemma 6.3.5 (Update the stack counter). *For any expressions E_1, E_2 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$, $s_2 =< \mathrm{H}, [\![E_2]\!]_{s_1}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ then the following holds:*

1. $[\![E_1[\mathbf{cntr}\backslash E_2]]\!]_{s_1} = [\![E_1]\!]_{s_2}$
2. $s_1 \vDash \psi[\mathbf{cntr}\backslash E_2] \iff s_2 \vDash \psi$

Lemma 6.3.6 (Return value property). *For any expression E_1 and E_2, for any two states s_1 and s_2 such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$,$s_2 =< \mathrm{H}, [\![E_2]\!]_{s_1} >^{norm}$ then the following holds:*

1. $[\![E_1[\mathbf{result}\backslash E_2]]\!]_{s_1} = [\![E_1]\!]_{s_2}$
2. $s_1 \vDash \psi[\mathbf{result}\backslash E_2] \iff s_2 \vDash \psi$

6.4 Proof of Correctness

In the following, we shall consider also that we manipulate meaningful states. A meaningful state means that the heap is well - formed as defined in Definition 3.4.1. Moreover, in a meaningful state local variables and the stack contain only references from the current heap or integer values.

Definition 6.4.1 (Meaningful state). *The state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *is meaningful if the heap respects conditions in Def. 3.4.1 and every local variable* l *(*$l \in \mathrm{Reg}$*)* $l \in \mathrm{H.Loc} \vee l \in$ **int** *Moreover, the stack also must contain well formed values :* $(\forall i, 0 \leq i \leq \mathrm{Cntr} \Rightarrow \mathrm{St}(i) \in \mathrm{H.Loc} \vee \mathrm{St}(i) \in$ **int**$)$

Actually, the operational semantics preserves the meaningful states, i.e. if an instruction starts execution in a meaningful state it terminates execution in such state

Lemma 6.4.1 (Operational semantics preserves meaningful states). *If the state* s *with the following configuration* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *is meaningful and occurs in the execution of method* `m` *then the state* s' *resulting from one execution step, i.e.* $\mathtt{m} \vdash \mathtt{m}[\mathrm{Pc}] : s \hookrightarrow s'$ *is also meaningful.*

Our first concern now will be to establish that the rules for single bytecode instructions have the following property: 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 6.4.2 (Single execution step correctness). *Let us have a program* `P` *and a method* `m` *in* `P`*. For every intermediate state* $s_n =< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >$ *and initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *of the execution of method* `m` *if the following conditions hold:*

- $\mathtt{m.body}[0] : s_0 \hookrightarrow^*_{\mathtt{k}} s_n$
- $\mathtt{m.body}[\mathrm{Pc}_n] : s_n \hookrightarrow_{\mathtt{k}} s_{n+1}$
- $s_n \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$
- *all methods in* `P` *respect their specification at level* `k-1`
- *the verification conditions for* `P` *are valid formulas*

then :

- *if* $\mathtt{m.body}[\mathrm{Pc}_n] =$ `return` *and* $s_{n+1} =< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}$ *then* $s_{n+1} \vDash \mathtt{m.normalPost}$ *holds*
- *if* $\mathtt{m.body}[\mathrm{Pc}_n] \neq$ `athrow` *throws a not handled exception of type* Exc*,* $s_{n+1} =< \mathrm{H}_{n+1}, \mathtt{ref} >^{exc}$ *and* $\mathsf{TypeOf}\ (\mathtt{ref}) = \mathrm{Exc}$ *and* $\mathsf{newRef}(\mathrm{H}_n, \mathrm{Exc}) = (\mathrm{H}_{n+1}, \mathtt{ref})$ *then* $< \mathrm{H}_{n+1}, \mathtt{ref} >^{exc} \vDash \mathtt{m.excPostSpec}(\mathrm{Exc})$ *holds.*
- *if* $\mathtt{m.body}[\mathrm{Pc}_n] =$ `athrow` *throws a not handled exception of type* Exc *and moreover we have that* $s_{n+1} =< \mathrm{H}_n, \mathrm{St}(\mathrm{Cntr}) >^{exc}$ *and* $\mathrm{H.TypeOf}\ (\mathrm{St}(\mathrm{Cntr})) = \mathrm{Exc}$ *then* $< \mathrm{H}_n, \mathrm{St}(\mathrm{Cntr}) >^{exc} \vDash \mathtt{m.excPostSpec}(\mathrm{Exc})$ *holds.*
- *else* $s_{n+1} \vDash\ inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$ *holds*

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

Return instruction By initial hypothesis we have that the *wp* of the current instruction holds

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

From Lemma 6.3.6, which describes a substitution property for the expression **result**, we get:

$$< \mathrm{H}_n, [\![\mathbf{st}(\mathrm{cntr})]\!]_{<\mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n>} >^{norm} \vDash \mathsf{normalPost}$$

By the definition of the evaluation function, we can get that the postcondition normalPost holds in the configuration $< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}$

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

From the operational semantics for `return` (Section 3.8), we have that the state $s_{n+1} =< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}$ which allows us to conclude that this case holds.

Instance creation From the definition of the *wp* instruction for the instruction `new` we obtain that the following holds:

$$\begin{array}{l} < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash \\ \quad \forall \mathtt{ref}, (\neg\, \mathbf{instances}(\mathtt{ref}) \wedge \mathbf{typeof}(\mathtt{ref}) = C \wedge \mathtt{ref} \neq \mathbf{null}) \Rightarrow \\ \qquad\qquad\qquad [\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ \qquad inter(i, i+1, \mathtt{m}) \;\; [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathtt{ref}] \\ \qquad\qquad\qquad [\mathtt{f} \backslash \mathtt{f}(\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, C)}_{\mathtt{f}:\mathbf{Field}} \end{array} \tag{8.1.1.1}$$

Moreover, from the operational semantics of the `new` instruction, we obtain that the state s_{n+1} is of the form

$$s_{n+1} =< \mathrm{H}_{n+1}, \mathrm{Cntr}_n + 1, \mathrm{St}_n(\oplus \mathrm{Cntr}_n + 1 \rightarrow \mathtt{ref}'), \mathrm{Reg}_n, \mathrm{Pc}_n + 1 > \tag{8.1.1.2}$$

where the heap H_{n+1} in state s_{n+1} is obtained from the allocation operator $\mathsf{newRef}(\mathrm{H}, C) = (\mathrm{H}_{n+1}, \mathtt{ref}')$

We can instantiate in (8.1.1.1) with $\mathtt{ref}'$. Moreover, because the $\mathtt{ref}'$ is not **null** not in the heap H_n and is of type C (see Def. 3.4.2) we get that

$$\begin{array}{l} < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash \\ \qquad\qquad\qquad [\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ \qquad inter(i, i+1, \mathtt{m}) \;\; [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathtt{ref}'] \\ \qquad\qquad\qquad [\mathtt{f} \backslash \mathtt{f}(\oplus \mathtt{ref}' \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, C)}_{\mathtt{f}:\mathbf{Field}} \end{array}$$

From lemmas 6.3.5, 6.4 and 6.3.2, 6.3.3 which state substitution properties for the stack counter, the stack and the heap and from (8.1.1.2), we conclude that this case holds

Field update The cases `getfield`, `astore`, `aload`, `arraylength` are similar to this case. This instruction may potentially throw a `NullExc` as the object reference whose field is updated may be **null**. Thus, we should consider three cases : the case when the reference is **null** and the `NullExc` exception is not caught, case when the reference is **null** and the `NullExc` exception is caught, and the case when the reference is not **null**. We consider in the following only the case, when an uncaught exception of type `NullExc` is thrown. By initial hypothesis, we have that the *wp* function holds. In particular, we consider the case when an exception is thrown we get that

$$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostRTE}(i, \mathtt{NullExc}) \tag{8.1.3.1}$$

From the operational semantics of the instruction `putfield` in case of a thrown exception we get that $\mathbf{st}(\mathbf{cntr}) = \mathbf{null}$ and thus from (8.1.3.1) we conclude that

$$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n > \vDash \mathtt{m}.\mathsf{excPostRTE}(i, \mathtt{NullExc})$$

We unfold the function excPostRTE (see its Def. 5.3.2.2) and obtain

$$\begin{array}{l} s_1 \vDash \\ \mathtt{m}.\mathsf{excPostSpec}(\mathtt{NullExc}) \end{array} \begin{array}{l} \forall \mathtt{ref}(\neg\, \mathbf{instances}(\mathtt{ref}) \wedge \mathtt{ref} \neq \mathbf{null} \wedge \mathbf{typeof}(\mathtt{ref}) = \mathtt{NullExc}) \Rightarrow \\ \qquad [\mathbf{cntr} \backslash 0] \\ \qquad [\mathbf{st}(0) \backslash \mathtt{ref}] \\ \qquad [\mathtt{f} \backslash \mathtt{f}(\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f.Type}))]^{\mathsf{subtype}(\mathtt{f.declaredIn}, \mathtt{NullExc})}_{\mathtt{f}} \end{array} \tag{8.1.3.2}$$

We assume that the exception is not caught. Thus from the operational semantics of `putfield` we get that the instruction execution terminates in a terminal exceptional state $< \mathrm{H}', \mathtt{ref}' >^{exc}$

$$s_{n+1} =< \mathrm{H}', \mathtt{ref}' >^{exc}, where\ (\mathrm{H}', \mathtt{ref}') = \mathsf{newRef}(\mathrm{H}, \mathtt{NullExc}) \tag{8.1.3.3}$$

From Def.3.4.2 of the function newRef, we know that the reference `ref'` is not **null**, is not in the heap H_n and moreover it is of type `NullExc`. This allows us to instantiate (8.1.3.2) and obtain

$$\begin{array}{l} s_n \vDash \mathtt{m}.\mathsf{excPostSpec}(\mathtt{NullExc}) \\ \end{array} \begin{array}{l} [\mathbf{cntr}\backslash 0] \\ [\mathbf{st}(0)\backslash \mathtt{ref}'] \\ [\mathtt{f}\backslash \mathtt{f}(\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(\mathtt{f}.\mathsf{Type}))]_{\mathtt{f}}^{\mathsf{subtype}(\mathtt{f}.\mathsf{declaredIn}, \mathtt{NullExc})} \end{array} \tag{8.1.3.4}$$

From substitution lemmas 6.3.5, 6.3.2, 6.4 and 6.3.3 conclude that the result holds in this case.

method invocation In the following, we ignore the part of the predicate calculated by the *wp* predicate transformer which concerns the exceptional termination for reasons of clarity. Treating the whole definition of the *wp* for method invokation is done in a rather similar way as the part that we sketch here. By initial hypothesis, we get that the *wp* of method invocation holds. This in particular means that the precondition `n` .pre of the invoked method `n` holds

$$s_n \vDash \mathtt{n}\ .\mathsf{pre}[\mathbf{reg}(s)\backslash \mathbf{st}(\mathbf{cntr} + s - \mathtt{n}\ .\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \tag{8.1.4.1}$$

Moreover, we get that the postcondition `n` .normalPost of the invoked method implies the intermediate predicate *inter*(i, i+1, `m`) between the current and the next instruction in the execution

$$\begin{array}{l} \forall res, m(m \in \mathtt{n}.\mathsf{modif}), \\ s_n \vDash \quad \mathtt{n}\ .\mathsf{normalPost}[\mathbf{result}\backslash res][\mathbf{reg}(s)\backslash \mathbf{st}(\mathbf{cntr} + s - \mathtt{n}\ .\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \Rightarrow \\ \quad inter(i, i+1, \mathtt{m})[\mathbf{cntr}\backslash \mathbf{cntr} - \mathtt{n}\ .\mathsf{nArgs}][\mathbf{st}(\mathbf{cntr} - \mathtt{n}\ .\mathsf{nArgs})\backslash res] \end{array} \tag{8.1.4.2}$$

As the verification conditions for the program `P` are valid, we can get that the verification conditions that `n'` is a behavioral subtype the overriden method `n` and we get that the first two conditions from Def. 5.3.2 (page 76) hold which concern the behavioral subtyping of `n'` and `n`. From this fact and that we consider only meaningful states and from the validity of (8.1.4.2) we deduce that the precondition of `n'` holds in state s_n

$$s_n \vDash \mathtt{n}'.\mathsf{pre}[\mathbf{reg}(i)\backslash \mathbf{st}(\mathbf{cntr} + s - \mathtt{n}\ .\mathsf{nArgs})]_{i=0}^{\mathtt{n}.\mathsf{nArgs}} \tag{8.1.4.3}$$

From the operational semantics of method invocation the method lookup function *lookUp* will find the method `n'` which overrides method `n` and which will be actually executed. The method `n'` will be executed in initial state s' described below and ends in the state $< \mathrm{H}', \mathrm{Res} >^{norm}$ Because `n'` respects its specification level at `k-1` we can apply the initial hypothesis for `n'` and the concrete initial and final states and we then get

$$\begin{array}{l} s' \vDash \mathtt{n}'.\mathsf{pre} \Rightarrow < \mathrm{H}', \mathrm{Res} >^{norm} \vDash \mathtt{n}'.\mathsf{normalPost} \\ \\ where \\ s' =< \mathrm{H}_n, 0, [\,], [\mathrm{St}_n(\mathrm{Cntr}_n - \mathtt{n}'.\mathsf{nArgs}), \ldots, \mathrm{St}_n(\mathrm{Cntr}_n)], 0 > \\ \mathrm{Cntr}_{n+1} = \mathrm{Cntr}_n - \mathtt{n}.\mathsf{nArgs} + 1 \\ \mathrm{St}_{n+1} = \mathrm{St}_n(\oplus \mathrm{Cntr}_n \rightarrow \mathrm{Res}) \\ \mathrm{Pc}_{n+1} = \mathrm{Pc}_n + 1 \\ \mathrm{H}_{n+1} = \mathrm{H}' \end{array} \tag{8.1.4.4}$$

From the fact that method $\mathtt{n'}$ respects its specification at level k, (8.1.4.3) and (8.1.4.4) we get that the postcondition of $\mathtt{n'}$ holds in the terminal state $< \mathrm{H'}, \mathrm{Res} >^{norm}$ w.r.t. the initial state s' of the execution of method $\mathtt{n'}$

$$< \mathrm{H'}, \mathrm{Res} >^{norm} \vDash \mathtt{n'}.\mathsf{normalPost} \tag{8.1.4.5}$$

Because the postcondition $\mathtt{n'}.\mathsf{normalPost}$ does not mention the operand stack, neither the stack counter and because of the relation between s_{n+1} and $< \mathrm{H'}, \mathrm{Res} >^{norm}$ we can conclude from (8.1.4.5) that the following holds

$$s_n \vDash \mathtt{n'}.\mathsf{normalPost}[\mathbf{reg}(s) \backslash \mathbf{st}(\mathbf{cntr} + s - \mathtt{n}\ .\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}}[\mathbf{result} \backslash \mathrm{Res}]$$

This and the fact that the states operational semantics preserves meaningful states (Lemma 6.4.1), we can apply the verification condition concerning the fact that the method $\mathtt{n'}$ is a behavioral subtype of n, the postcondition of the former is stronger than latter

$$s_n \vDash \mathtt{n}.\mathsf{normalPost}[\mathbf{reg}(s) \backslash \mathbf{st}(\mathbf{cntr} + s - \mathtt{n}\ .\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}}[\mathbf{result} \backslash \mathrm{Res}]$$

We can apply the resulting judgment with modus ponens to (8.1.4.4) where we initialize the quantification over the modified locations with their values in state s_n and the variable *res* to Res

$$s_n \vDash \ \mathit{inter}(i, i+1, \mathtt{m})[\mathbf{cntr} \backslash \mathbf{cntr} - \mathtt{n}\ .\mathsf{nArgs}][\mathbf{st}(\mathbf{cntr} - \mathtt{n}\ .\mathsf{nArgs}) \backslash \mathrm{Res}]$$

From Lemma 6.3.5 for substitution of the stack counter as well as the stack and by the operational semantics of the instruction `invoke` in case of normal termination we conclude that this case holds.

Qed.

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 6.4.3. *Let us have a program* P *and a method* m *in* P*. For every two state configurations* $< \mathrm{H}_0, \mathrm{Cntr}_0, \mathrm{St}_0, \mathrm{Reg}_0, \mathrm{Pc}_0 >$ *and* $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >$ *denoted respectively with* s_0 *and* s_n*, such that :*

- $\mathtt{m}.\mathsf{body}[0] : s_0 \hookrightarrow_{\mathtt{k}}^{n} s_n$
- $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_n] : s_n \hookrightarrow_{\mathtt{k}} s_{n+1}$
- $\forall i, (\ 0 \leq i \leq n),\ s_i \vDash \mathit{wp}(\ \mathrm{Pc}_i, \mathtt{m})$
- *all methods in* P *respect their specification at level* k-1
- *the verification conditions for* P *are valid formulas*

then the following holds:

1. *if* $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_n] = \mathtt{return}$ *and* $s_{n+1} =< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}$ *then* $s_{n+1} \vDash \mathtt{m}.\mathsf{normalPost}$ *holds.*
2. *if* $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_n] \neq \mathtt{athrow}$ *throws a not handled exception of type* Exc*,* $s_{n+1} =< \mathrm{H}_{n+1}, \mathtt{ref} >^{exc}$*,* $\mathrm{H}.\mathsf{TypeOf}\ (\mathtt{ref}) = \mathrm{Exc}$ *and* $\mathsf{newRef}(\mathrm{H}_n, \mathrm{Exc}) = (\mathrm{H}_{n+1}, \mathtt{ref})$ *then the following holds* $< \mathrm{H}_{n+1}, \mathtt{ref} >^{exc} \vDash \mathtt{m}.\mathsf{excPostSpec}(\mathrm{Exc})$.
3. *if* $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_n] = \mathtt{athrow}$ *throws a not handled exception of type* Exc *and we have moreover that* $s_{n+1} =< \mathrm{H}_n, \mathrm{St}(\mathrm{Cntr}) >^{exc}$ *and* $\mathrm{H}.\mathsf{TypeOf}\ (\mathrm{St}(\mathrm{Cntr})) = \mathrm{Exc}$ *then* $< \mathrm{H}_n, \mathrm{St}(\mathrm{Cntr}) >^{exc} \vDash \mathtt{m}.\mathsf{excPostSpec}(\mathrm{Exc})$ *holds.*

4. else $s_{n+1} \vDash wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$ *holds*

Proof : The proof is by case analysis on the execution relation $\longrightarrow$ between the current instruction and the next instruction. We have three cases: the case when the next execution step does not enter a cycle (the next instruction is not a loop entry in the sense of Def.3.9.1) 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 $\longrightarrow_l$) 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)

Case 1 the next instruction to be executed is not a loop entry instruction. Following Def. 5.3.1.1 of the function *inter* in this case we get $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m}) = wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$. By initial hypothesis, we have moreover that the weakest predicate for Pc_n is valid in state s_n, i.e. $s_n \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$. Thus by the previous lemma 6.4.2 we know that the predicate $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$ holds in s_{n+1}, i.e. $s_{n+1} \vDash inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$. This actually means that the judgment $s_{n+1} \vDash wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$ holds.

Case 2 Pc_n is not a loop end and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of bytecode instructions of the method $\mathtt{m}$. 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 $\mathtt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$.

Because initial hypothesis we have that $s_n \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$, from Lemma 6.4.2 we have that the intermediate predicate $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$ holds in state s_{n+1}, i.e. the following holds $s_{n+1} \vDash inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$. From the Def. 5.3.1.1 of the predicate *inter*, we get that in the case for an edge between a loop entry and a loop end $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$ is of the form:

$$s_{n+1} \vDash I \ \wedge \forall mod_i, i = 1..s\,(I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}))$$

We can get from the last formulation and the semantics of the universal quantification that

$$\begin{array}{l} s_{n+1} \vDash I \\ s_{n+1} \vDash I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}) \end{array}$$

By modus ponens, this allows to conclude that $wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$ holds in state s_{n+1}

Case 3 Instruction Pc_n is an end of a cycle and the next instruction to be executed is a loop entry instruction at index *loopEntry*, i.e. $loopEntry = \mathrm{Pc}_{n+1}$ (i.e. the execution step is of kind $\longrightarrow^l$). 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 consider the case when the current instruction is a sequential instruction. The cases when the current instruction is a jump instruction are similar. By initial hypothesis we have that $s_n \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$. From Def. 5.3.1.1 for the case when the current instruction is a loop end and the next instruction is a loop entry, we get that the initial hypothesis is equivalent to

$$s_{n+1} \vDash I \tag{8.3.1}$$

From def. 3.9.1, we conclude that there is a prefix $subP = \mathtt{m.body}[0] \longrightarrow^* loopEntry$ of the current execution path which does not pass through Pc_n. We can conclude that the transition between *loopEntry* and its predecessor k (which is at index k in $\mathtt{m.body}$) in the path $subP$ is not a backedge. By hypothesis we know that $\forall i, 0 \leq i \leq n, s_i \vDash wp(\ \mathrm{Pc}_i, \mathtt{m})$. From def.5.3.1.1 and lemma 6.4.2 we conclude

$$\exists k, 0 \leq k \leq n \Rightarrow s_k \vDash \ \ I \wedge \forall mod_i, i = 1..s\,(\ \ I \Rightarrow wp(\ loopEntry, \mathtt{m})\ \ ) \tag{8.3.2}$$

It also follows that the states s_k and s_{n+1} are the same except for the locations in the modifies list modif of the loop $s_k =^{\mathsf{modif}} s_{n+1}$ because $\mathtt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$ We obtain from (8.3.2) by instantiating every location mod_i in by its value in s_{n+1}

$$s_{n+1} \vDash I \Rightarrow wp(\ loopEntry, \mathtt{m}) \tag{8.3.3}$$

From (8.3.1) and (8.3.3) and because $loopEntry = \mathrm{Pc}_{n+1}$ we conclude that

$$s_{n+1} \vDash wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$$

Qed.

We now show that starting execution of a method in a state where the *wp* predicate for the entry instruction holds implies that the *wp* for all the instructions in the execution path hold.

Lemma 6.4.4 (*wp* precondition for method entry point holds initially). *Let us have a program* `P` *and a method* `m` *in* `P`*. Let us have states* s_0 *and* s_n *such that:*

- *execution of method* `m` *starts in state* s_0
- *makes* n *steps to reach the intermediate state* s_n*:* $s_0 \hookrightarrow_{\mathtt{k}}^{n} s_n$
- $s_0 \vDash wp(\ 0, \mathtt{m})$ *holds*
- *all methods in* `P` *respect their specification at level* `k-1`
- *the verification conditions for* `P` *are valid formulas*

then the following holds

$$\forall i, 0 < i \leq n,\ s_i \vDash wp(\ \mathrm{Pc}_i, \mathtt{m})$$

Proof : Induction over the number of execution steps n

1. $s_0 \hookrightarrow_{\mathtt{k}} s_1$*. By initial hypothesis we have that* $s_0 \vDash wp(\ 0, \mathtt{m})$ *we can apply lemma 6.4.3, we get that* $s_1 \vDash wp(\ \mathrm{Pc}_1, \mathtt{m})$ *and thus, the case when one step is made from the initial state* s_0 *holds*
2. *Induction step:* $s_0 \hookrightarrow_{\mathtt{k}}^{n-1} s_{n-1}$ *and*
   $\forall i, 0 < i \leq n-1,\ s_i \vDash wp(\ \mathrm{Pc}_i, \mathtt{m})$ *and there can be made one step* $s_{n-1} \hookrightarrow_{\mathtt{k}} s_n$*. Lemma 6.4.3 can be applied and we get that (1)* $s_n \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$*. From the induction hypothesis and (1) follows that*
   $$\forall i, 0 < i \leq n,\ s_i \vDash wp(\ \mathrm{Pc}_i, \mathtt{m})$$

Qed.

Having the last lemma we can establish that if a method starts execution in a state in which the *wp* precondition for the method entry instruction holds and the method terminates then the method postcondition holds in the method final state.

Lemma 6.4.5 (*wp* precondition for method entry point holds initially). *Let us have a program* `P` *and a method* `m` *in program* `P`*. For all states* s_0 *and* s_n *such that* $\mathtt{m} : s_0 \Downarrow_{\mathtt{k}} s_n$ *and let* $s_0 \vDash wp(0\ , \mathtt{m})$ *holds. Assume that all methods in* `P` *respect their specification at level* `k-1` *and the verification condition for* `P` *are valid formulas. Let the program counter in state* s_n *points to an instruction* `return` *or an instruction which throws an unhandled exception of type* `Exc`*, then the following holds:*

- *if* $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_{n-1}] = \mathtt{return}$ *then* $s_n \vDash \mathtt{m}.\mathsf{normalPost}$
- *if* $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_{n-1}]$ *throws a not handled exception of type* Exc *then* $s_n \vDash \mathtt{m}.\mathsf{excPostSpec}(\mathrm{Exc})$ *holds.*

Proof: Let $s_0 \hookrightarrow_{\mathtt{k}}^{*} s_n$ *and* $\mathtt{m}.\mathsf{body}[\mathrm{Pc}_n]$ *is a* `return` *or an instruction that throws a not handled exception. Applying lemma 6.4.4, we can get that* $\forall i, 0 \leq i < n,\ s_i \vDash wp(\ \mathrm{Pc}_i, \mathtt{m})$*. We apply lemma 6.4.3 for the case for a* `return` *or instruction that throws an unhandled exception which allows to conclude that the current statement holds.*
Qed.

Now, we establish under what conditions we can conclude that a method respects its specification at level `k`.

Lemma 6.4.6 (method respects its specification at level k). *Let us have a program* P *and a method* m *in* P*. If all methods in* P *respect their specification at level* k -1 *and the verification conditions for* P *are valid formulas then* m *respects its specification at level* k

The proof follows directly from Lemma 6.4.5. We are now ready to prove Theorem 6.1.1.

Theorem 6.1.1. *If the verification conditions of program* P *(see subsection 5.3.4) are valid then* P *is correct w.r.t. Def. 6.1.3*

Proof: We want to establish now that if for every m in every class Class in a program P the respective verification conditions are valid then the P is correct as defined in Def. 6.1.3. This, in particular, means that for all levels $k \geq 0$ the program is correct in level k. We reason by induction on the level k.

base case k = 0 In this case, because we can not have an execution step at level 0, we conclude that all methods at level 0 are correct

induction step Let us have that the program is correct at level k-1. In particular, this means that for every class Class, every method m in Class is correct at level k-1. We can apply Lemma 6.4.6 to every single method m in P, the induction hypothesis and the initial hypothesis that verification conditions of m are valid and we obtain that every method m is correct w.r.t. its specification at level k. This means that the program is correct at level k.

From the base case and the inductive case, we can conclude that the program P is correct for every level k and thus, we have proved the statement.

Qed.

6.5 Related work

Proving the soundness of a verification framework is important as such a proof can guarantee the reliability of such framework. That is why most of the programming logic or verification conditions are also supplied with such a proof. Being a verification condition generator, a Hoare logic tailored to a structured or unstructured language, its proof of soundness requires a formalization of the operational semantics w.r.t. the proof of soundness is done.

Let us make a review of the verification programming logics tailored to source programs and provided with a proof of correctness. The proof presented here has been inspired by the proof of soundness presented by Nipkow in [88] of a Hoare logic tailored to structures language which supports as while loops, exceptions, expression with side effects, recursive procedures, procedure calls. Moreover, the author gives a completeness proof as well as a proof of total correctness. The proof has been done in the theorem prover Isabelle/HOL. The modeling of the assertion language uses a shallow embedding, i.e. the assertions are functions from program states to the propositions of the theorem prover Isabelle/HOL. The same induction over the method depth call underlines the proof.

The Loop tool designed to support JML and tailored to the PVS theorem prover provides also a proof of soundness of the weakest precondition calculus which underlines its verification scheme [61]. The Jive system which is an interactive program theorem prover tailored to a Java subset is also proved correct [79].

In [11], Bannwart and Muller define a Hoare style logic for a Java-like bytecode and prove its correctness on paper using a similar proof technique as used here.

The bytecode logic presented M. Wildmoser and T. Nipkow [106] is also supplied with a proof of soundness done in the theorem prover Isabelle/HOL. There, the soundness result is expressed w.r.t. to a safety policy. A safety policy is expressed not only in terms of method pre and postconditions but but also as annotations at intermediate program points. In particular, the authors prove the soundness of their verification condition generator with respect to the policy for no arithmetic overflow. This means that the policy is expressed also as assertions which accompany instructions and which express that the instruction will not throw an arithmetic exception. Their soundness result states that if the verification conditions are provable then the assertions which express the safety policy holds at the particular program states. We have decided not prove the soundness of

intermediate assertions in order to keep things simple, although we consider this will not present a major problem.

A verification framework whose objectives is to provide a full automation may sacrifice completeness or even soundness. This can be especially the case of verification systems for source code verification where user will be not willing to spend hours on the verification process. For instance, ESC/java [72] is based on intentional trade-offs of unsoundness with other properties of the checker, such as frequency of false alarms (incompleteness) and efficiency. Such unsoundness is introduced from different sources: unsound loop checking, unsound verification of behavioral subtyping, etc. As far as the tool gives adequate results to the user and allows him to discover many other bugs in the program, giving up soundness is a good price to pay. However, for bytecode logic this is not desirable. A potential application of a bytecode logic are mobile code scenarios where the client system may rely on such a logic and thus, the soundness of the logic is necessary.

Moreover, we consider that a bytecode logic must be designed directly over the bytecode. In particular, applying transformations over the bytecode programs may turn the proof of soundness in a difficult task. The proof for soundness presented here would seem large w.r.t. the proof of soundness of Spec# presented in [14]. Spec# relies on several transformations over the bytecode. First, a transformation of the potentially irreducible control flow graph to a reducible one is done. The second step is converting the reducible control flow graph into an acyclic graph by removing the loop back-edges. The third step consists in translating programs into a single-assignment form. The fourth step is converting the program into a guarded command language and finally, the fifth and last step is passifying the guarded command language program which means changing assignments to `assume` statements. The proof presented in [14] is done for programs written in passified guarded command language. But what is the formal guarantee that the initial bytecode program is also correct? A proof of soundness for Spec# which takes into account all the transformation stages can be already complex.

Chapter 7

Equivalence between Java source and bytecode proof obligations

In this chapter, we will look at the relationship between the verification conditions generated for a Java like source programming language and the verification conditions generated for the bytecode language as defined in Chapter 5. More particularly, we argue that they are syntactically equivalent if the compiler is nonoptmizing and satisfies certain conditions.

As we have already discussed in the introductory part of the thesis, the traditional PCC framework and the certifying compiler are limited to decidable properties like well-typedness of the code or safe access to the memory. This is due to the fact that traditional PCC relies on a complete automation.

The relation of the verification conditions over bytecode and source code can be used for building an alternative PCC architecture which can deal with complex security policies. Note that in case of a non-trivial security policy neither an automatic inference of the specification, nor an automatic generation of the proof will be possible. In those cases, such an equivalence can be exploited by the code producer to generate the certificate interactively over the source code. Because of the equivalence between the proof obligations over source and bytecode programs (modulo names and types), their proof certificates are also the same and thus, the certificate generated interactively over the source code can be sent along with the code.

In the following, Section 7.1 presents a simple non optimizing compiler from the source language presented in Chapter 2, Section 2.1 to the bytecode language presented in Chapter 3, Section 3.8. In the section, we will also discuss certain properties of the compiler that we define which are conditions sufficient for establishing the equivalence between the verification conditions over source programs produced by the calculus presented in Chapter 2, Section 2.4 and the verification conditions over the source compilation into bytecode produced by the calculus presented in Chapter 5. Finally, Section 7.3 presents an overview of existing work in the field.

7.1 Compiler

We now turn to specify a simple compiler from the source language presented in the previous section into the bytecode language discussed in Chapter 3. The function which transforms source constructs in the body of method $\mathtt{m}$ into bytecode instructions is denoted with $\ulcorner\urcorner_{\mathtt{m}}$ and its signature is:

$$\ulcorner\urcorner_{\mathtt{m}} : nat * (S \cup E) \longrightarrow list \;\; \mathrm{I} * nat$$

The compiler function takes two arguments: a natural number s from which the labeling of the compilation starts, the statement S or expression E to be compiled and returns a pair which contains a list of bytecode instructions which is the compilation of the source construct and a natural number which is the greatest label in the compilation. In the following, we will allow us to use instead the notation $\ulcorner s, S, e\urcorner_{\mathtt{m}} = [i_s \ldots i_e]$ to refer to the list of instructions which is the first component in the pair resulting from the compilation of statement S starting with index s $\ulcorner s, S\urcorner_{\mathtt{m}} = ([i_s \ldots i_e], e)$.

Although the compiler is realistic, we do not compile variable, class and method names but rather use the same names on bytecode and source code. Taking into account these facts is not difficult but we have made this choice for the sake of simplicity.

The produced bytecode targets a stack based virtual machine[1]. Thus, a basic principle of the compiler is that expressions are compiled into a sequence of instructions whose execution (if it terminates normally) will leave the virtual machine in a state where the value of the expression is on the top of the operand stack.

Of course, the compiler must preserve the intended semantics of source control structures. For instance, the compilation of a compositional statement $S_1; S_2$ should be such that the bytecode resulting from the compilation of statement S_2 is always executed after the compilation of statement S_1 if the latter terminates normally. For cases like compositional, loop and conditional statements, the compiler will use only bytecode instructions in order to preserve their intended meaning. However, this is more complicated for try catch statements. In particular, for every method $\mathtt{m}$ the virtual machine relies on the method exception handler table $\mathtt{m.excHndlS}$(see Section 3.2) and on a search function in it *findExcHandler*(see Section 3.5) in order to decide where to transfer the control in case of a thrown exception. Thus, in order to generate method exception handler table, we define a function addExcHandler which adds elements in $\mathtt{m.excHndlS}$ whenever the compiler function $\ulcorner\urcorner_{\mathtt{m}}$ reaches a try catch statement. The function has the following signature:

$$\mathsf{addExcHandler} : \mathbf{Method} * nat * nat * nat * \mathbf{Class}_{exc}$$

The meaning of the new element is that every exception of type Exc thrown in between the instructions $start \ldots end$ can be handled by the code starting at index h. The formal definition of the procedure is the following:

$$\begin{array}{l}\mathsf{addExcHandler}(\mathtt{m},\ start,\ end,\ h,\ \mathtt{Exc}) = \\ \quad \mathtt{m.excHndlS} := \{(start, end, h, \mathtt{Exc}), \mathtt{m.excHndlS}\}\end{array}$$

We can remark that when the procedure addExcHandler adds a new element in the exception handler table the new element is added at the beginning of the exception handler table $\mathtt{m.excHndlS}$. This means that the more inner is a try catch statement over a statement the smaller index it will have in the exception handler table. This is an important detail which explains why the virtual machine transfers control to the proper exception handler table. This is because in our formalization of the virtual machine, the function *findExcHandler*(see Section 3.5, page 33) returns the first such index in the exception handler table excHndlS starting the search from its beginning.

What we have been describing up to now corresponds to the design of a standard Java compiler. However, for the aims of the current section, we have to take into account the specification of source statements. In particular, we have seen in the chapter 2 where the source language was presented, that the syntax for loop statements explicitly mentions the loop invariant and the modified locations in a loop iteration. In order to preserve the loop specification in the bytecode, we provide the compiler with a function addLoopSpec which adds loop specification elements in the loop specification table (the latter has been introduced in Chapter 5.1 page 62, Section 5.2 page 64). The function has the following signature:

$$\mathsf{addLoopSpec} : \mathbf{Method} * nat * P * list\ E$$

It takes as arguments a method $\mathtt{m}$, an index in the array of bytecode instructions i of $\mathtt{m}$, a predicate $\mathtt{INV}$, a list of modified expressions $\mathtt{modif}$ and adds a new element in the table of loop invariants $\mathtt{m.loopSpecS}$ of method $\mathtt{m}$. Or more formally :

$$\begin{array}{l}\mathsf{addLoopSpec}(\mathtt{m},\ i,\ \mathtt{INV},\ \mathtt{modif}) = \\ \quad \mathtt{m.loopSpecS} := \{(i, \mathtt{INV}, \mathtt{modif}), \mathtt{m.loopSpecS}\}\end{array}$$

In the next subsection 7.1.1, we proceed with the definition of the compiler function $\ulcorner\urcorner_{\mathtt{m}}$ for expressions and statements. In subsection 7.1.2, we discuss the properties of the bytecode produced by the compiler.

[1] We have already described the the JVM in Chapter 3 which is stack based

7.1.1 Compiling source program constructs in bytecode instructions

The compiler is defined inductively over the structure of the source constructs. In the following, we shall focus only on the compiler cases which we consider interesting or particular and we will assume that the rest of the cases are evident from the compiler definition.

Fig. 7.1 shows the compilation scheme for expressions. Let us look in more detail the compilation of the instance creation expression **new** `Class`(E). This case is not trivial because of the way the new instance is initialized. The first instruction in the compilation is the instruction s : `new Class` which creates a new reference in the heap and pushes it on the stack top (see for the operational semantics of the instruction in Section 3.8). Once the new instance is created the instructor must be invoked in order to initialize its (instance) fields. This is done by first duplicating the reference (instruction `dup`) and then invoking the constructor `constr(Class)` of `Class` by passing it as argument the duplicated reference. This has as effect that after the invokation of the constructor the stack top contains the reference to the newly created instance which is now initialized. Note that this compilation follows closely the JVM specification (see [75, §7.8]) which mandates how standard Java compilers must compile instance creation expressions.

$$
\begin{array}{lcl}
\ulcorner s, \mathit{IntConst}, s \urcorner_{\mathtt{m}} & = & s : \mathtt{push}\ \mathit{IntConst} \\
\ulcorner s, \mathbf{null}, s \urcorner_{\mathtt{m}} & = & s : \mathtt{push}\ \mathbf{null} \\
\ulcorner s, \mathbf{this}, s \urcorner_{\mathtt{m}} & = & s : \mathtt{load}\ \mathbf{this} \\
\ulcorner s, E.f, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{getfield}\ f \\
\ulcorner s, E.m(), e \urcorner_{\mathtt{m}} & = & \ulcorner s, E, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{invoke}\ m \\
\ulcorner s, \mathit{Var}, s \urcorner_{\mathtt{m}} & = & s : \mathtt{load}\ \mathit{Var} \\
\ulcorner s, E_1\ \mathit{op}\ E_2, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E_1, e' \urcorner_{\mathtt{m}}; \\
 & & \ulcorner e'+1, E_2, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{arith\_op} \\
\ulcorner s, (\mathtt{Class})\ E, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{checkcast\ Class}; \\
\ulcorner s, \mathbf{new}\ \mathtt{Class}(), s+2 \urcorner_{\mathtt{m}} & = & s : \mathtt{new\ Class}; \\
 & & s+1 : \mathtt{dup}; \\
 & & s+2 : \mathtt{invoke\ constr(Class)};
\end{array}
$$

Figure 7.1: Definition of the compiler for expressions

Fig. 7.2 presents the definition of the compiler function for statements which does not affect the exception handler table neither affect the specification tables. Let us explain in more detail the rule for conditional statement. First the conditional expression is compiled. Its compilation comprises instructions from index s to $e''+1$ where the latter is the conditional branch instruction $e''+1$: `if_`*cond* $e'''+2$. Remind that the `if_`*cond* instruction will compare the two stack top elements (w.r.t. some condition) and if they fulfill the condition in question, the control will be transfered to instruction at index $e'''+2$, otherwise the next instruction at index $e''+2$ will be executed. Note that at index $e'''+2$ starts the compilation of the then branch $\ulcorner e''+2, S_1, e \urcorner_{\mathtt{m}}$ and at index $e''+2$ starts the compilation of the else branch $\ulcorner e''+2, S_2), e''' \urcorner_{\mathtt{m}}$. After the compilation of the else branch follows a $e''+1$: `goto` $e+1$ instruction which jumps at the first instruction outside the compilation of the branch statement.

Fig. 7.3 shows the compiler definition for statements whose compilation will change the the exception handler of the current method. The first such statement is the try catch statement.

$$
\begin{array}{lcl}
\ulcorner s, S_1; S_2, e \urcorner_{\mathtt{m}} & = & \ulcorner s, S_1, e' \urcorner_{\mathtt{m}} \\
 & & \ulcorner e'+1, S_2, e \urcorner_{\mathtt{m}} \\
 & & \\
\ulcorner s, \mathtt{if}\ (E_1\, cond\, E_2)\ \mathtt{then}\ \{S_1\}\ \mathtt{else}\ \{S_2\}, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E_1, e' \urcorner_{\mathtt{m}}; \\
 & & \ulcorner e'+1, E_2, e'' \urcorner_{\mathtt{m}}; \\
 & & e''+1 : \mathtt{if\_}cond\ e'''+2; \\
 & & \ulcorner e''+2, S_2, e''' \urcorner_{\mathtt{m}} \\
 & & e'''+1 : \mathtt{goto}\ e; \\
 & & \ulcorner e'''+2, S_1, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{nop} \\
 & & \\
\ulcorner s, Var = E, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E, e-1 \urcorner_{\mathtt{m}} \\
 & & e : \mathtt{store}\ Var \\
 & & \\
\ulcorner s, E_1.f = E_2, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E_1, e' \urcorner_{\mathtt{m}}; \\
 & & \ulcorner e'+1, E_2, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{putfield\ f} \\
 & & \\
\ulcorner s, \mathtt{athrow}\ E, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{athrow}; \\
 & & \\
\ulcorner s, \mathtt{return}\ E, e \urcorner_{\mathtt{m}} & = & \ulcorner s, E, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \mathtt{return}
\end{array}
$$

Figure 7.2: Definition of the compiler for statements

The compiler compiles the normal statement S_1 and the exception handler S_2. Note that in the exception handler table of the bytecode representation of method `m`, a new line is added which states that the if one of bytecode instructions from index s to index e' throw a not handled exception of type `ExcType` control will be transfered to the instruction at index `ExcType`.

Fig 7.3 contains also the compilation scheme for a try finally statement. Remind that the semantics of the statement is that however the try statement terminates execution the finally statement must be executed after it and the whole statement terminates execution as the try terminated execution. Thus, the compilation of statement S_1 is followed by the compilation of statement S_2. This should assure that after the normal execution of S_1 S_2 will be executed. However, we have also to guarantee that S_2 will execute after S_1 if the latter terminates on exception. For this, we create an exception handler which protects S_1 from any exception thrown. The exception handler stores the reference to the thrown object in variable l then executes the compiled finally statement S_2, then loads the exception reference on the stack and re-throws it.

This compilation differs from the compilation scheme in the JVM specification for finally statements, which requires that the subroutines must be compiled using `jsr` and `ret` instructions. However, the semantics of the programs produced by the compiler presented here and a compiler which follows closely the JVM specification is equivalent. In the following, we discuss informally why this is true. The semantics of a `jsr` k instruction is to jump to the first instruction of the compiled subroutine which starts at index k and pushes on the operand stack the index of the next instruction of the `jsr` that caused the execution of the subroutine. The first instruction of the compilation of the subroutine stores the stack top element in the local variable at index k (i.e. stores in the local variable at index k the index of the instruction following the `jsr` instruction). Thus, after the code of the subroutine is executed, the `ret` k instruction jumps to the instruction following the corresponding `jsr`. This behavior can be actually simulated by programs without `jsr` and `ret` but which inline the subroutine code at the places where a `jsr` to the subroutine is done. We assume that the local variable l is not used in the compilation of the statement S_2, which guarantees that after any execution which terminates normally of $\ulcorner e''+3, S_2, e-2 \urcorner_{\mathtt{m}}$ the local variable l will still hold the thrown object. We also assume that the statement S_1 does not contain a `return` instruction.

We have put separately in Fig.7.4 the compiler definition for loop statements as its compilation is particular because it is the unique case where the specification tables of the current method

$$
\begin{array}{l}
\ulcorner s, \texttt{try}\ \{S_1\}\ \texttt{catch}\ (\texttt{ExcType}\ \mathit{Var})\{S_2\}, e\urcorner_{\texttt{m}} = \\
\quad \ulcorner s, S_1, e'\urcorner_{\texttt{m}}; \\
\quad e' + 1 : \texttt{goto}\ e; \\
\quad \ulcorner e' + 2, S_2, e - 1\urcorner_{\texttt{m}}; \\
\quad e : \texttt{nop} \\
\quad \mathsf{addExcHandler}(\texttt{m},\ s,\ e',\ e' + 2,\ \texttt{ExcType}) \\
\\
\ulcorner s, \texttt{try}\ \{S_1\}\ \texttt{finally}\ \{S_2\}, e\urcorner_{\texttt{m}} = \\
\quad \ulcorner s, S_1, e'\urcorner_{\texttt{m}}; \\
\quad \ulcorner e' + 1, S_2, e''\urcorner_{\texttt{m}}; \\
\quad e'' + 1 : \texttt{goto}\ e; \\
\\
\quad \{\text{ default exception handler}\} \\
\quad e'' + 2 : \texttt{store}\ l; \\
\quad \ulcorner e'' + 3, S_2, e - 3\urcorner_{\texttt{m}}; \\
\quad e - 2 : \texttt{load}\ l; \\
\quad e - 1 : \texttt{athrow}; \\
\quad e : \texttt{nop} \\
\quad \mathsf{addExcHandler}(\texttt{m},\ s,\ e',\ e'' + 2,\ \texttt{Exception})
\end{array}
$$

Figure 7.3: Definition of the compiler for statements that change the exception handler table

are affected. Particularly, the compiler adds a new element in the table of loop invariants of the method `m`. This new element relates the index $e' + 1$ with the loop invariant `INV` and the list of modified expressions `modif`. In the following, the index of the first instruction of the compilation of the loop test (in the figure this is the index $e' + 1$) will be frequently referred to. Thus, we introduce the notation $\mathbf{loopEntry}_S$ to refer to the first instruction in the compilation of the test of the loop statement S. Actually, in the control flow graph this instruction represents a loop entry instruction following Def. 3.9.1, Section 3.9.

$$
\begin{array}{l}
\ulcorner s, \texttt{while}\ (E_1\ \mathit{cond}\ E_2)[\texttt{INV}, \texttt{modif}]\ \ \{S\}, e\urcorner_{\texttt{m}} = \\
\quad s : \texttt{goto}\ e' + 1; \\
\quad \ulcorner s + 1, S, e'\urcorner_{\texttt{m}}; \\
\quad \ulcorner e' + 1, E_1, e''\urcorner_{\texttt{m}}; \\
\quad \ulcorner e'' + 1, E_2, e - 1\urcorner_{\texttt{m}}; \\
\quad e : \texttt{if\_}\mathit{cond}\ s + 1; \\
\\
\quad \mathsf{addLoopSpec}(\texttt{m},\ e' + 1,\ \texttt{INV},\ \texttt{modif})
\end{array}
$$

Figure 7.4: Definition of the compiler for the loop statement

For an illustration, we show in Fig. 7.5 the source of method square which calculates the the square of the parameter `i`. First, the absolute value of i is stored in the variable v. Then the while statement calculates the sum of the impair positive numbers whose whole division by 2 is smaller than v which is the square of `i`. The example is also provided with specification written in JML. The specification states that the method returns the square of its parameter and that the loop invariant is (0 <= s) && (s <= v) && sqr == s*s. In Fig. 7.6, we then show the respective compilation of method square. The example shows the correspondence between the bytecode (left) resulting from the compilation described above and the source lines(right) of method square from Fig. 7.5.

We can see in the figure how the branch statement is compiled (bytecode instructions from 4 to 12). It also shows us the somewhat unusual way into which the while statement is compiled. More particularly, the compilation of the test of the **while** is after its body while semantically the test must be executed at every iteration before the loop body. The compilation is actually correct because the instruction 15 **goto** 28 jumps to the compilation of the line **while**(s<v) and thus, the execution proceeds in the expected way. We have also marked the instructions which correspond

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

Figure 7.5: METHOD SQUARE WRITTEN IN OUR SOURCE LANGUAGE

to the loop entry and loop end which are the instructions at index 28 and 27.

Note that the bytecode has been generated by a standard Java compiler. We have modified the compilation of s = s+1; to match the definition of our compiler[2] and introduced the `nop` instructions. Note also that we keep the same names on bytecode and source. This is done for the sake of simplicity and in this chapter, we shall use always this convention.

7.1.2 Properties of the compiler function

In this subsection, we will focus on the properties of the bytecode produced by the compiler presented above. These properties although a straightforward consequence of the compiler definition are actually important for establishing formally the equivalence between source and bytecode proof obligations. Note that a standard non-optimizing Java compiler generates code which respects them and thus, we do not restrict at all the compilation process. We outline the categories of properties that we shall consider:

- compilation of a statement (expressions) may contain instructions which either target instructions inside the compilation of a statement or the first instruction of the next statement, if there is next statement (Such property is given in Lemma 7.1.2.1).
- expressions are compiled into a sequence of instructions which does not contains jumps (treated in Lemma 7.1.2.2).
- a compilation of a source statement contains cycles only if the source statement contains cycles (treated in Lemma 7.1.2.3).
- exception handler preservation (Lemmas 7.1.2.4, 7.1.2.5 treat those cases).

In the following, we use the notation $\mathcal{SE}$ when we refer both to statements S and expressions E. Let us take a closer look at the properties in question.

The first property that we observe is that the last instruction e in the compilation $\ulcorner s, S, e \urcorner_{\mathtt{m}}$ of a statement S is always in execution relation (see Fig. 3.10 for the definition of execution relation between instructions) with the instruction $e + 1$.

In order to get a precise idea of what we mean, the reader may take a look at the example in Fig. 7.6. There, we can remark that the last instruction of the compilation of the statement **int** sqr = 0; is the instruction 1 **store** sqr and that it is in execution relation with the instruction

[2] the Java compiler will tend to compile incrementation into **iinc**

```
                    int square( int i )
0 const 0           int sqr  = 0;
1 store sqr

2 const 0           int v = 0;
3 store v

4 load i            if ( i >= 0)
5 ifge 10

6 load i                else{v =−i;}
7 neg
8 store v

9 goto 12

10 load i               then{v = i;}
11 store v

12 nop

13 const 0          int s = 0;
14 store s

15 goto 28

16 load sqr             sqr = sqr + 2∗s + 1;
17 const 2
18 load s
19 mul
20 add
21 const 1
22 add
23 store sqr

24 load s               s = s+1;
25 const 1
26 add
27 store s          LOOP END

28 load s           LOOP ENTRY while( s < v )
29 load v
30 if_icmplt 16

31 load sqr         return sqr;
32 return
```

Figure 7.6: RELATION BETWEEN BYTECODE AND SOURCE CODE OF METHOD SQUARE FROM FIG. 7.5

at index 2. The same holds also for the compilation of the **if** statement where the last instruction in its compilation is 12 **nop** and is in execution relation with 13 **const** 0. Actually, the compilation of every statement or expression in the example has this property.

Lemma 7.1.2.1 (Compilation of statements and expressions). *For any statement or expression $\mathcal{SE}$ which does not terminate on* `return`*or* `athrow`*, start label s and end label e, the compiler will produce a list of bytecode instruction $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$ such that instruction $e+1$ may execute after e, i.e. $e \longrightarrow e+1$*

Next, we give a definition for a set of instructions such that they execute sequentially which will be used for establishing afterwords the properties of the bytecode instructions resulting in expression compilation.

Definition 7.1.2.1 (Block of instructions). *We say that the list of instructions $l = [i_1 \dots i_n]$ is a block of instructions in the compilation of method* `m` *if*

- *none of the instructions is a target of an instruction i_j which does not belong to l except for i_1*
- *none of the instructions in the set is a jump instruction, a* `return` *or an* `athrow` *instruction i.e. $\forall j, i_1 \leq j < i_n, (j \longrightarrow j+1)$*
- *none of the instructions except for the first one can be a loop entry in the sense of Def. 3.10 $\forall j, i_1 \leq j < i_n, \neg\ (j \longrightarrow^l j+1)$*

We denote such a list of instructions with $i_1; ...; i_n$

The next lemma states that the compilation of an expression E results in a block of bytecode instructions. For instance, consider the compilation of the expression sqr + 2∗s + 1; in Fig. 7.6 which comprised between instructions 16-22. Instructions 16-22 satisfy the three points from the above definition.

Lemma 7.1.2.2 (Compilation of expressions). *For any expression E, starting label s and end label e, the compilation $\ulcorner s, E, e \urcorner_{\mathtt{m}}$ is a block of bytecode instruction in the sense of Def. 7.1.2.1*

The following statement concerns loops. In particular, it says that a cycle appears in the compilation of a statement S only if S contains a loop. Moreover, cycles in the bytecode produced by the compiler have exactly one entry. For instance, we can see in Fig. 7.6 that the unique cycle in the bytecode corresponds to the source loop and that the instruction marked with LOOP END and the instruction marked with LOOP ENTRY correspond respectively to the end instruction in the compilation of the source loop body and to start instruction in the compilation of the compilation of the **while** test. Stated formally, we get the following property.

Lemma 7.1.2.3 (Cycles in the control flow graph). *The compilation $\ulcorner s, S, e \urcorner_{\mathtt{m}}$ of a statement S may contain an instruction k and j which are respectively a loop entry and a loop end in the sense of Def.3.9.1 (i.e. there exists j such that $j \longrightarrow^l k$) if and only if S contains a substatement S' which is a loop statement:*

$$j = \textbf{\textit{loopEntry}}_{S'} - 1 \wedge k = \textbf{\textit{loopEntry}}_{S'}$$

In the following, we will need the notion of substatement relation. For denoting that $\mathcal{SE}'$ is a substatement of $\mathcal{SE}$ (i.e. $\mathcal{SE}'$ is contained in $\mathcal{SE}$) we shall use the notation $\mathcal{SE}[\mathcal{SE}']$. We also use the notion of a direct substatement(subexpression) which means the following:

Definition 7.1.2.2 (Direct substatement(subexpression)). *The statement (expression) $\mathcal{SE}'$ is a direct substatement (subexpression) of $\mathcal{SE}$ if $\mathcal{SE}'$ is contained in $\mathcal{SE}$ and there does not exist $\mathcal{SE}''$ such that $\mathcal{SE}''$ is contained in $\mathcal{SE}$ and $\mathcal{SE}'$ is contained in $\mathcal{SE}''$. For denoting that $\mathcal{SE}'$ is a direct substatement of $\mathcal{SE}$ we use the notation $\mathcal{SE}[[\mathcal{SE}']]$.*

Direct substatements are also substatements. But the contrary does not always hold. For, instance if we have $S = \mathtt{try}\{S_1; S_2\}\mathtt{catch}(\text{Exc } e)\{S_3\}$, the statement $S_1; S_2$ is a direct substatement of S and is also its substatement. However, S_2 is a substatement of S but is not a direct substatement of S.

The following property states that the exceptions thrown by a statement which is not a try catch and the exceptions thrown by its direct substatement will be handled by the same exception handler, i.e. the result of the function *findExcHandler* will be the same if we pass as an argument their respective last instructions. Note that we can establish this property only for direct substatement because substatements could be contained in try catch substatements and thus, they break the property. Next, we shall focus on properties which concern the compilation of exception handlers. As we saw in the previous section, the compiler keeps track of the exception handlers by adding them in the exception handler table.

We illustrate this by the example in Fig. 7.7 which shows both the bytecode (on the left) and source code(on the right) of the method abs. The method abs gets as parameter n an object

```
int abs(Number n)                 int abs(Number n )
0 const 0                         int abs = 0;
1 store abs

                                  try {
2 load n                              if (n.value >= 0)
3 getfield Number.value
4 iflt 9

5 load n                                  then {abs = n.value;}
6 getfield Number.value
7 store abs

8 goto 21

9 load n                                  else {abs = −n.value;}
18 getfield Number.value
19 neg
20 store abs
21 nop

22 goto 26
                                  }
                                  catch(NullPointerExc e) {
23 store e                            abs = −1
24 const −1
25 store abs
                                  }
26 nop

27 load abs                       return abs
28 return

abs.ExcHandler=
  startPc     =   2
  endPc       =   21
  handlerPc   =   22
  exc         =   NullExc
```

Figure 7.7: RELATION BETWEEN BYTECODE AND SOURCE CODE OF METHOD ABS

reference of class type Number. The class Number models a integer numbers where the value of the integer represented by an instance of the class is stored in field value. Thus, method abs returns the absolute value of the integer field n.value of the parameter n. The absolute value is stored in the method local variable abs. This is implemented by the if statement. However, in the test of the if statement, we dereference the parameter n ignoring whether it is **null** or not. That is why the **if** statement may throw a `NullExc` and thus, it is wrapped in a **try catch** block. The exception handler (the statement following the **catch** keyword) stores -1 in the local variable abs which stands for that the the object n is **null**. Finally the method returns the value of the local variable abs.

Consider in Fig. 7.7 the bytecode version of the program. The last instruction in the compilation of the **if** statement is the instruction at index 21 and the last instruction in the compilation of the **else** branch is 7. For any exception type `Exc`, the application *findExcHandler*(`Exc` ,21,abs.**ExcHandler**) returns the same value as *findExcHandler*(`Exc` ,7,abs.**ExcHandler**).

Lemma 7.1.2.4 (Exception handler property for statements). *Assume that we have a statement S which is not a try catch neither a try finally statement in method* `m`*. Assume that statement S' is its direct substatement, i.e. $S[[S']]$. Let their respective compilations be $\ulcorner s, S, e \urcorner_{\mathtt{m}}$*

and $\ulcorner s', S', e' \urcorner_{\mathtt{m}}$*, then the exception handlers for the instruction points* e *and* e' *are the same:*

$$\forall \mathtt{Exc}, \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = \mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS})$$

A similar property can be established for expressions. This time however, the property holds for every instruction in the compilation of an expression.

Lemma 7.1.2.5 (Exception handler property for expressions). *For every expression* E*, for every instruction* i *inside the compilation* $\ulcorner s, E, e \urcorner_{\mathtt{m}}$*, any exception type* `Exc` *thrown by* i *will be handled by the same exception handler as in the case where* `Exc` *is thrown by the last instruction* e *in the compilation, i.e.*

$$\forall \mathtt{Exc}, \forall\ i, s \leq i < e, \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = \mathit{findExcHandler}(\mathtt{Exc}, i, \mathtt{m.excHndlS})$$

The next property states that a try catch statement `try` $\{S_1\}$ `catch` (`ExcType` *Var*) $\{S_2\}$ is such that any exception `E` which is a subtype of `ExcType` and thrown by the last instruction in the compilation of the try statement S_1 is handled by the compilation of the exception handler S_2. Moreover, an an exception `E'` which is not a subtype of `ExcType` thrown by the last instruction in the compilation of S_1 or the last instruction in the compilation of S_2 will be handled by the same exception handler.

For instance, the last instruction in the compilation of the try catch statement in Fig. 7.7 is the instruction at index 26 and the last instruction in the compilation of the try statement is the instruction at index 21. From the exception handler, we can see that any exception except `NullExc` which might be thrown from these instructions will be handled in the same way.

Lemma 7.1.2.6 (Exception handlers and try catch statements). *For every try catch statement* `try` $\{S_1\}$ `catch` (`ExcType` *Var*)$\{S_2\}$ *whose compilation results in* $\ulcorner s, S_1, e' \urcorner_{\mathtt{m}}; e'+1 : \mathtt{goto}\ e;$ $\ulcorner e'+2, S_2, e-1 \urcorner_{\mathtt{m}}; e : \mathtt{nop}$ *and modifies the exception handler table* addExcHandler(`m`, s, e', $e'+2$, `ExcType`) *is such that the following holds*

$$\forall \mathtt{Exc}, \neg(\mathtt{Exc} <: \mathtt{ExcType}) \Rightarrow \left(\begin{array}{l} \mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS}) = \\ \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) \end{array} \right)$$
$$\wedge$$
$$\mathit{findExcHandler}(\mathtt{ExcType}, e', \mathtt{m.excHndlS}) = e' + 2$$

7.2 Establishing the equivalence between verification conditions on source and bytecode level

In the following, we proceed with establishing the equivalence between source and bytecode proof obligations. As we have already seen, the source programming language supports expressions and statements. Because statements and expressions play different roles in a source program language they need a different treatment here in this proof. Let us remind briefly about the semantics of these constructs and their compilation. Expressions evaluate to a value and thus, their compilation affects the operand stack on execution time. As we have discussed previously, because we compile for a stack based virtual machine, an expression compilation results in a sequence of instructions whose execution must leave on the stack top the expression value. Statements have a different role in the language. They do not have values, but they control the control flow in the program.

We focus now on the relation between the *wp* predicate transformer functions for expressions on bytecode and source level. But before entering into technical details, we illustrate this relation by an example given in Fig.7.8. The figure contains three parts. The first part shows the compilation of a source expression sqr + 2∗s in a method `m` starting at index `i`. There we show the steps that the compiler will take for the compilation of the expression: compile the access of the variable sqr, the multiplication 2∗s and compile their addition.

The second part calculates the preconditions of the instructions resulting from the expression compilation against a postcondition that the stack top element is equal to 5 ($\mathbf{st}(\mathbf{cntr}) = 5$). Actually, this postcondition requires that the evaluation of the expression must be equal to 5. This is because as we said in the beginning of the section the compiler translates source expressions to

a sequence of bytecode instructions such that their execution must leave the expression value on the stack top. Note that every instruction is followed by its postcondition and is preceded by its weakest precondition. This means that the weakest predicate of an instruction is the postcondition of the predecessor instruction. This is because compilation of expressions may not contain loop entries (see previous Section 7.1.2.2, page 102).

The third part shows how the precondition of the source expression is calculated w.r.t. the postcondition $v = 5$ where v is the special logical variable which stands for the value of the expression sqr + 2∗s. Thus, the bytecode postcondition $\mathbf{st}(\mathbf{cntr}) = 5$ and the source postcondition $v = 5$ express the same condition respectively on source and bytecode. Note that substituting the abstract variable v with the stack top in the source postcondition $v = 5[v\backslash\mathbf{st}(\mathbf{cntr})]$ results in the bytecode postcondition.

Let us focus on the intermediate stages in the calculation of the precondition of the whole expression. We may remark that the resulting postcondition of the source expression 2∗s is $v_{sqr} + v_{2*s} = 5$ and that the postcondition at line 1.9 for the last instruction of its compilation (instruction i+3: **mul**) is $\mathbf{st}(\mathbf{cntr} - 1) + \mathbf{st}(\mathbf{cntr}) = 5$. Substituting in the source postcondition the abstract variable v_{2*s} with $\mathbf{st}(\mathbf{cntr})$ and v_{sqr} with $\mathbf{st}(\mathbf{cntr}\text{-}1)$ results in the bytecode postcondition

$$\textit{(1)}\ (v_{sqr} + v_{2*s} = 5)[v_{2*s}\backslash\mathbf{st}(\mathbf{cntr})][v_{sqr}\backslash\mathbf{st}(\mathbf{cntr} - 1)] \equiv \mathbf{st}(\mathbf{cntr} - 1) + \mathbf{st}(\mathbf{cntr}) = 5$$

We can remark that the precondition of the first instruction (i+1: **const** 2) of the compilation of the expression 2∗s is $\mathbf{st}(\mathbf{cntr}) + 2 * \mathrm{s} = 5$. We may remark that the precondition of the source expression 2∗s in the third part of the figure is equivalent to $v_{sqr} + 2 * \mathrm{s} = 5$. Substituting in the source precondition the abstract variable v_{sqr} with $\mathbf{st}(\mathbf{cntr})$ results in a formula which is equivalent to the bytecode precondition

$$\textit{(2)}\ (v_{sqr} + 2{*}\mathrm{s} = 5)[v_{sqr}\backslash\mathbf{st}(\mathbf{cntr})] \equiv \mathbf{st}(\mathbf{cntr}) + 2{*}\mathrm{s} = 5$$

Equivalences *(1)* and *(2)* give the intuition of how the predicate transformers work over source expressions and their bytecode compilation. The predicate transformer over the instructions representing an expression substitutes the stack top element expression $\mathbf{st}(\mathbf{cntr})$ with the value of the expression and the stack counter is decremented. The predicate transformer over a source expression calculates a predicate where the abstract variable representing its value is substituted with the expression value. More formally, this is expressed in the next lemma.

Theorem 7.2.1. *For every expression E and its compilation $\ulcorner s, E, e \urcorner_{\mathtt{m}}$ let have a formula ψ and expressions $w_1 \ldots w_k, k \geq 0$ such that*

$$\psi[v\backslash\mathbf{st}(\mathbf{cntr})][w_i\backslash\mathbf{st}(\mathbf{cntr} - i)]_{i=1\ldots k} = \mathit{inter}(e, e + 1, \mathtt{m}) \wedge$$
$$\forall \mathrm{Exc}, \mathsf{excPost}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathrm{Exc}, e)$$

then the following holds

$$wp^{src}(E, \psi, \mathsf{excPost}, \mathtt{m})_v[w_i\backslash\mathbf{st}(\mathbf{cntr} - i + 1)]_{i=1\ldots k} = wp(\ s, \mathtt{m})$$

Proof: the proof is by structural induction over the expression structure. We sketch the cases for field access and arithmetic expressions.

field access From the compiler definition we have

$$\ulcorner s, E.\mathtt{f}, e \urcorner_{\mathtt{m}} = \begin{array}{l} \ulcorner s, E, e - 1 \urcorner_{\mathtt{m}}; \\ e : \mathtt{getfield\ f} \end{array} \tag{7.2.1.1}$$

By initial hypothesis we have that we have a formula ψ over the source language such that if the abstract variables w_i which stand for expression values are substituted with stack expressions the formula will be the same as the formula $\mathit{inter}(e, e + 1, \mathtt{m})$. Or formally, for some k such that $k \geq 0$ we have the following equality:

$$\psi[v\backslash\mathbf{st}(\mathbf{cntr})][w_i\backslash\mathbf{st}(\mathbf{cntr} - i)]_{i=1\ldots k} = \mathit{inter}(e, e + 1, \mathtt{m}) \tag{7.2.1.2}$$

$$
\begin{array}{lcl}
 & & \ulcorner i, \mathrm{sqr}, i \urcorner_{\mathtt{m}} \\
\ulcorner i, \mathrm{sqr}\ +\ 2{*}\mathrm{s}, i+4 \urcorner_{\mathtt{m}} & = & \ulcorner i+1, 2{*}\mathrm{s}, i+3 \urcorner_{\mathtt{m}} \\
 & & \mathrm{i}{+}4\text{: } \textbf{add} \\
\text{where} & & \\
\ulcorner i, \mathrm{sqr}, i \urcorner_{\mathtt{m}} & = & \mathrm{i}\text{: } \textbf{load } \mathrm{sqr} \\
 & & \\
 & & \mathrm{i}{+}1\text{: } \textbf{const } 2 \\
\ulcorner i+1, 2{*}\mathrm{s}, i+3 \urcorner_{\mathtt{m}} & = & \mathrm{i}{+}2\text{: } \textbf{load } \mathrm{s} \\
 & & \mathrm{i}{+}3\text{: } \textbf{mul}
\end{array}
$$

$$
\begin{array}{ll}
1.1 & \mathrm{sqr} + 2 * \mathrm{s} = 5 \\
1.2 & \mathrm{i}\text{: } \textbf{load } \mathrm{sqr} \\
1.3 & \textbf{st}(\textbf{cntr}) + 2 * \mathrm{s} = 5 \\
1.4 & \mathrm{i}{+}1\text{: } \textbf{const } 2 \\
1.5 & \textbf{st}(\textbf{cntr} - 1) + \textbf{st}(\textbf{cntr}) * \mathrm{s} = 5 \\
1.6 & \mathrm{i}{+}2\text{: } \textbf{load } \mathrm{s} \\
1.7 & \textbf{st}(\textbf{cntr} - 2) + \textbf{st}(\textbf{cntr} - 1) * \textbf{st}(\textbf{cntr}) = 5 \\
1.8 & \mathrm{i}{+}3\text{: } \textbf{mul} \\
1.9 & \textbf{st}(\textbf{cntr} - 1) + \textbf{st}(\textbf{cntr}) = 5 \\
1.10 & \mathrm{i}{+}4\text{: } \textbf{add} \\
1.11 & \textbf{st}(\textbf{cntr}) = 5
\end{array}
$$

$$
\begin{array}{ll}
2.1 & wp^{src}(\mathrm{sqr}\ +\ 2{*}\mathrm{s}, v = 5, \mathsf{excPost}, \mathtt{m})_v = \\
2.2 & wp^{src}(\mathrm{sqr}, wp^{src}(2{*}\mathrm{s}, (v = 5)[v \backslash v_{sqr} + v_{2*s}], \mathsf{excPost}, \mathtt{m})_{v_{2*s}}, \mathsf{excPost}, \mathtt{m})_{v_{sqr}} = \\
2.3 & wp^{src}(\mathrm{sqr}, wp^{src}(2, wp^{src}(\mathrm{s}, (v_{sqr} + v_{2*s} = 5)[v_{2*s} \backslash v_2 * v_s], \mathsf{excPost}, \mathtt{m})_{v_s}, \mathsf{excPost}, \mathtt{m})_{v_2}, \mathsf{excPost}, \mathtt{m})_{v_{sqr}} = \\
2.4 & wp^{src}(\mathrm{sqr}, wp^{src}(2, (v_{sqr} + v_2 * v_s = 5)[v_s \backslash s], \mathsf{excPost}, \mathtt{m})_{v_2}, \mathsf{excPost}, \mathtt{m})_{v_{sqr}} = \\
2.5 & wp^{src}(\mathrm{sqr}, (v_{sqr} + v_2 * \mathrm{s} = 5)[v_2 \backslash 2], \mathsf{excPost}, \mathtt{m})_{v_{sqr}} = \\
2.6 & (v_{sqr} + 2 * \mathrm{s} = 5)[v_{sqr} \backslash \mathrm{sqr}] = \\
2.7 & \mathrm{sqr} + 2 * \mathrm{s} = 5
\end{array}
$$

Figure 7.8: Expression, its compilation and their respective preconditions

From the definition of the *wp* for source field access expressions, we have also

$$
\begin{array}{rl}
 & wp^{src}(E\mathtt{.f}, \psi, \mathsf{excPost}, \mathtt{m})_v = wp^{src}(E, \psi', \mathsf{excPost}, \mathtt{m})_{v_1} \\
where & \\
 & v_1 \neq \textbf{null} \Rightarrow \psi[v \backslash v_1\mathtt{.f}] \\
\psi' = & \wedge \\
 & v_1 = \textbf{null} \Rightarrow \mathsf{excPost}(\mathtt{NullExc})
\end{array}
\tag{7.2.1.3}
$$

Because the execution relation between e and $e+1$ is not a loop backedge by Lemma 7.1.2.2 which establishes that compilation of expressions does not contain loop entries and from the Def. 5.3.1.1 of the function *inter*: *(5)* $inter(e-1, e, \mathtt{m}) = wp(\ e, \mathtt{m})$

definition of the *wp* function for `getfield`

$$
\begin{array}{rl}
 & \textbf{st}(\textbf{cntr}) \neq \textbf{null} \Rightarrow\ inter(e, e+1, \mathtt{m})[\textbf{st}(\textbf{cntr}) \backslash \textbf{st}(\textbf{cntr})\textbf{.f}] \\
wp(\ e, \mathtt{m}) = & \wedge \\
 & \textbf{st}(\textbf{cntr}) = \textbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostIns}(\mathtt{NullExc}, e)
\end{array}
$$

From (7.2.1.1), (7.2.1.2) and (7.2.1.3) and the above facts for the *wp* function over bytecode and source respectively, we conclude:

$$
\psi'[v_1 \backslash \textbf{st}(\textbf{cntr})][w_i \backslash \textbf{st}(\textbf{cntr} - i)]_{i=1\ldots k} =\ inter(e-1, e, \mathtt{m}) \tag{7.2.1.4}
$$

For the exceptional postcondition functions on source and bytecode we get from Lemma 7.1.2.5 and the initial hypothesis that

$$
\forall \mathrm{Exc}, \mathsf{excPost}(\mathrm{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathrm{Exc}, e-1) \tag{7.2.1.5}
$$

We apply the induction hypothesis over (7.2.1.4) and (7.2.1.5) and obtain

$$
wp^{src}(E, \psi', \mathsf{excPost}, \mathtt{m})_{v_1}[w_i \backslash \textbf{st}(\textbf{cntr} - i + 1)]_{i=1\ldots k} = wp(s\ , \mathtt{m})
$$

Finally, (7.2.1.3) and the last equality allows us to conclude that this case holds

arithmetic expression From the compiler definition we have

$$(1)\ \ulcorner s, E_1\ op\ E_2, e\urcorner_{\mathtt{m}} = \begin{array}{l}\ulcorner s, E_1, e'\urcorner_{\mathtt{m}};\\ \ulcorner e'+1, E_2, e-1\urcorner_{\mathtt{m}};\\ e : op\end{array}$$

As in the previous case, we can conclude that there exists a formula ψ over source expressions with the following property

$$\psi[v\backslash\mathbf{st}(\mathbf{cntr})][w_i\backslash\mathbf{st}(\mathbf{cntr}-i)]_{i=1\ldots k} = \ inter(e, e+1, \mathtt{m}) \tag{7.2.1.6}$$

We also remind the formulation of the *wp* function for source arithmetic expressions

$wp^{src}(E_1\ op\ E_2, \psi, \mathsf{excPost}^{src}, \mathtt{m})_v = wp^{src}(E_1, wp^{src}(E_2, \psi', \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$
with $\psi' = \psi[v\backslash v_1 op v_2]$

It follows from Lemma 7.1.2.2 about expressions that the compilation of an expression results in a list of instructions which does not contain loop entries. Thus, from the Def.5.3.1.1 of the function *inter* we get

$$inter(e-1, e, \mathtt{m}) = wp(\ e, \mathtt{m})$$

It follows from the *wp* for an arithmetic instruction

$$wp(\ e, \mathtt{m}) = \ inter(e, e+1, \mathtt{m})[\mathbf{cntr}\backslash\mathbf{cntr}-1][\mathbf{st}(\mathbf{cntr}-1)\backslash\mathbf{st}(\mathbf{cntr}-1)\ op\ \mathbf{st}(\mathbf{cntr})]$$

which because of (7.2.1.6) is equal to

$$\psi[v\backslash\mathbf{st}(\mathbf{cntr})][w_i\backslash\mathbf{st}(\mathbf{cntr}-i)]_{i=1\ldots k}[\mathbf{cntr}\backslash\mathbf{cntr}-1][\mathbf{st}(\mathbf{cntr}-1)\backslash\mathbf{st}(\mathbf{cntr}-1)\ op\ \mathbf{st}(\mathbf{cntr})]$$

Because the formula ψ refers to source expressions and does not contain stack expressions we can conclude by applying substitutions

$$\psi[v\backslash\mathbf{st}(\mathbf{cntr}-1)op\mathbf{st}(\mathbf{cntr})][w_i\backslash\mathbf{st}(\mathbf{cntr}-i+1)]_{i=1\ldots k}$$

which is equal to

$$\psi'[v_2\backslash\mathbf{st}(\mathbf{cntr})][v_1\backslash\mathbf{st}(\mathbf{cntr}-1)][w_i\backslash\mathbf{st}(\mathbf{cntr}-i+1)]_{i=1\ldots k}$$

From the last equalities we can apply the induction hypothesis over E_2 and ψ' and get

$$wp^{src}(E_2, \psi', \mathsf{excPost}^{src}, \mathtt{m})_{v_2}[v_1\backslash\mathbf{st}(\mathbf{cntr})][w_i\backslash\mathbf{st}(\mathbf{cntr}-i+1)]_{i=1\ldots k} = wp(\ e'+1, \mathtt{m}) \tag{7.2.1.7}$$

As it follows from Lemma 7.1.2.2 we have that $inter(e', e'+1, \mathtt{m}) = wp(\ e'+1, \mathtt{m})$ and because of the last result (7.2.1.7), we can apply the induction hypothesis over E_1, we conclude that this case holds.

Qed.

The next lemma states the same property but this time for the compilation of statements.

Theorem 7.2.2. *Let us have the statement S, its compilation $\ulcorner s, S, e\urcorner_{\mathtt{m}}$ in method $\mathtt{m}$, the formula ψ and the exceptional function* $\mathsf{excPost} : \mathtt{ExcType} \to P$ *such that*

1. *if $e+1$ exists then $\psi = \ inter(e, e+1, \mathtt{m})$*
2. *if $e+1$ does not exist and $e =$* `return` *then $\psi = \mathtt{m}.\mathsf{normalPost}$*

3. $\forall \mathtt{Exc}, \mathsf{excPost}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e)$

then the following holds:

$$wp^{src}(S, \psi, \mathsf{excPost}, \mathtt{m}) = wp(s\ , \mathtt{m})$$

Proof : the proof is by structural induction on statements and uses the properties of the compiler shown before. We give here the proof for the cases of compositional statement, while and try catch statement. The rest of the cases proceed in a similar way.

compositional statement

$$wp^{src}(S_1; S_2, \psi, \mathsf{excPost}, \mathtt{m}) = wp^{src}(S_1, wp^{src}(S_2, \psi, \mathsf{excPost}^{src}, \mathtt{m}), \mathsf{excPost}^{src}, \mathtt{m})$$

From the compiler definition we get

$$\ulcorner s, S_1; S_2, e \urcorner_{\mathtt{m}} = \ulcorner s, S_1, e' \urcorner_{\mathtt{m}}; \ulcorner e' + 1, S_2, e \urcorner_{\mathtt{m}}$$

It follows from the induction hypothesis over S_2 and the initial hypothesis about the post-condition

$$wp^{src}(S_2, \psi, \mathsf{excPost}, \mathtt{m}) = wp(e' + 1\ , \mathtt{m}) \tag{7.2.2.1}$$

Lemma B.1 states that e and $e+1$ are in execution relation Lemma 7.1.2.3 states that loop edges ($\longrightarrow^l$) appear only on loop statement compilation and thus, the edge between e' and $e'+1$ is not a loop edge. In that case from Def. 5.3.1.1 of the function *inter* we get

$$inter(e', e'+1, \mathtt{m}) = wp(e' + 1\ , \mathtt{m}) \tag{7.2.2.2}$$

The statement S_1 is a strict substatement of $S_1; S_2$ and thus, from Property 7.1.2.4 follows

$$\forall \mathtt{Exc}, \mathtt{m}.\mathit{findExcHandler}(\ \mathtt{Exc}, e, \mathtt{m}.\mathsf{excHndlS}) = \mathtt{m}.\mathit{findExcHandler}(\ \mathtt{Exc}, e', \mathtt{m}.\mathsf{excHndlS})$$

From the above conclusion and the Def. 5.3.2.1 of function excPostIns

$$\forall \mathtt{Exc},\ \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e') \tag{7.2.2.3}$$

From (7.2.2.1), (7.2.2.2) and (7.2.2.3) we get that

$$wp^{src}(S_1, wp^{src}(S_2, \psi, \mathsf{excPost}, \mathtt{m}), \mathsf{excPost}, \mathtt{m}) = wp(s\ , \mathtt{m})$$

From this last equality we conclude that this case holds.

while statement Let us remind the definition of the *wp* for while statements

$$\begin{array}{l} wp^{src}(\mathtt{while}\ (E_1\ cond\ E_2)\ [\mathtt{INV}, \mathtt{modif}]\ \ \{S\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) = \\ \mathtt{INV} \wedge \\ \forall\ m, m \in \mathtt{modif}, \\ \mathtt{INV} \Rightarrow wp^{src}(E_1, wp^{src}(E_2, \mathsf{P}, \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1} \\ \\ where \\ \mathsf{P} = (v_1\, cond v_2) \Rightarrow\ wp^{src}(S, \mathtt{INV}, \mathsf{excPost}^{src}, \mathtt{m}) \\ \wedge \\ \neg(v_1\, cond v_2) \Rightarrow \mathsf{nPost}^{src} \end{array}$$

We remind that the compilation of the while statement results as follows

$$\begin{array}{l} \ulcorner s, \mathtt{while}\ (E_1\ cond\ E_2)[\mathtt{INV}, \mathtt{modif}]\ \ \{S\}, e \urcorner_{\mathtt{m}} = \\ s : \mathtt{goto}\ e' + 1; \\ \ulcorner s + 1, S, e' \urcorner_{\mathtt{m}}; \\ \ulcorner e' + 1, E_1, e'' \urcorner_{\mathtt{m}}; \\ \ulcorner e'' + 1, E_2, e - 1 \urcorner_{\mathtt{m}}; \\ e : \mathtt{if\_}cond\ s + 1; \\ \\ \\ \mathsf{addLoopSpec}(\mathtt{m},\ e' + 1,\ \mathtt{INV},\ \mathtt{modif}) \end{array}$$

From Lemma 7.1.2.3 we know that the execution relation between e' and $e'+1$ is a loop execution relation. From Def. 5.3.1.1, 68 of the function *inter* we conclude that

$$inter(e', e'+1, \mathtt{m}) = \mathtt{INV} \tag{7.2.2.4}$$

From Lemma 7.1.2.4 we get that the exception handlers for the last index e' in the compilation of S and the index e are the same. Thus, from the initial hypothesis for the exception handler function, we conclude that

$$\forall \mathtt{Exc}, \mathsf{excPost}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e') \tag{7.2.2.5}$$

We can apply the induction hypothesis over (7.2.2.4) and (7.2.2.5) and statement S and we get

$$wp^{src}(S, \mathtt{INV}, \mathsf{excPost}^{src}, \mathtt{m}) = wp(s+1\ , \mathtt{m}) \tag{7.2.2.6}$$

From Lemma 7.1.2.3 we get that the execution relation between e and $s+1$ is not a loop execution relation. From def. 5.3.1.1, we conclude that $wp(s+1\ , \mathtt{m}) = \ inter(e, s+1, \mathtt{m})$. Thus, from (7.2.2.6) and the definition of the *wp* function for `if_`*cond*, we conclude that the *wp* of the instruction at index e is equivalent to

$$\begin{array}{l} wp(e\ , \mathtt{m}) = \\ \mathbf{st}(\mathbf{cntr})\ cond\ \mathbf{st}(\mathbf{cntr}-1) \Rightarrow wp^{src}(S, \mathtt{INV}, \mathsf{excPost}^{src}, \mathtt{m}) \\ \wedge \\ \neg(\mathbf{st}(\mathbf{cntr})\ cond\ \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow \ inter(e, e+1, \mathtt{m}) \end{array}$$

Using the latter, the previous Lemma 7.2.1 for expressions is applied twice over E_1 and E_2 and thus, we obtain

$$\begin{array}{l} wp^{src}(E_1, wp^{src}(E_2, \\ \\ wp(e'+1\ , \mathtt{m}) \end{array} \begin{array}{l} (v_1\, cond v_2) \Rightarrow \ wp^{src}(S, \mathtt{INV}, \mathsf{excPost}^{src}, \mathtt{m}) \\ \wedge \\ \neg(v_1\, cond v_2) \Rightarrow \mathsf{nPost}^{src} \end{array} , \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1} =$$

From Lemma 7.1.2.3, we get that $e'+1$ is a loop entry instruction. From Def. 5.3.1.1 of the function *inter*, we get that

$$\begin{array}{l} wp(s\ , \mathtt{m}) = \ inter(s, e'+1, \mathtt{m}) = \\ \mathtt{INV}\ \wedge \forall\ m, m \in \mathtt{modif}, (\mathtt{INV} \Rightarrow wp(e'+1\ , \mathtt{m})) \end{array}$$

From the last results, we obtain that this case holds.

try catch statement Let us remind the definition of the weakest precondition predicate transformer for try catch statement

$$\begin{array}{l} wp^{src}(\mathtt{try}\ \{S_1\}\mathtt{catch}\ (\mathtt{ExcType}\ \mathit{Var})\{S_2\}, \psi, \mathsf{excPost}, \mathtt{m}) = \\ wp^{src}(S_1, \psi, \mathsf{excPost}(\oplus \mathtt{ExcType} \rightarrow wp^{src}(S_2, \psi, \mathsf{excPost}, \mathtt{m})), \mathtt{m}) \end{array} \tag{7.2.2.7}$$

Also, by definition of the compiler function, we have that the compilation of try catch statement results in the following list of instructions

$$\begin{array}{l} \ulcorner s, \mathtt{try}\ \{S_1\}\ \mathtt{catch}\ (\mathtt{ExcType}\ \mathit{Var})\{S_2\}, e\urcorner_{\mathtt{m}} = \\ \ulcorner s, S_1, e'\urcorner_{\mathtt{m}}; e'+1 : \mathtt{goto}\ e; \ulcorner e'+2, S_2, e-1\urcorner_{\mathtt{m}}; e : \mathtt{nop} \\ \mathsf{addExcHandler}(\mathtt{m},\ s,\ e',\ e'+2,\ \mathtt{ExcType}) \end{array}$$

We apply the induction hypothesis over S_2 and the initial hypothesis

$$wp^{src}(S_2, \psi, \mathsf{excPost}, \mathtt{m}) = wp(\ e'+2, \mathtt{m})$$

From Lemma 7.1.2.6, we know that the indexes e and e' has the same exception handlers

$$\forall \mathtt{Exc}, \neg(\mathtt{Exc} <: \mathtt{ExcType}) \Rightarrow \\ (\mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS}) = \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS})) \wedge \\ \mathit{findExcHandler}(\mathtt{ExcType}, e', \mathtt{m.excHndlS}) = e' + 2$$

From the definition of excPostIns and the initial hypothesis about the exceptional postcondition functions for indexes e and e'

$$\forall \mathtt{Exc}, \mathsf{excPost}(\oplus \mathtt{ExcType} \rightarrow \mathit{wp}(\ e' + 2, \mathtt{m}))(\mathtt{Exc}) = \mathtt{m.excPostIns}(\mathtt{Exc}, e') \qquad (7.2.2.8)$$

Lemma 7.1.2.3 tells us that there are no loop edges between e' and $e' + 1$, $e' + 1$ and e, e and $e + 1$. Thus, we can conclude from the definition of the function *inter* that

$$\mathit{inter}(e', e' + 1, \mathtt{m}) = \ \mathit{inter}(e' + 1, e, \mathtt{m}) = \ \mathit{inter}(e, e + 1, \mathtt{m})$$

We apply induction hypothesis over the last conclusion and (7.2.2.7), (7.2.2.8), we get that this case holds.

Qed.

As a conclusion of the current chapter, we would like to make several remarks. Such an equivalence between proof obligations on source and bytecode and the fact that we have proof for the soundness of the bytecode verification condition generator gives us the soundness of the source weakest precondition calculus:

Theorem 7.2.3 (Soundness of bytecode *wp* implies soundness of source *wp*). *If the weakest precondition over bytecode programs is sound then the weakest precondition over source programs is sound.*

Another point is that here we ignore the difference between names and types on source and bytecode level. In this chapter, we have considered that the compiler does not change variable, class and method names. This is not true for Java compilers, as these names are basically compiled into indexes in the constant pool table. The second point in the formalization presented here is that the compilers compiles boolean types into integer types. This neither holds for real Java compiler. However, this difference is a minor detail and this means that the equivalence between source and bytecode verification conditions in Java can be established modulo names.

7.3 Related work

Several works dealing with the relation between the verification conditions over source and its compilation into a low level programming language exist.

Barthe, Rezk and Saabas in [21] also argue that verification conditions produced over source code and bytecode produced by a nonoptimizing compiler are equivalent. The verification condition generators over source and bytecode are such that verification conditions are discharged immediately when they are generated. This is different from our approach where the verification condition generator are propagated by the verification condition generator up to the entry point of a method body. However, the first technique requires much stronger annotations than the latter. The source language which they use supports method invokation, exception throwing and handling. The evaluation of expressions do not throw exceptions which simplifies the reasoning over expressions.

In [99], Saabas and Uustalu present a goto language provided with a compositional structure called SGoto. They give a Hoare logic rules for the SGoto language and compiler from the source language into SGoto. They show that if a source program has a Hoare logic derivation against a pre and postcondition then its compilation in SGoto will also have a Hoare logic derivation in the aforementioned Hoare logic rules for the SGoto language. A limitation of such an approach is that the bytecode logic is defined over structured pieces of code. In particular, in a PCC framework

this could be problematic. The first reason is that the code producer must supply along with the bytecode and the certificate the structure of the unstructured code that the client has used to generate the certificate. The second reason is that the certificate can be potentially large as it consists of a Hoare logic derivation and thus, contains annotation for every instruction in the bytecode. In a PCC scenario, this could slow down the downloading time of the certificate if it comes via a network or could be problematic if it must be stored on a device where it will be checked especially if the device has limited resources.

In [12], F.Bannwart and P.Muller show how to transform a Hoare style logic derivation on source Java like program into a Hoare style logic derivation of a Java bytecode like program. This solution however has similar shortcoming as the previously cited work.

We would like to make a final remark concerning the ability to use such a scheme. In order that a verification framework be able to exploit the equivalence between source and bytecode verification conditions it must be provided with both verification on source and bytecode. For instance, the Spec# [15] programming system being not provided with a mechanism for source verification may not benefit from this fact.

Chapter 8

Constrained memory consumption policies using verification condition generator

Memory consumption policies provide a means to control resource usage on constrained devices, and play an important role in ensuring the overall quality of software systems, and in particular resistance against resource exhaustion attacks. Such memory consumption policies have been previously enforced through static analysis, which yield automatic bounds at the cost of precision, or run-time analysis, which incur an overhead that is not acceptable for constrained devices.

Several approaches have been suggested to date to enforce memory consumption policies for programs; all approaches are automatic, but none of them is ideally suited for TPDs(short for Trusted Personal Devices), either for their lack of precision, or for the runtime penalty they impose on programs:

- *Static analysis and abstract interpretations:* in such an approach, one performs an abstract execution of an approximation of the program. The approximation is chosen to be coarse enough to be computable, as a result of which it yields automatically bounds on memory consumption, but at the cost of precision. Such methods are not very accurate for recursive methods and loops, and often fail to provide bounds for programs that contain dynamic object creation within a loop or a recursive method;

- *Proof-carrying code:* here the program comes equipped with a specification of its memory consumption, in the form of statements expressed in an appropriate program logic, and a certificate that establishes that the program verifies the memory consumption specification attached to it. The approach potentially allows for precise specifications. However, existing works on proof carrying code for resource usage sacrifice the possibility of enforcing accurate policies in favor of the possibility of generating automatically the specification and the certificate, in line with earlier work on certifying compilation;

- *Run-time monitoring:* here the program also comes equipped with a specification of its memory consumption, but the verification is performed at run-time, and interrupted if the memory consumption policy is violated.

  Such an approach is both precise and automatic, but incurs a runtime overhead which makes it unsuitable for TPDs.

In this chapter, we study the use of logical methods to specify and verify statically precise memory consumption policies for Java bytecode programs. In particular, we describe a methodology how to specify precise memory consumption policies for (sequential) Java.

Our broad conclusion is that logical methods can provide a suitable means to specify and verify expressive memory consumption policies, with a minimal runtime overhead.

The remainder of the chapter is organized as follows. Section 8.1 gives several motivating examples. In section 8.2, we begin by describing the principles of the methodology for writing

specifications for guaranteeing constrained memory consumption. Section 8.3 illustrates the approach with several examples on recursive methods, exception, inheritance. Finally, section 8.4 presents an overview of related approaches.

8.1 Motivating example

In order to illustrate the principles of our approach, let us consider the following program:

```
public void m(A a){
  if (a == null) {
    a = new A();
  }
   a.b = new B();
}
```

For modeling the memory consumption of this program, we introduce a **ghost** variable MemUsed that accounts for memory consumption; more precisely, the value of MemUsed at any given program point is meant to provide an upper bound to the amount of memory consumed so far. To keep track of the memory consumption, we perform immediately after every bytecode that allocates memory an increment of MemUsed by the amount of memory consumed by the allocation. Thus, if the programmer specifies that ka and kb is the memory consumed by the allocation of an instance of class A and B respectively, the program must be annotated as:

```
public void m(A a) {
  if (a == null) {
    a = new A();
    //@ set MemUsed = MemUsed + ka;
  }
   a.b = new B();
   //@ set MemUsed = MemUsed + kb;}
```

Such annotations allow to compute at run-time the memory consumption of the program. However, we are interested in static prediction of memory consumption, and resort to preconditions and postconditions to this end. Even for a simple example as above, one can express the specification at different levels of granularity. For example, fixing the amount of memory that the the program may use Max one can specify that the method will use at most ka + kb memory units and will not overpass the authorized limit to use Max with the following specification:

```
//@  requires MemUsed + ka + kb <= Max
//@ ensures  MemUsed <= \old(MemUsed) + ka + kb
public void m(A a) {
  if (a == null) {
    a = new A();
    //@ set MemUsed = MemUsed+ ka;
  }
   a.b = new B();
   //@ set MemUsed = MemUsed + kb;
}
```

Or try to be more precise and relate memory consumption to inputs with the following specification:

```
//@ requires a == null ==> MemUsed + ka + kb <= Max &&
    !(a == null) ==> MemUsed + kb <= Max
//@ ensures \old(a) == null ==>
    MemUsed <= \old(MemUsed) + ka + kb &&
    !(\old(a) == null) ==> MemUsed <= \old(MemUsed) + kb
public void m(A a) {
  if (a == null) {
```

```
    a = new A();
  }
  a.b = new B();
}
```

More complex specifications are also possible. For example, one can take into account whether the program will throw an exception or not. using (possibly several) exceptional postconditions stating that $\texttt{k}_\texttt{E}$ memory units are allocated in case the method exits on exception `E`.

8.2 Principles

Let us begin with a very simple memory consumption policy which aims at enforcing that programs do not consume more than some fixed amount of memory `Max`. To enforce this policy, we first introduce a ghost variable `MemUsed` that represents at any given point of the program the memory used so far. Then, we annotate the program both with the policy and with additional statements that will be used to check that the application respects the policy.

The precondition of the method m should ensure that there must be enough free memory for the method execution. Suppose that we know an upper bound of the allocations done by method m in any execution. We will denote this upper bound by `methodConsumption`(m). Thus there must be at least `methodConsumption`(m) free memory units from the allowed `Max` when method m starts execution. Thus the precondition for the method m is:

$$\textbf{requires } \texttt{MemUsed} + \texttt{methodConsumption}(\text{ m }) \leq \texttt{Max}.$$

The precondition of the program entry point (i.e., the method from which an application may start its execution) should state that the program has not allocated any memory, i.e. require that variable `MemUsed` is 0:

$$\textbf{requires } \texttt{MemUsed} == 0.$$

The normal postcondition of the method m must guarantee that the memory allocated during a normal execution of m is not more than some fixed number `methodConsumption`(m) of memory units. Thus for the method m the postcondition is:

$$\textbf{ensures } \texttt{MemUsed} \leq \textbf{old(}\texttt{MemUsed}\textbf{)} + \texttt{methodConsumption}(\text{ m }).$$

The exceptional postcondition of the method m must say that the memory allocated during an execution of m that terminates by throwing an exception `Exception` is not more than `methodConsumption`(m) units. Thus for the method m the exceptional postcondition is:

$$\textbf{exsures}\texttt{Exception } \texttt{MemUsed} \leq \textbf{old(}\texttt{MemUsed}\textbf{)} + \texttt{methodConsumption}(\text{ m }).$$

Loops must also be annotated with appropriate invariants. Let us assume that loop l iterates no more than $\texttt{iter}(l)$ and let $\texttt{loopConsumption}(l)$ be an upper bound of the memory allocated per iteration in l. Below we give a general form of loop specification w.r.t. the property for constraint memory consumption. The loop invariant of a loop l states that at every iteration the loop body is not going to allocate more than $\texttt{loopConsumption}(l)$ memory units and that the iterations are no more than $\texttt{iter}(l)$. We also declare an expression which guarantees loop termination, i.e. a variant (here an integer expression whose values decrease at every iteration and is always bigger or equal to 0).

$$\begin{array}{ll}
\textbf{modifies} & i, \texttt{MemUsed} \\
\texttt{INV}: & \texttt{MemUsed} \leq \texttt{MemUsed}^{Before_l} + i * \texttt{loopConsumption}(l) \\
 & \wedge \\
 & i \leq \texttt{iter}(l) \\
\texttt{variant}: & \texttt{iter}(l) - i
\end{array}$$

A special variable appears in the invariant, $\texttt{MemUsed}^{Before_l}$. It denotes the value of the consumed memory just before entering for the first time the loop l. At every iteration the consumed memory must not go beyond the upper bound given for the body of loop.

For every instruction that allocates memory the ghost variable `MemUsed` must also be updated accordingly. For the purpose of this paper, we only consider dynamic object creation with the bytecode `new`; arrays are left for future work and briefly discussed in the conclusion.

The function `allocInstance` : $Class \rightarrow int$ gives an estimation of the memory used by an instance of a class. Note that the memory allocated for a class instance is specific to the implementation of the virtual machine. At every program point where a bytecode `new A` is found, the ghost variable `MemUsed` must be incremented by `allocInstance`(A). This is achieved by inserting a ghost assignment immediately after any `new` instruction, as shown below:

```
new A
//set MemUsed = MemUsed+allocInstance(A).
```

8.3 Examples

We illustrate hereafter our approach by several examples.

8.3.1 Inheritance and overridden methods

Overriding methods are treated as follows: whenever a call is performed to a method m , we require that there is enough free memory space for the maximal consumption by all the methods that override or are overridden by m . In Fig. 8.1 we show a class `A` and its extending class `B`, where `B` overrides the method m from class `A`. Method m is invoked by n. Given that the dynamic type of the parameter passed to n is not known, we cannot know which of the two methods will be invoked. This is the reason for requiring enough memory space for the execution of any of these methods.

```
Specification of method
 m in class A:

requires MemUsed + k  <= Max
modifies MemUsed
ensures  MemUsed  <= \old(MemUsed) + k
```

```
Specification for method m in class B:

requires MemUsed + l  <= Max
modifies MemUsed
ensures  MemUsed  <=  \old(MemUsed) + l
```

```
void method n(A a)
...
//{ prove MemUsed <= MemUsed +max(l,k) }
invokevirtual m A
//{ assume MemUsed <= \old{MemUsed} + max(l,k)}
...
```

Figure 8.1: Example of overridden methods

8.3.2 Recursive Methods

In Fig. 8.2 the bytecode of the recursive method `m` and its specification is shown. We show a simplified version of the bytecode; we assume that the constructors for the class `A` and `C` do not allocate memory. Besides the precondition and the postcondition, the specification also includes information about the termination of the method: `variant` **reg**(1), meaning that the local variable **reg**(1) decreases on every recursive call down to and no more than 0, guaranteeing that the execution of the method will terminate.

We explain first the precondition. If the condition of line `1` is not true, the execution continues at line `2`.

In the sequential execution up to line `7`, the program allocates at most `allocInstance`(A) memory units and decrements by 1 the value of **reg**(1). The instruction at line `8` is a recursive call to `m`, which either will take the same branch if **reg**(1) > 0 or will jump to line `12` otherwise, where it allocates at most `allocInstance`(A)+`allocInstance`(C) memory units. On returning from the recursive call one more allocation will be performed at line `9`. Thus `m` will execute, **reg**(1) times, the instructions from lines `4` to `35`, and it finally will execute all the instructions from lines `12` to `16`. The postcondition states that the method will perform no more than **old(reg**(1)**)** recursive calls (i.e., the value of the register variable in the pre-state of the method) and that on every recursive call it allocates no more than two instances of class `A` and that it will finally allocate one instance of class `A` and another of class `C`.

8.3.3 More precise specification

We can be more precise in specifying the precondition of a method by considering what are the field values of an instance, for example. Suppose that we have the method m as shown in Fig. 8.3. We assume that in the constructor of the class `A` no allocations are done. The first line of the method m initializes one of the fields of field `b`. Since nothing guarantees that field `b` is not **null**, the execution may terminate with `NullPointerException`. Depending on the values of the parameters passed to m , the memory allocated will be different. The precondition establishes what is the expected space of free resources depending on if the field `b` is **null**or not. In particular we do not require anything for the free memory space in the case when `b` is **null**. In the normal postcondition we state that the method has allocated an object of class `A`. The exceptional postcondition states that no allocation is performed if `NullpointerException` causes the execution termination.

8.4 Related work

The use of type systems has been a useful tool for guaranteeing that well typed programs run within stated space-bounds. Previous work along these lines defined typed assembly languages, inspired on [80] while others emphasized the use of type systems for functional languages [8, 54, 56].

For instance in [7] the authors present a first-order linearly typed assembly language which allows the safe reuse of heap space for elements of different types. The idea is to design a family of assembly languages which have high-level typing features (e.g. the use of a special *diamond* resource type) which are used to express resource bound constraints. Closely related to the previous-mentioned paper, [105] describes a type theory for certified code, in which type safety guarantees cooperation with a mechanism to limit the CPU usage of untrusted code. Another recent work is [5] where the resource bounds problem is studied in a simple stack machine. The authors show how to perform type, size and termination verifications at the level of the byte-code.

An automatic heap space usage static analysis for first-order functional programs is given in [55]. The analysis both determines the amount of free cells necessary before execution as well as a safe (under)-estimate of the size of a *free-list* after successful execution of a function. These numbers are obtained as solutions to a set of linear programming (LP) constraints derived from the program text. Automatic inference is obtained by using standard polynomial-time algorithms for solving LP constraints. The correctness of the analysis is proved with respect to an operational semantics that explicitly keeps track of the memory structure and the number of free cells.

A logic for reasoning about resource consumption certificates of higher-order functions is defined in [34]. The certificate of a function provides an over-approximation of the execution time of a

```
 requires   MemUsed +
            reg(1)*2*allocInstance(A) +
            allocInstance(A) +
            allocInstance(C) <=  Max

 variant    reg(1)

 ensures    reg(1) >= 0
            &&
            MemUsed <= old(MemUsed)+
            \old(reg(1))*2*allocInstance(A)+
            allocInstance(A) +
            allocInstance(C)

public void m()
//local variable loaded on
//the operand stack of method m
 0 load 1
// if \reg(1) <= 0 go to 12
 1 ifle 12}
 2 new A
// here reg(1) > 0
//@ set MemUsed = MemUsed +  allocInstance(A)
 3 invokespecial A.init
 4 aload 0
 5 iload 1
 6 iconst 1
// reg(1) decremented with 1
 7 isub
//recursive call with the new
//value of reg(1)
 8 invokevirtual D.m
 9 new A
//set MemUsed = MemUsed + allocInstance(A)
 10 invokespecial A.init
 11 goto 16
 target of the jump at  1
 12 new A
//set MemUsed = MemUsed +  allocInstance(A)
 13 invokespecial A.init
 14 new  C
//set MemUsed = MemUsed +  allocInstance(C)
 15 invokespecial C.init
 16 return
```

```
public class D {
  public void m( int i) {
    if (i > 0) {
      new A();
      m(i - 1);
      new A();
    } else {
      new C();
      new A();
   }
  }
}
```

Figure 8.2: Example of a recursive method

call to the function. The logic only defines what is a correct deduction of a certificate and has no inference algorithm associated with it. Although the logic is about computation time the authors claim it could be extended to measure memory consumption.

Another mechanical verification of a byte code language is [29], where a constraint-based algorithm is presented to check the existence of `new` instructions inside intra- and inter-procedural loops. It is completely formalized in Coq and a certified analyzer is obtained using Cog's extraction mechanism. The time complexity of such analysis performs quite good but the auxiliary memory used does not allow it to be on-card. Their analysis is less precise than ours, since they work on an abstraction of the execution traces not considering the number of times a cycle is iterated (there are no annotations). Along these lines, a similar approach has been followed by [100]; no mechanical proof nor implementation is provided in such work.

Other related research direction concerns runtime memory analysis. The work [49] presents a method for analyzing, monitoring and controlling dynamic memory allocation, using pointer and scope analysis. By instrumenting the source code they control memory allocation at run-time. In order to guarantee the desired memory allocation property, in [47] is implemented a runtime monitor to control the execution of a Java Card applet. The applet code is instrumented: a call to a monitor method is added before a `new` instruction. Such monitor method has as parameter the size of the allocation request and it halts the execution of the applet if a predefined allocation bound is exceeded.

A similar results are presented in [31]. The verifier is based on a variant of Dijkstra's weakest precondition calculus using "generalized predicates", which keeps track of the resource units available. Besides adding loop invariants, pre- and post-conditions, the programmer must insert "acquires" annotations to reserve the resource units to be consumed. Our approach has the advantage of treating recursive methods and exceptions, not taken into account in [31]. Another difference with our work is that we operate on the bytecode instead of on the source code.

```
requires  reg(1) !=  null ==>  MemUsed +  allocInstance(A) <= Max
modifies  MemUsed
ensures   MemUsed  <= \old(MemUsed) +  \allocInstance(A) \\
exsures   NullPointerExc  MemUsed == old(MemUsed)

0 load 0
1 getfield C.b
2 load 2
3 putfield B.i
4 new A
//set MemUsed =  MemUsed + allocInstance(A)
5 dup
6 invokespecial A.init
7 store 1
8 return
```

```
public class C{
 B b;
 public void m(A a,int i){
   b.i = i ;
   a = new A();
 }
}
```

Figure 8.3: Example of a method with possible exceptional termination

Chapter 9

A low-footprint Java-to-native compilation scheme using BML

In this chapter, we will focus on the use of our verification scheme in Java native compiler optimizations. Let us first see what is the context and motivations for applying formal program verification in compiler optimization.

Enabling Java on embedded and restrained systems is an important challenge for today's industry and research groups [81]. Java brings features like execution safety and low-footprint program code that make this technology appealing for embedded devices which have obvious memory restrictions, as the success of Java Card witnesses. However, the memory footprint and safety features of Java come at the price of a slower program execution, which can be a problem when the host device already has a limited processing power. As of today, the interest of Java for smart cards is still growing, with next generation operating systems for smart cards that are closer to standard Java systems [64, 52], but runtime performance is still an issue. To improve the runtime performances of Java systems, a common practice is to translate some parts of the program bytecode into native code.

Doing so removes the interpretation layer and improves the execution speed, but also greatly increases the memory footprint of the program: it is expected that native code is about three to four times the size of its Java counterpart, depending on the target architecture. This is explained by the less-compact form of native instructions, but also by the fact that many safety-checks that are implemented by the virtual machine must be reproduced in the native code. For instance, before dereferencing a pointer, the virtual machine checks whether it is `null` and, if it is, throws a `NullExc`. Every time a bytecode that implements such safety-behaviors is compiled into native code, these behaviors must be reproduced as well, leading to an explosion of the code size. Indeed, a large part of the Java bytecode implement these safety mechanisms.

Although the runtime checks are necessary to the safety of the Java virtual machine, they are most of the time used as a protection mechanism against programming errors or malicious code: A runtime exception should be the result of an exceptional, unexpected program behavior and is rarely thrown when executing sane code - doing so is considered poor programming practice. The safety checks are therefore without effect most of the time, and, in the case of native code, uselessly enlarge the code size.

Several studies proposed to factorize these checks or in some case to eliminate them, but none proposed a complete elimination without hazarding the system security. In the following, we use formal proofs to ensure that run-time checks can never be true in a program, which allows us to completely and safely eliminate them from the generated native code. The programs to optimize are JML-annotated against runtime exceptions and verified by the JACK. We have been able to remove almost all of the runtime checks on tested programs, and obtained native ARM thumb code which size was comparable to the original bytecode.

The remainder of this paper is organized as follows. In section 9.1, we overview the methods used for compiling Java bytecode into native code, and evaluate the previous work aiming at optimizing runtime exceptions in the native code. Section 9.2 is a brief presentation of the runtime

exceptions in Java. Then, section 9.3 describes our method for removing runtime exceptions on the basis of formal proofs. We experimentally evaluate this method in section 9.4 and discuss its limitations in 9.5.

9.1 Ahead-of-time & just-in-time compilation

Compiling Java into native code common on embedded devices. This section gives an overview of the different compilation techniques of Java programs, and points out the issue of runtime exceptions.

Ahead-of-Time (AOT) compilation is a common way to improve the efficiency of Java programs. It is related to Just-in-Time (JIT) compilation by the fact that both processes take Java bytecode as input and produce native code that the architecture running the virtual machine can directly execute. AOT and JIT compilation differ by the time at which the compilation occurs. JIT compilation is done, as its name states, just-in-time by the virtual machine, and must therefore be performed within a short period of time which leaves little room for optimizations. The output of JIT compilation is machine-language. On the contrary, AOT compilation compiles the Java bytecode way before the program is run, and links the native code with the virtual machine. In other words, it translates non-native methods into native methods (usually C code) prior to the whole system execution. AOT compilers either compile the Java program entirely, resulting in a 100% native program without a Java interpreter, or can just compile a few important methods. In the latter case, the native code is usually linked with the virtual machine. AOT compilation have no or few time constraints, and can generate optimized code. Moreover, the generated code can take advantage of the C compiler's own optimizations.

JIT compilation in interesting by several points. For instance, there is no prior choice about which methods must be compiled: the virtual machine compiles a method when it appears that doing so is beneficial, e.g. because the method is called often. However, JIT compilation requires embedding a compiler within the virtual machine, which needs resources to work and writable memory to store the compiled methods. Moreover, the compiled methods are present twice in memory: once in bytecode form, and another time in compiled form. While this scheme is efficient for decently-powerful embedded devices such as PDAs, it is inapplicable to very restrained devices like smartcards or sensors. For them, ahead-of-time compilation is usually preferred because it does not require a particular support from the embedded virtual machine outside of the ability to run native methods, and avoids method duplication. AOT compilation has some constraints, too: the compiled methods must be known in advance, and dynamically-loading new native methods is forbidden, or at least very unsafe.

Both JIT and AOT compilers must produce code that exactly mimics the behavior of the Java virtual machine. In particular, the safety checks performed on some bytecode must also be performed in the generated code.

9.2 Java runtime exceptions

The JVM [74] specifies a safe execution environment for Java programs. Contrary to native execution, which does not automatically control the safety of the program's operations, the Java virtual machine ensures that every instruction operates safely. The Java environment may throw predefined runtime exceptions at runtime, like the following ones:

`NullPointerException` This exception is thrown when the program tries to dereference a `null` pointer. Among the instructions that may throw this exceptions are: `getfield`, `putfield`, `invokevirtual`, `invokespecial`, the set of *type*`astore` instructions[1] may throw such an exception.

`ArrayIndexOutOfBoundsException` If an array is accessed out of its bounds, this exception is thrown to prevent the program from accessing an illegal memory location. According to

[1] the JVM instructions are parametrized, thus we denote by *type*`astore` the set of array store instructions, which includes `iastore`, `sastore`, `lastore`, ...

the Java Virtual Machine specification, the instructions of the family *type*`astore` and *type*`aload` may throw such an exception.

`ArithmeticException` This exception is thrown when exceptional arithmetic conditions are met. Actually, there is only one such case that may occur during runtime, namely the division of an integer by zero, which may be done by `idiv`, `irem`, `ldiv` and `lrem`.

`NegativeArraySizeException` Thrown when trying to allocate an array of negative size. `newarray`, `anewarray` and `multianewarray` may throw such an exception.

`ArrayStoreException` Thrown when an object is attempted to be stored into an array of incompatible type. This exception may be thrown by the `aastore` instruction.

`ClassCastException` Thrown when attempting to cast an object to an incompatible type. The `checkcast` instruction may throw such an exception.

`IllegalMonitorStateException` Thrown when the current thread is not the owner of a released monitor, typically by `monitorexit`.

If the JVM detects that executing the next instruction will result in an inconsistency or an illegal memory access, it throws a runtime exception, that may be caught by the current method or by other methods on the current stack. If the exception is not caught, the virtual machine exits. This safe execution mode implies that many checks are made during runtime to detect potential inconsistencies. For instance, the `aastore` bytecode, which stores an object reference into an array, may throw three different exceptions:

- `NullPointerException`, if the reference to the array is `null`,
- `ArrayIndexOutOfBoundsException`, if the index in which to store the object is not within the bounds of the array,
- `ArrayStoreException`, if the object to store is not assignment-compatible with the array (for instance, storing an `Integer` into an array of `Boolean`).

Of the 202 bytecodes defined by the Java virtual machine specification, we noticed that 43 require at least one runtime exception check before being executed. While these checks are implicitly performed by the bytecode interpreter in the case of interpreted code, they must explicitly be issued every time such a bytecode is compiled into native code, which leads to a code size explosion. Ishizaki et al. measured that bytecodes requiring runtime checks are frequent in Java programs: for instance, the natively-compiled version of the SPECjvm98 `compress` benchmark has 2964 exception check sites for a size of 23598 bytes. As for the `mpegaudio` benchmark, it weights 38204 bytes and includes 6838 exception sites [59]. The exception check sites therefore make a non-neglectable part of the compiled code.

Figure 9.1 shows an example of Java bytecode that requires a runtime check to be issued when being compiled into native code.

It is, however, possible to eliminate these checks from the native code if the execution context of the bytecode shows that the exceptional case never happens. In the program of figure 9.1, the lines 2 and 3 could have been omitted if we were sure that for all possible program paths, `j` can never be equal to zero at this point. This allows to generate less code and thus to save memory. Removing exception check sites is a topic that has largely been studied in the domain of JIT and AOT compilation.

9.3 Optimizing ahead-of-time compiled Java code

For verifying the bytecode that will be compiled into native code, we use the JACK verification framework. In particular, we use the compiler from JML to BML and the bytecode verification condition generator.

Verifying that a bytecode program does not throw Runtime exceptions using JACK involves several stages:

Java version:

```
iload i
iload j
idiv
ireturn
```

C version:

```
1 int i, j;
2 if (j == 0)
3   THROW(ArithmeticException);
4 RETURN_INT(i / j);
```

Figure 9.1: A Java bytecode program and its (simplified) C-compiled version. The behavior of the division operator in Java must be entirely reproduced by the C program, which leads to the generation of a runtime exception check site

1. writing the JML specification at the source level of the application, which expresses that no runtime exceptions are thrown.
2. compiling the Java sources into bytecode
3. compiler the JML specification into BML specification and add it in the class file
4. generating the verification conditions over the bytecode and its BML specification, and proving the verification conditions 9.3.2. During the calculation process of the verification conditions, they are indexed with the index of the instruction in the bytecode array they refer to and the type of specification they prove (e.g. that the proof obligation refers to the exceptional postcondition in case an exception of type `Exc` is thrown when executing the instruction at index `i` in the array of bytecode instructions of a given method). Once the verifications are proved, information about which instructions can be compiled without runtime checks is inserted in user defined attributes of the class file.
5. using these class file attributes in order to optimize the generated native code. When a bytecode that has one or more runtime checks in its semantics is being compiled, the bytecode attribute is checked in order to make sure that the checks are necessary. It indicates that the exceptional condition has been proved to never happen, then the runtime check is not generated.

Our approach benefits from the accurateness of the JML specification and from the bytecode verification condition generator. Performing the verification over the bytecode allows to easily establish a relationship between the proof obligations generated over the bytecode and the bytecode instructions to optimized.

In the rest of this section, we explain in detail all the stages of the optimization procedure.

9.3.1 Methodology for writing specification against runtime exception

We now illustrate with an example what are the JML annotations needed for verifying that a method does not throw a runtime exception. Figure 9.2[2] shows a Java method annotated with a JML specification. The method `clear` declared in class `Code_Table` receives an integer parameter `size` and assigns `0` to all the elements in the array field `tab` whose indexes are smaller than the value of the parameter `size`. The specification of the method guarantees that if every caller respects the method precondition and if every execution of the method guarantees its postcondition then the method `clear` never throws an exception of type or subtype `java.lang.Exception`[3]. This is expressed by the class and method specification contracts. First, a class invariant is declared which states that once an instance of type `Code_Table` is created, its array field `tab` is not null. The class invariant guarantees that no method will throw a `NullExc` when dereferencing (directly or indirectly) `tab`.

[2]although the analysis that we describe is on bytecode level, for the sake of readability, the examples are also given on source level

[3]Note that every Java runtime exception is a subclass of `java.lang.Exception`

```
final class Code_Table {
  private/*@spec_public */short tab[];

  //@invariant tab != null;

  ...

  //@requires size <= tab.length;
  //@ensures true;
  //@exsures (Exception) false;
  public void clear(int size) {
  1  int code;
  2  //@loop_modifies code, tab[*];
  3  //@loop_invariant code <= size && code >= 0;
  4  for (code = 0; code < size; code++) {
  5    tab[code] = 0;
     }
  }
}
```

Figure 9.2: A JML-annotated method

The method precondition requires the `size` parameter to be smaller than the length of `tab`. The normal postcondition, introduced by the keyword `ensures`, basically says that the method will always terminate normally, by declaring that the set of final states in case of normal termination includes all the possible final states, i.e. that the predicate `true` holds after the method's normal execution[4]. On the other hand, the exceptional postcondition for the exception `java.lang.Exception` says that the method will not throw any exception of type `java.lang.Exception` (which includes all runtime exceptions). This is done by declaring that the set of final states in the exceptional termination case is empty, i.e. the predicate `false` holds if an exception caused the termination of the method. The loop invariant says that the array accesses are between index `0` and index `size - 1` of the array `tab`, which guarantees that no loop iteration will cause a `ArrayIndexOutOfBoundsException` since the precondition requires that `size <= tab.length`.

9.3.2 From program proofs to program optimizations

In this phase, the bytecode instructions that can safely be executed without runtime checks are identified. Depending on the complexity of the verification conditions, Jack can discharge them to the fully automatic prover Simplify, or to the Coq and AtelierB interactive theorem prover assistants.

There are several conditions to be met for a bytecode instruction to be optimized safely – the precondition of the method the instruction belongs to must hold every time the method is invoked, and the verification condition related to the exceptional termination must also hold. In order to give a flavor of the verification conditions we deal with, figure 9.3 shows part of the verification condition related to the possible `ArrayIndexOutOfBounds` exceptional termination of instruction `11 sastore` in figure 4, which is actually provable.

Once identified, proved instructions can be marked in user-defined attributes of the class file so that the compiler can find them.

[4]Actually, after terminating execution the method guarantees that the first `size` elements of the array tab will be equal to 0, but as this information is not relevant to proving that the method will not throw runtime exceptions we omit it

$$
\begin{array}{ll}
\ldots & \\
\texttt{length}(\texttt{tab}(\mathbf{reg}(0)) \leq \mathbf{reg}(2)_{15} \vee \mathbf{reg}(2)_{15} < 0 & \\
\wedge & \\
\mathbf{reg}(2)_{15} \geq 0 & \Rightarrow \mathbf{false} \\
\wedge & \\
\mathbf{reg}(2)_{15} < \mathbf{reg}(1) & \\
\wedge & \\
\mathbf{reg}(1) \leq \texttt{length}(\texttt{tab}(\mathbf{reg}(0))) &
\end{array}
$$

Figure 9.3: The verification condition for the ArrayIndexOutOfBoundException check related to the sastore instruction of figure 4

9.4 Experimental results

This section presents an application and evaluation of our method on various Java programs.

9.4.1 Methodology

We have measured the efficiency of our method on two kinds of programs, that implement features commonly met in restrained and embedded devices. **crypt** and **banking** are two smartcard-range applications. **crypt** is a cryptography benchmark from the Java Grande benchmarks suite, and **banking** is a little banking application with full JML annotations used in [28]. **scheduler** and **tcpip** are two embeddable system components written in Java, which are actually used in the JITS [1] platform. **scheduler** implements a threads scheduling mechanism, where scheduling policies are Java classes. **tcpip** is a TCP/IP stack entirely written in Java, that implements the TCP, UDP, IP, SLIP and ICMP protocols. These two components are written with low-footprint in mind ; however, the overall system performance would greatly benefit from having them available in native form, provided the memory footprint cost is not too important.

For every program, we have followed the methodology described in section 9.3 in order to prove that runtime exceptions are not thrown in these programs. We look at both the number of runtime exception check sites that we are able to remove from the native code, and the impact on the memory footprint of the natively-compiled methods with respect to the unoptimized native version and the original bytecode. The memory footprint measurements were obtained by compiling the C source file generated by the JITS AOT compiler using GCC 4.0.0 with optimization option `-Os`, for the ARM platform in thumb mode. The native methods sizes are obtained by inspecting the .o file with `nm`, and getting the size for the symbol corresponding to the native method.

Regarding the number of eliminated exception check sites, we also compare our results with the ones obtained using the JC virtual machine mentioned in 9.6, version 1.4.6. The results were obtained by running the `jcgen` program on the benchmark classes, and counting the number of explicit exception check sites in the generated C code. We are not comparing the memory footprints obtained with the JITS and JC AOT compilers, for this result would not be pertinent. Indeed, JC and JITS have very different ways to generate native code. JITS targets low memory footprint, and JC runtime performance. As a consequence, a runtime exception check site in JC is heavier than one in JITS, which would falsify the experiments. Suffices to say that our approach could be applied on any AOT compiler, and that the most relevant measurement is the number of runtime exception check sites that remains in the final binary - our measurements on the native code memory footprint are just here to evaluate the size impact of exception check sites.

9.4.2 Results

Table 9.1 shows the results obtained on the four tested programs. The three first columns indicate the number of check sites present in the bytecode, the number of explicit check sites emitted by JC, and the number of check sites that we were unable to prove useless and that must be present in our

Table 9.1: Number of exception check sites and memory footprints when compiled for ARM thumb

Program	# of exception check sites			Memory footprint (bytes)		
	Bytecode	JC	Proven AOT	Bytecode	Naive AOT	Proven AOT
crypt	190	79	1	1256	5330	1592
banking	170	12	0	2320	5634	3582
scheduler	215	25	0	2208	5416	2504
tcpip	1893	288	0	15497	41540	18064

Table 9.2: Human work on the tested programs

Program	Source code size (bytes)		Proved lemmas	
	Code	JML	Automatically	Manually
crypt	4113	1882	227	77
banking	11845	15775	379	159
scheduler	12539	3399	226	49
tcpip	83017	15379	2233	2191

optimized AOT code. The last columns give the memory footprints of the bytecode, unoptimized native code, and native code from which all proved exception check sites are removed.

On all the tested programs, we were able to prove that all but one exception check site could be removed. The only site that we were unable to prove from **crypt** is linked to a division, which divisor is a computed value that we were unable to prove not equal to zero. JC has to retain 16% of all the exception check sites, with a particular mention for **crypt**, which is mainly made of array accessed and had more remaining check sites.

The memory footprints obtained clearly show the heavy overhead induced by exception check sites. Despite of the fact that the exception throwing convention has deliberately been simplified for our experiments, optimized native code is less than half the size of the non-optimized native code. The native code of **crypt**, which heavily uses arrays, is actually made of exception checking code at 70%.

Comparing the size of the optimized native versions with the bytecode reveals that proved native code is just slightly bigger than bytecode. The native code of **crypt** is 27% bigger than its bytecode version. Native **scheduler** only weights 13.5% more that its bytecode, **tcpip** 16.5%, while **banking** is 54% heavier. This last result is explained by the fact that, being an application and not a system componant, **banking** includes many native-to-java method invokations for calling system services. The native-to-java calling convention is costly in JITS, which artificially increases the result.

Finally, table 9.2 details the human work required to obtain the proofs on the benchmark programs, by comparing the amount of JML code with respect to the comments-free source code of the programs. It also details how many lemmas had to be manually proved.

On the three programs that are annotated for the unique purpose of our study, the JML overhead is about 30% of the code size. The **banking** program was annotated in order to prove other properties, and because of this is made of more JML annotations than actual code. Most of the lemmas could be proved by Simplify, but a non-neglectable part needed human-assistance with Coq. The most demanding application was the TCP/IP stack. Because of its complexity, nearly half of the lemmas could not be proved automatically.

The gain in terms of memory footprint obtained using our approach is therefore real. One may also wonder whether the runtime performance of such optimized methods would be increased. We did the measurements, and only noticed a very slight, almost undetectable, improvement of the execution speed of the programs. This is explained by the fact that the exception check sites conditions are always false when evaluated, and therefore the amount of supplementary code executed is very low. The bodies of the proved runtime exception check sites are, actually, dead

code that is never executed.

9.5 Limitations

Our approach suffers from some limitations and usage restrictions, regarding its application on multi-threaded programs and in combination with dynamic code loading.

9.5.1 Multi-threaded programs

As we said in section 9.3, JACK only supports the sequential subset of Java. Because of this, we are unable to prove check sites related to monitor state checking, that typically throws an `IllegalMonitorStateException`. However, they can be simplified if it is known that the system will never run more than one thread simultaneously. It should be noted, that Java Card does not make use of multi-threading and thus doesn't suffer from this limitation.

9.5.2 Dynamic code loading

Our removal of runtime exception check sites is based on the assumption that a method's preconditions are always respected at all its call sites. For closed systems, it is easy to verify this property, but in the case of open systems which may load and execute any kind of code, the property could not always be ensured. In the case where the set of applications that will run on the system is not statically known, our approach could not be safely applied on public methods since dynamically-loaded code may call them without respecting their preconditions. However, a solution is to verify the methods of every dynamically loaded class before it is loaded w.r.t. the specification of the classes already installed classes and their methods.

9.6 Related work

Toba [92] is a Java-to-C compiler that transforms a whole Java program into a native one. Harissa [82] is a Java environment that includes a Java-to-C compiler as well as a virtual machine, and therefore supports mixed execution. While both environments implement some optimizations, they are not able to detect and remove unused runtime checks during ahead-of-time compilation. The JC Virtual Machine [2] is a Java virtual machine implementations that converts class files into C code using the Soot [104] framework, and runs their compiled version. It supports redundant exceptions checks removal, and is tuned for runtime performance, by using operating system signals in order to detect exceptional conditions like null pointer dereferencing. This allows to automatically remove most of the `NullPointerException`-related checks.

In [58] and [10], Hummel et al. use a Java compiler that annotates bytecodes with higher-level information known during compile-time in order to improve the efficiency of generated native code. [59] proposes methods for optimizing exceptions handling in the case of JIT compiled native code. These works rely on knowledge that can be statically inferred either by the Java compiler or by the JIT compiler. In doing so, they manage to efficiently factorize runtime checks, or in some cases to remove them. However, they are still limited to the context of the compiled method, and do not take the whole program into account. Indeed, knowing properties about a the parameters of a method can help removing further checks.

We propose to go further than these approaches, by giving more precise directives as to how the program behaves in the form of JML annotations. These annotations are then used to get formal behavioral proofs of the program, which guarantee that runtime checks can safely be eliminated for ahead-of-time compilation.

Chapter 10

Conclusion

10.1 Results

We have presented an infrastructure for verification of Java bytecode programs which allows to reason about potentially sophisticated functional and security properties and which benefits from verification over Java source programs. We have also introduced the bytecode specification language BML tailored to Java bytecode, a compiler from the Java source specification language JML to BML and a verification condition generator for Java bytecode programs. We have shown that the verification procedure is correct w.r.t. a big step operational semantics of Java bytecode programs for a particular subset of BML. We also show that the verification procedure for Java like programs and Java like bytecode are syntactically equivalent (modulo names and types).

Currently, from Fig.1.5 presented in the introductory Chapter 1.2, we have developed the following components

- prototype of a verification condition generator based on the weakest precondition calculus presented in this thesis.
- a compiler from the corresponding subset of JML to BML and an encoding of BML in the class file format

These two components have been integrated in the JACK [28] verification framework developed and supported by our research team Everest at INRIA Sophia Antipolis which has been initially designed for the verification of Java source programs annotated with JML specification.

We would like to give a brief description of the implementation of the verification condition generator. The bytecode verification condition generator works as follows. For the verification of a class file containing BML specification, it will generate verification conditions for every method of this class including the constructors. For generating the verification conditions concerning a method implementation, first the control flow graph corresponding to the bytecode instruction is built. The latter is transformed into an acyclic control flow graph where the back-edges are removed. Then the verification procedure proceeds by generating over every execution path in the control flow graph its corresponding verification conditions. For every path which terminates by throwing an uncaught exception, the postcondition is the specified exceptional postcondition for this case. For the paths which terminate normally, the normal postcondition is taken. For every path which terminates with an instruction which is dominated by a loop entry and whose direct successor is the same loop entry, the postcondition is the corresponding loop invariant.

The bytecode verification in Jack uses the intermediate language for the verification conditions and thus, bytecode verification conditions can be translated to several different theorem provers - Simplify [37] which is an automatic decision procedure, the Atelier B and the Coq interactive theorem prover assistants.

The bytecode verification condition generator benefits also from the original user friendly interface of the JACK tool. In particular, the user can see the verification conditions in his favorite language - Java, Simplify, Coq or B. The lemmas are classified to what part of the annotation they refer to, as for instance, a lemma which refers to the establishment of the postcondition, or

the preservation of the loop invariant. The hypothesis in the lemma also hold the index of the instruction from which they originate. We have used the prototype of the bytecode verification condition generator for the case studies presented in Chapter 9.

10.2 Future work

In the following, we identify the directions for extending the work presented in this thesis

10.2.1 Verification condition generator

A first direction for future work concerning the verification condition generator is its extension. Currently, it works only for the sequential fragment of Java. But realistic applications rely often on multi - threading which is difficult to verify against a functional specifications or security policies. One of the important aspects of the correctness of multi - threaded programs is the absence of deadlocks, and race conditions. Such properties can be ensured by type systems [42, 43] or static verification based on program logic [44]. The earliest works in the field of parallel program verification are the Owicki and Gries approach [89] and the rely - guarantee approach. However, the first approach is not modular and requires a large amount of verification conditions while for the second, the annotation procedure can not be automatized.

Extending our verification scheme for bytecode will certainly be based on a more recent work where one of the basic concerns is to establish method atomicity [46]. The notion of a statement atomicity states that however a statement is interleaved with other parallel programs, the result of its execution will not change. The atomicity can be detected via static checking [46] using type systems. Thus, the program verification process is separated in two parts - first checking for program atomicity [46] are done and then verifying the functional correctness using methodologies for sequential programs as Hoare style reasoning. In this last approach in the case of Java, the basic concern is to establish the atomicity of method bodies, i.e. method execution does not depend on the possible interleaving with threads. Recently, E.Rodriguez and al. in [97] proposed an extension for JML for multi threaded programs. Their proposal introduces new specification keywords which allow to express that a variable is locked or that a method is atomic.

Next point which can be interesting is to provide a machine checked proof for the soundness result. This can be especially interesting for a PCC where such a proof will be necessary for the client site. Currently, the soundness statement is expressed in terms of method contracts. Moreover, here we have assumed the soundness of the frame conditions. But an extension of the soundness theorem w.r.t. to assertions which must hold at particular program points may be useful for expressing for instance, safety policies which concern intermediate program states.

10.2.2 Property coverage for the specification language

Another direction which may be pursued as a future work of the thesis is the extension of the expressiveness of the specification language BML. So far, BML supports method contracts - method pre and post conditions, frame conditions, intermediate annotations as for instance loop invariants, class specifications as well as special specification operators. These are very useful aspects which allow for dealing with complex properties and gives a semantics on bytecode level to a relatively small subset of the high specification language JML which corresponds to JML Level 0 [1]. But it is certainly of interest to support more features of JML in BML as this will turn the latter language richer. However, the meaning of JML constructs (at least from our experience up to now) is the same as the meaning of their corresponding part in BML.

An important example is the JML construct for pure methods which has been identified as a challenge in the position paper [65]. These methods does not modify the program state and thus, pure methods can be used in specifications (only side effect free expressions may occur in expressions). This gives more expressive specifications as with them, for instance, specification can talk about the result of method invocation or use pure methods as a predicate relating their initial and final state. Moreover, a methodology which allows for using methods that may have

[1] http://www.cs.iastate.edu/ leavens/JML/jmlrefman/jmlrefman_2.html#SEC19

unobservable side effects is also of interest [84]. Formalizing and establishing the meaning of pure methods is difficult and a literature exists for this problem [35]. As we said above, the treatment of pure methods is the same on source and bytecode.

Also, support for specification constructions for alias control is certainly useful especially because it allows for a modular verification of class invariants and frame conditions. The alias control is guaranteed through ownership type systems which check that only an owner of a reference can modify its contents. This can considerably improve the current implementation for the verification of object invariants [38]. In particular, our way of proving object invariants is non modular - at every method call the invariants of all visible objects must be valid and they are assumed to hold when the call is terminated; similarly, when a method body is verified in its precondition the invariants of all visible objects are assumed to hold and at the end of the method body all these invariants must be established. In practice, it is very difficult to verify that all the invariants for all visible objects in a method hold. In order to keep the number of the verification conditions reasonable, we check the invariants only for the current object this and the objects received as parameters which is not sound.

10.2.3 Preservation of verification conditions

So far, we have shown that non-optimizing Java compilation preserves the form of the verification conditions on source and bytecode. We identify two basic directions for future work:

Source and non optimized bytecode verification conditions equivalent modulo We have experimented with the verification conditions on source and bytecode in JACK and saw that in practice they are almost equivalent syntactically. From one part, there are the difference in the types supported on bytecode and source level. For instance, the JVM does not provide support for boolean type values which are basically encoded as integer values. The same is true for byte and short values. Another difference is the identifiers for variables and fields. For instance, in Java names for fields, method local variables and parameters are their identifiers which are given by the program developer. On bytecode method local variables and parameters are encoded as elements of the method register table and field names are encoded as numbers of the constant pool table of the class. A simple but useful extension to the prototype for bytecode verification is a compiler from source proof obligations to bytecode proof obligations which overcomes those differences. This can be considered also as a step towards the building a PCC architecture where the certificate generation benefits from the source level verification and thus allows for treating sophisticated security policies.

Relation between verification conditions on Java source and optimized Java bytecode The equivalence between verification conditions on source and the corresponding non optimized bytecode is important as it allows that bytecode programs benefit from source verification. In particular, it makes feasible Proof Carrying Code for sophisticated client requirements. However, a step further in this direction is to investigate the relation between source programs and their bytecode counterpart produced by an optimizing compiler. This is interesting for the following reasons. It is a fact that interpretation of bytecode on the JVM is slower than execution by its corresponding assembly code. In order to speed up the execution time for a Java bytecode program, one might use a just-in-time (JIT for short) compilation which translates on the fly the bytecode into the machine specific language. However, JIT compilation can potentially slow the execution exactly because it does compilation on the fly. Another possibility is to perform optimizations on the bytecode. Currently, most of the Java compilers do not support much optimizations. However, there do already exist Java optimizing compilers, for instance the Soot optimization framework[2] and most probably the number of the Java optimizing compilers will increase with the evolution of the Java language. This also means that verification of optimized programs will also become an issue. A first step in this direction is the work of C. Kunz et al.[17] who give an algorithm for translating certificates and annotations over a non optimized program into a certificate and annotation for its optimized version. Their work addresses optimizations like constant propagation, loop induction and dead register elimination for a simple language.

[2] http://www.sable.mcgill.ca/soot/

10.2.4 Towards a PCC architecture

The bytecode verification condition generator and the BML compiler is the first step towards a PCC framework. The missing part is the certificate format which comes along with the bytecode and which is the evidence for that the bytecode respects the client requirements. Defining an encoding of the certificate should take into account several factors:

- certificate size must be reasonably small. This is important, for instance, if the certified program comes over a network with a limited bandwidth
- certificates must be easily checked. This means that the certificate checker is small and simple. Of course, the code consumer might not want to spend all of its computation resources for checking that the certificate guarantees the program conformance to its policies.

Note that the certificate size and its checking complexity are dual: the bigger the certificate is more manageable is the checking process and vice versa. The problem becomes even more difficult if the certificate must be checked on the device because of the computational and space constraints.

Another perspective in this direction is the encoding of type systems into the bytecode logic. Type systems provide a high level of automation. Their encoding in the logic can be useful as the certificate can be generated automatically and thus, avoids the user interaction. However, type systems are conservative in the sense that they tend to reject a large amount of correct programs. A possible solution to this problem are hybrid certificates which combine both type systems and program logic. In this approach, the unknown code comes supplied with a derivation in the logic generated potentially with the help of user interaction for the parts of the code which can not be inferred by the type system. The client side then applies a type inference procedure over the unknown code and once it gets to the place in the parts of the code where the type inference does not work but for which there is a derivation in the certificate, he will type check that derivation. This is actually an approach which will be adopted in the Mobius project.

The objective of this thesis was to build a bytecode verification framework for dealing with potentially sophisticated security and functional policies. A further objective, pursued in the European project Mobius (short for Ubiquity, Mobility and Security) is to build basis for guaranteeing security and trust in program application in the presence of mobile and ubiquitous computing. We hope that we have convinced the reader for the importance of such techniques and in particular of the evolution from source verification to low level verification and the necessity of an interactive verification process for building evidence for the security of unknown applications.

Appendix A

Encoding of BML in the class file format

A.1 Class annotation

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

A.1.1 Ghost variables

Ghost_Field_attribute {

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

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

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

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

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

descriptor_index
The value of the descriptor_index item must be a valid index into the **constant_pool** table. The

constant_pool entry at that index must be a CONSTANT_Utf8 structure which must represent a valid Java field descriptor.

A.1.2 Class invariant

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

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

attribute_length
the length of the attribute in bytes - 6.

attribute_formula
code of the formula that represents the invariant

A.1.3 History Constraints

JMLHistoryConstraints_attribute {
u2 attribute_name_index;
u4 attribute_length;
formula attribute_formula;
}

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

attribute_length
the length of the attribute in bytes - 6.

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

A.2 Method annotation

A.2.1 Method specification

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

JMLMethod_attribute {
u2 attribute_name_index;
u4 attribute_length;
formula requires_formula;
u2 spec_count;
{ formula spec_requires_formula;

```
        u2 modifies_count;
        formula modifies[modifies_count];
        formula ensures_formula;
        u2 exsures_count;
        {  u2 exception_index;
           formula exsures_formula;
        } exsures[exsures_count];
    } spec[spec_count];
}
```

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

attribute_length
The length of the attribute in bytes.

requires_formula
The formula that represents the precondition

spec_count
The number of specification case.

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

spec_requires_formula
The formula that represents the precondition

modifies_count
The number of modified variable.

modifies[]
The array of modified formula.

ensures_formula
The formula that represents the postcondition

exsures_count
The number of exsures clause.

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

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

exsures_formula
The formula that represents the exceptional postcondition

A.2.2 Set

These are particular assertions that assign to ghost fields.

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

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

attribute_length
The length of the attribute in bytes.

set_count
The number of set statement.

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

index
The index in the bytecode where the assignment to the ghost field is done.

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

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

A.2.3 Assert

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

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

attribute_length
The length of the attribute in bytes.

assert_count
The number of assert statement.

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

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

predicate
the predicate that must hold at index **index** in the bytecode

A.2.4 Loop specification

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

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

attribute_length
The length of the attribute in bytes

loop_count
The length of the array of loop specifications

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

modifies_count
The number of modified variable.

modifies[]
The array of modified expressions.

invariant
The predicate that is the loop invariant. It is a formula written in the grammar specified in the section Formula

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

A.3 Codes for BML expressions and formulas

Formulas P		
Code	Symbol	Grammar
0x00	*true*	
0x01	*false*	
0x02	$\wedge$	$P\ P$
0x03	$\vee$	$P\ P$
0x04	$\Rightarrow$	$P\ P$
0x05	$\neg$	P
0x06	$\forall$	$bv\ P$
0x07	$\exists$	$bv\ P$
0x10	$=$	$E\ E$
0x11	$>$	$E\ E$
0x12	$<$	$E\ E$
0x13	$\leq$	$E\ E$
0x14	$\geq$	$E\ E$
0x16	$<:$	**type**(*ident*) **type**(*ident*)
0x17	$\neq$	$E\ E$

Expressions E		
Code	Symbol	Grammar
0x20	$+$	$E\ E$
0x21	$-$	$E\ E$
0x22	$*$	$E\ E$
0x23	$/$	$E\ E$
0x24	%	$E\ E$
0x25	$-$	E
0x40	`int constant`	i
0x50	**type**	*ident*
0x51	**elemtype**	E
0x52	**result**	
0x53	**typeof**	E
0x54	**TYPE**	
0x55	**old**	
0x61	**arrAccess**(,)	$E\ E$
0x63	**.**	$E\ E$
0x70	**this**	
0x80	**null**	
0x90	*FieldConstRef*	*ident*
0xA0	Reg	*digits*
0xD0	**EXC**	
0xE0	*bv*	*intLiteral*

Modifies *modLocation*		
Code	Symbol	Grammar
0xD2	**nothing**	
0xD3	**everything**	
0xD4	*arrayModAt*(,)	E *specIndex*
0xD5	**.**	E *FieldConstRef*
0xD6	Reg	*ident*

Index of modified array elements *specIndex*		
Code	Symbol	Grammar
0xE1	all	
0xE2	...	$E\ E$
0xE3	E	

Appendix B

Proofs of properties from Section 7.1.2

Lemma 7.1.2.1. *For any statement or expression $\mathcal{SE}$ which does not terminate on* `return` *or* `athrow`*, start label s and end label e, the compiler will produce a list of bytecode instruction $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$ such that instruction $e+1$ may execute after e, i.e. $e \longrightarrow e+1$*

Proof: The proof is by structural induction over the compiled statement. We sketch the case for compositional statement, the other cases being similar. Remind that by compiler definition, we get that the compilation of $S_1; S_2$ starting at index s is $\ulcorner s, S_1; S_2, e \urcorner_{\mathtt{m}} = \ulcorner s, S_1, e' \urcorner_{\mathtt{m}}; \ulcorner e'+1, S_2, e \urcorner_{\mathtt{m}}$. By induction hypothesis, we get that the lemma holds for $\ulcorner e'+1, S_2, e \urcorner_{\mathtt{m}}$ and we get that $e \longrightarrow e+1$ which means that this case holds. *Qed.*

In the next, we will need several auxiliary lemmas that will allow us to prove the statements from Section 7.1.2. The next lemma states that all the jump instructions in the compilation of a statement target instruction which are also in the compilation of the statement. For illustration, we may return back to the example Fig. 7.6 on page 101 and focus on the compilation of the **if** statement **if** (i >= 0) **then** {v = i} **else** {v = −i} which comprises instructions from 4 to 12. Note that sequential instructions have as successor the next instruction. Thus, sequential instructions respect this condition. Only jump instructions may cause control transfer outside the **if** compilation. We notice that the compilation contains two jump instructions. The first is the instruction 5 **ifge** 10 which jumps inside the compilation of the if statement and the instruction 9 **goto** 12 which jumps also inside. Thus, the compilation of the **if** statement respects the property.

Lemma B.1 (Jumps in statement compilation target instructions inside the statement compilation). *For any statement or expression $\mathcal{SE}$, the compiler will produce a list of bytecode instruction $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$ such that every jump instruction(*`goto` *or* `if_`*cond) which is in the compilation does not transfer the control outside the region of the compilation $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$.*

Proof: The proof is done by contradiction and uses structural induction. We sketch the proof for the compilation of the if statement, the rest of the cases being similar.
Suppose that this is not true. Recall that the compilation of a conditional statement starting at index s (Fig. 7.2 on page 98) is of the form:

$$\ulcorner s, \texttt{if}\ (E_1\, cond\, E_2)\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\}, e \urcorner_{\mathtt{m}} = \begin{array}{l} \ulcorner s, E_1, e' \urcorner_{\mathtt{m}} \\ \ulcorner e'+1, E_2, e'' \urcorner_{\mathtt{m}} \\ e''+1 : \texttt{if\_}cond\ e'''+2; \\ \ulcorner e''+2, S_2, e''' \urcorner_{\mathtt{m}} \\ e'''+1 : \texttt{goto}\ e \\ \ulcorner e'''+2, S_1, e-1 \urcorner_{\mathtt{m}}; \\ e : \texttt{nop} \end{array}$$

Because by induction hypothesis every jump in $\ulcorner e'''+2, S_1, e \urcorner_{\mathtt{m}}$ targets instructions inside $\ulcorner e'''+2, S_1, e \urcorner_{\mathtt{m}}$, it is not possible that there be a jump inside $\ulcorner e'''+2, S_1, e \urcorner_{\mathtt{m}}$ which targets outside the compilation of the conditional statement. Similarly, we conclude that such a jump cannot be contained in $\ulcorner e'''+2, S_1, e-1 \urcorner_{\mathtt{m}}$, $\ulcorner s, E_1, e' \urcorner_{\mathtt{m}}$ nor $\ulcorner e'+1, E_2, e'' \urcorner_{\mathtt{m}}$ The only cases that

remain possible are the jumps $e''+1$: `if_`*cond* $e'''+2$ and $e'''+1$: `goto` e. But for both instructions that is not true. Thus, the lemma holds for this case.

Qed.

The next property of the compiler is that any statement or expression is compiled in a list of bytecode instructions such that instructions inside the compilation of a statement or expression cannot be targeted by instructions which are outside the statement compilation except for the first instruction in the compilation. For instance, the instructions 15-30 in the compilation of the while statement in Fig. 7.6 on page 101 can be reached from outside of the statement compilation only by passing through the instruction at index 15.

Lemma B.2 (Compilation of statements and expressions cannot be jumped from outside inside). *For all statements and expressions $\mathcal{SE}'$ and $\mathcal{SE}$, such that $\mathcal{SE}'$ is a substatement of $\mathcal{SE}$ ($\mathcal{SE}[\mathcal{SE}']$) and their compilations are $\ulcorner s, \mathcal{SE}, e\urcorner_{\mathtt{m}}$ and $\ulcorner s', \mathcal{SE}', e'\urcorner_{\mathtt{m}}$. Let us have instruction at index j in the compilation of $\mathcal{SE}$ ($j \in \ulcorner s, \mathcal{SE}, e\urcorner_{\mathtt{m}}$) but which is not in the compilation of $\ulcorner s', \mathcal{SE}', e'\urcorner_{\mathtt{m}}$ ($\neg(j \in \ulcorner s', \mathcal{SE}', e'\urcorner_{\mathtt{m}})$). Let us also have instruction k in the compilation $\ulcorner s', \mathcal{SE}', e'\urcorner_{\mathtt{m}}$. Suppose that j may execute after k ($j \longrightarrow k$). It then follows that k is the first instruction in the compilation of $\mathcal{SE}'$*

*Proof:*The proof is by induction over the structure of statements and expressions. We sketch here the case for compositional statement, the other cases are similar and use previous Lemma B.1 and Lemma 7.1.2.1 The compositional statement $S_1; S_2$ has the compilation $\ulcorner s, S_1; S_2, e\urcorner_{\mathtt{m}}$ which by the compiler definition is $\ulcorner s, S_1, e'\urcorner_{\mathtt{m}}\ulcorner e'+1, S_2, e\urcorner_{\mathtt{m}}$. By induction hypothesis the lemma holds both for $\ulcorner s, S_1, e'\urcorner_{\mathtt{m}}$ and $\ulcorner e'+1, S_2, e\urcorner_{\mathtt{m}}$. It is also necessary to show that there are no jumps from the compilation of S_1 into the compilation of S_2 and vice versa. Both directions follow from Lemma B.1 that all jumps in a statement compilation are inside the statement. Moreover, from Lemma 7.1.2.1 we have that $e' \longrightarrow e'+1$ which conforms to the statement. Thus, the case for compositional statement holds.

Qed.

Lemma 7.1.2.2 (Compilation of expressions). *For any expression E, starting label s and end label e, the compilation $\ulcorner s, E, e\urcorner_{\mathtt{m}}$ is a block of bytecode instruction in the sense of Def. 7.1.2.1*

Proof:

Following the Def. 7.1.2.1 of block of bytecode instructions, we have to see if the compilation of an expression respects three conditions. The first condition of Def.7.1.2.1 states that none of the instructions is a target of an instruction outside of the compilation of the expression except from the first instruction. This follows from lemma B.2. The second condition in Def. 7.1.2.1 requires that there are no control transfer instructions (jumps, `return`and `athrow`) in the list of instructions representing the compilation of an expression, i.e. that every instruction in the compilation of an expression is in execution relation with the next instruction. This is established by induction over the structure of the expression. The third condition in Def. 7.1.2.1 states that the compilation $\ulcorner s, E, e\urcorner_{\mathtt{m}}$ is such that no instruction except possibly for the first instruction in the expression compilation is in a loop execution relation with its predecessor in the sense of Def. 3.9.1 in Chapter 5, Section 3, page 25. Assume that this is not the case. This would mean that there exist $i, s < i \leq e$ such that between it and its predecessor there is a loop edge $i-1 \longrightarrow^{l} i$. Following Def. 3.9.1 this would mean that every execution path reaching instruction $i-1$ must pass before through instruction i. As all the instructions in the compilation of an expression are sequential in order that the latter be true there should be a jump to instruction i from outside the compilation of the expression. But this contradicts the first condition. Thus, it follows that our hypothesis is false and we can conclude that the third condition of Def. 7.1.2.1 holds for compilation of expressions.

Qed.

We shall now proceed to the lemma which establishes that there are loops in the bytecode control flow graph corresponding to the compilation of a statement only if the statement contains loops.

Lemma 7.1.2.3. *The compilation $\ulcorner s, S, e \urcorner_{\mathtt{m}}$ of a statement S may contain an instruction k and j which are respectively a loop entry and a loop end in the sense of Def. 3.9.1, page 43 (i.e. there exists j such that $j \longrightarrow^{l} k$) if and only if S contains a substatement S' which is a loop statement:*

$$j = \textbf{\textit{loopEntry}}_{S'} - 1 \wedge k = \textbf{\textit{loopEntry}}_{S'}$$

Proof: By structural induction over the compiled statement. The direction when statement contains a loop statement is trivial. We will show the other direction for compositional statements and if statement.

Compositional statement Let us have the statement $S_1; S_2$ and its compilation $\ulcorner s, S_1; S_2, e \urcorner_{\mathtt{m}}$. From the compiler definition, we have that

$$\ulcorner s, s_1; s_2, e \urcorner_{\mathtt{m}} = \ulcorner s, s_1, e' \urcorner_{\mathtt{m}} \ulcorner e' + 1, s_2, s \urcorner_{\mathtt{m}}$$

By induction hypothesis, the lemma holds for the compilations of S_1 and S_2 which are $\ulcorner s, S_1, e' \urcorner_{\mathtt{m}}$ and $\ulcorner e' + 1, S_2, s \urcorner_{\mathtt{m}}$. Let us see which are the other possible execution edges in the compilation.

Lemma 7.1.2.1 and Lemma B.2 is $e' \longrightarrow e' + 1$. We will show by contradiction that the execution relation between e' and $e' + 1$ is not a loop execution relation. Assume that the execution is a loop execution relation, i.e. $e' \longrightarrow^{l} e' + 1$. Following Def. 3.9.1, this means that every path P in the control flow graph from the program entry point instruction which reaches e' has a subpath $subP$ which does not pass through e' and which passes through $e' + 1$. This is possible in two cases:

- if there is a jump from outside $\ulcorner s, S_1; S_2, e \urcorner_{\mathtt{m}}$ to the instruction $e' + 1$. Two possibilities exist. $S_1; S_2$ is a substatement of statement S_3, i.e. $S_3[S_1; S_2]$ and an instruction from the compilation $\ulcorner s'', S_3, e'' \urcorner_{\mathtt{m}}$ of S_3 but which does not belong to $\ulcorner s, S_1; S_2, e \urcorner_{\mathtt{m}}$ jumps to $e' + 1$. This is not possible following Lemma B.2. The other possibility is that S_3 precedes or follows $S_1; S_2$, i.e. $S_3; S_1; S_2$ or $S_1; S_2; S_3$. But it is not possible that such a jump exists from Lemma B.1
- there is a jump instruction in $\ulcorner s, S_1, e' \urcorner_{\mathtt{m}}$ to $e' + 1$ which is not the instruction e' but this is not possible following Lemma B.1

Conditional statement By definition, its compilation results in

$$\ulcorner s, \texttt{if } (E_1 \, cond \, E_2) \texttt{ then } \{S_1\} \texttt{ else } \{S_2\}, e \urcorner_{\mathtt{m}} = \begin{array}{l} \ulcorner s, E_1, e' \urcorner_{\mathtt{m}} \\ \ulcorner e' + 1, E_2, e'' \urcorner_{\mathtt{m}} \\ e'' + 1 : \texttt{if\_}cond\ e''' + 2 \\ \ulcorner e'' + 2, S_2, e''' \urcorner_{\mathtt{m}} \\ e''' + 1 : \texttt{goto } e \\ \ulcorner e''' + 2, S_1, e - 1 \urcorner_{\mathtt{m}} \\ e : \texttt{nop} \end{array}$$

By induction hypothesis, we get that the lemma holds for the substatement compilations. The possible loop execution edges are :

- $e' \longrightarrow^{l} e' + 1$ This would mean that every execution path reaching e' passes before through $e' + 1$. Let us see how $e' + 1$ can be reached from the program entry point. From Lemma B.2 we know that there could be no jumps from outside the statement compilation inside it except for the first instruction. Thus, every control flow path P reaching $e' + 1$ has a subpath $subP_s$ which first reaches the instruction at index s and does not pass through any instruction from the conditional statement. Because all the instructions in $\ulcorner s, E_1, e' \urcorner_{\mathtt{m}}$ are sequential (Lemma 7.1.2.2), every path from the program entry point reaching $e' + 1$ passes through e'. Thus the assumption is false. We may apply similar reasoning to establish that it is not true that $e'' \longrightarrow^{l} e'' + 1$ neither $e'' + 1 \longrightarrow^{l} e'' + 2$, $e'' + 1 \longrightarrow^{l} e''' + 2$
- $e''' \longrightarrow^{l} e''' + 1$ This is not possible because e''' can be reached from the program entry point by a path P which has a subpath $subP_s$ which reaches s and which does not pass through any instruction from the conditional statement. From the instruction s, the

control flow path passes through $s \ldots e', e'+1 \ldots e'', e''+1, e''+2 \ldots e'''$. Thus there is a path from the program entry point to e''' which does not pass through $e'''+1$ and thus, it is not true that $e''' \longrightarrow^{l} e'''+1$ In the same way, we can show that it is not true that $e'''+1 \longrightarrow^{l} e$, neither $e-1 \longrightarrow^{l} e$

Qed.

For establishing Property 7.1.2.4, we will need several auxiliary lemmas. First, we have to show that the regions described in the exception handler table elements correspond to statements which are declared in the try clause of try catch or try finally statements. For instance, we can return back to the example in Fig. 7.7 on page 103 and see that the exception handler table abs.**ExcHandler** contains one element which describes the unique exception handler in the method. In particular, it states that the region between 2 and 20 is protected from `NullExc` by the bytecode starting at index 22. We may remark that the region between 2 and 20 corresponds to the compilation of the **if** statement.

Lemma B.3 (Exception handler element corresponds to a statement). *Every element* (s, e, eH, Exc) *in the exception handler table* $\mathtt{m}.\mathsf{excHndlS}$ *resulting from the compilation of method* $\mathtt{m}$ *is such that exist statements* S_1, S_2 *and* S *such that* $\ulcorner s, S_1, e \urcorner_{\mathtt{m}}$ *and statement* S *is either a try catch statement of the form* $S = \mathtt{try}\{S_1\}\mathtt{catch}(\text{Exc})\{S_2\}$ *or a try finally statement of the form* $S = \mathtt{try}\{S_1\}\mathtt{finally}\{S_2\}$

Proof: The proof is done by contradiction and follows directly from the definition of the compiler. Particularly, from the compiler definition, we get that elements are added in $\mathtt{m}.\mathsf{excHndlS}$ only in the cases of try catch and try finally statement compilation and that the guarded region in the the newly added element correspond to the try statement.
Qed.
In the following when we refer to the fact that a statement S is either a try catch or a try finally statement with a try substatement S' and we are not interested in the catch or finally part we denote this with $S = \mathtt{try}S' \ldots$

From the compiler definition, we can also see that the indexes of the instructions corresponding to the compilation $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$ of $\mathcal{SE}$ are all comprised in between s and e.

Lemma B.4 (Indexes of the compilation of expressions and statements). *The compilation* $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$ *of* $\mathcal{SE}$ *is such that*

- $s \leq e$
- *every instruction in* $\ulcorner s, \mathcal{SE}, e \urcorner_{\mathtt{m}}$ *has an index in between* s *and* e

The latter follows directly from the compiler definition function.

We establish now several properties concerning substatement relation which has been introduced in Section 7.1.2, page 7.1.2. The next lemma establishes that the substatement relation between statements is preserved by the compiler. In particular, we establish that if a statement is a substatement of another then all of its instructions are contained in the compilation of the other one. Also, if two statements are not in a substatement relation neither their compilations are.

Lemma B.5 (Substatement relation preserved by the compiler). *For all statements* S_1 *and* S_2*, with respective compilations* $\ulcorner s_1, S_1, e_1 \urcorner_{\mathtt{m}}$ *and* $\ulcorner s_2, S_2, e_2 \urcorner_{\mathtt{m}}$ *the following holds*

- *if* S_2 *is a substatement of* S_1 *(*$S_1[S_2]$*) then* $s_1 \leq s_2$ *and* $e_2 \leq e_1$
- *if* S_2 *is not substatement of* S_1 *(*$\neg S_1[S_2]$*), neither* S_1 *is a substatement of* S_2 *(*$\neg S_2[S_1]$*) then* $e_1 < s_2$ *or* $e_2 < s_1$

This also follows from the compiler function.

The next lemma states that if an instruction is part of the compilation of two source statements then we have that these statements are in a substatement relation.

Lemma B.6 (Common instructions in the compilation of statements). *For all statements S_1 and S_2, with respective compilations $\ulcorner s_1, S_1, e_1 \urcorner_{\mathtt{m}}$ and $\ulcorner s_2, S_2, e_2 \urcorner_{\mathtt{m}}$ if it is true that $s_1 \leq k \leq e_1$ and $s_2 \leq k \leq e_2$ then $S_1[S_2]$ or $S_2[S_1]$*

Proof: by contradiction.
The case when S_1 and S_2 are the same is trivial. Let S_1 and S_2 are different. Assume that the above is not true, i.e. *(1)* $\ulcorner s_1, S_1, e_1 \urcorner_{\mathtt{m}}$, *(2)* $\ulcorner s_2, S_2, e_2 \urcorner_{\mathtt{m}}$, *(3)* $s_1 \leq k \leq e_1 \wedge s_2 \leq k \leq e_2$, *(4)* $\neg S_1[S_2] \wedge \neg S_2[S_1]$. From *(4)* and previous Lemma B.5, case 2 we obtain $e_1 < s_2$ or $e_2 < s_1$ But this is in contradiction with *(3)* and thus the lemma holds also in this case.
Qed.

Lemma 7.1.2.4 (Exception handler property). *Let us have a statement S which is not a try catch statement neither a try finally statement in method* `m`. *Assume that statement S' is its direct substatement, i.e. $S[[S']]$. Let their respective compilations be $\ulcorner s, S, e \urcorner_{\mathtt{m}}$ and $\ulcorner s', S', e' \urcorner_{\mathtt{m}}$, then the exception handlers for the instruction points e and e' are the same:*

$$\forall \mathtt{Exc}, \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = \mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS})$$

Proof: by contradiction
Assume the following:
(1) $S[[S']]$, S' is a strict substatement of S
(2) $\ulcorner s, S, e \urcorner_{\mathtt{m}}$, the compilation of S is in between s and e
(3) $\ulcorner s', S', e' \urcorner_{\mathtt{m}}$, the compilation of S' is in between s' and e'
(4) $\exists s_1, e_1, eH_1, s_2, e_2, eH_2,$ Exc such that
$(s_1, e_1, eH_1,$ Exc$)$ is in the exception handler table `m.excHndlS`
$(s_2, e_2, eH_2,$ Exc$)$ is in the exception handler table `m.excHndlS`
$\mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = eH_1$
$\mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS}) = eH_2$
$eH_1 \neq eH_2$
From definition of the function *findExcHandler* in Section 3.5, page 33, we get that if *findExcHandler* returns an exception handler for an index this means that the index is in the region protected by the exception handler:
(5) $s_1 \leq e \leq e_1$
(6) $s_2 \leq e' \leq e_2$
From Lemma B.3 we know that protected regions in the exception handler table correspond to source statements:
(7) $\exists S_1, stmt_1^{try}$ such that $\ulcorner s_1, S_1, e_1 \urcorner_{\mathtt{m}}$ and $stmt_1^{try} = \mathtt{try}\{S_1\} \ldots$
(8) $\exists S_2, stmt_2^{try}$ such that $\ulcorner s_2, S_2, e_2 \urcorner_{\mathtt{m}}$ and $stmt_2^{try} = \mathtt{try}\{S_2\} \ldots$
From Lemma B.5 we know that the compiler preserves the substatement relation and thus, from *(1)* we conclude that the first and last indexes in the compilation of S' are in the region determined by the first s and last index e of the compilation of statement S:
(9) $s \leq s' \wedge e' \leq e$
From Lemma B.4 for instructions in a source statement compilation:
(10) $s \leq e \wedge s' \leq e'$
We conclude from *(9)* and *(10)* that:
(11)$s \leq e' \leq e$
From *(11)*, *(6)* and Lemma B.6 for common instructions in the compilation we get that either S_2 is a substatement of S or vice versa
(12) $S[S_2] \vee S_2[S]$
Similarly, from *(6)* and *(3)* we get:
(13) $S'[S_2] \vee S_2[S']$
From *(12)* and *(13)* we have 4 cases:

$S'[S_2]$ We show first that the cases when S_2 is a direct substatement of S', i.e. $S'[S_2]$ is not possible. Because from *(8)*, we get that S_2 is a direct substatement of S_2^{try} we conclude that

S_2^{try} is a substatement of S', i.e. $S'[S_2^{try}]$. But from Lemma B.5 and the way try catch and try finally statements are compiled we get that $e_2 < e'$. From the last inequality and *(6)* we get a contradiction

$S_2[S'] \wedge S[S_2]$ This is in contradiction with *(1)* which states that S' is a direct substatement of S

$S_2[S'] \wedge S_2[S]$ From this, by Lemma B.5 for compiler substatement preservation we get that *(14)* $s_2 \leq e \leq e_2$ and $s_2 \leq e' \leq e_2$. Using *(7)*, we also get by a similar reasoning as in the first case that it is not possible that the statement S_1 is a substatement of S, i.e. we have that *(15)* $\neg S[S_1]$. From *(2)* and *(5)* by Lemma B.5, we get that *(16)* $S[S_1] \vee S_1[S]$ holds. From *(15)* and *(16)* we get *(17)* $S_1[S]$ which also implies that *(18)* $S_1[S']$ as S' is a (direct) substatement of S. From *(17)* and *(18)* and Lemma B.5, we conclude that *(19)* $s_1 \leq e \leq e_1$ and $s_1 \leq e' \leq e_1$. From *(14)* and *(19)* and definition of the function *findExcHandler* Section 3.5, page 33 we conclude that either *findExcHandler*$(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = e_1$ and *findExcHandler*$(\mathtt{Exc}, e', \mathtt{m.excHndlS}) = e_1$ or that
findExcHandler$(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = e_2$ and *findExcHandler*$(\mathtt{Exc}, e', \mathtt{m.excHndlS}) = e_2$. But this means *findExcHandler*$(\mathtt{Exc}, e, \mathtt{m.excHndlS}) =$
findExcHandler$(\mathtt{Exc}, e', \mathtt{m.excHndlS})$ which is contradiction with *(4)*.

Qed.

Bibliography

[1] Java In The Small. http://www.lifl.fr/RD2P/JITS/.

[2] JC Virtual Machine. http://jcvm.sourceforge.net/.

[3] *Essential .NET*, volume Volume 1: The Common Language Runtime. Addison-Wesley, 2002.

[4] S. Alagić and M. Royer. Next generation of virtual platforms. Article in `odbms.org`, 2005. Available from `http://odbms.org/about_contributors_alagic.html`.

[5] R.M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. Research report 17-2004, LIF, Marseille, France, 2004.

[6] D. Aspinall, L. Beringer, M. Hofmann, H. Loidl, and A. Momigliano. A program logic for resource verification. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, 2004.

[7] D. Aspinall and A. Compagnoni. Heap-bounded assembly language. *Journal of Automated Reasoning*, 31(3-4):261–302, 2003.

[8] D. Aspinall and M. Hofmann. Another type system for in-place update. In *ESOP*, volume 2305 of *LNCS*, pages 36–52, 2002.

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

[10] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (ajit) complilation system. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 142–151, New York, NY, USA, 1999. ACM Press.

[11] F. Y. Bannwart and P. Müller. A logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

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

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO '05)*, LNCS. springer, 2005.

[14] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31(1):82–87, 2006.

[15] 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.

[16] G. Barthe, L. Burdy, J. Charles, B. Gregoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: a tool for validation of security and behaviour of Java applications. In *Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 200x. To appear.

[17] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In *Proceedings of the 13th International Static Analysis Symposium (SAS)*, LNCS, Seoul, Korea, August 2006. Springer-Verlag.

[18] 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.

[19] 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.

[20] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 86–95, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In *Formal Aspects in Security and Trust*, pages 112–126, 2005.

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

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

[24] Egon Borger and Wolfram Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *Mathematical Foundations of Computer Science*, pages 17–35, 1998.

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

[26] L. Burdy. A treatment of partiality: Its application to the b method. 1998.

[27] 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.

[28] 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. Springer, 2003.

[29] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In *Proc. of 13th International Symposium on Formal Methods (FM'05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer-Verlag, 2005.

[30] Cristiano Calcagno, Peter O'Hearn, and Richard Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557–581, 2003.

[31] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. JVer: A Java Verifier. In *Proceedings of the Conference on Computer Aided Verification (CAV'05)*, 2005.

[32] 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.

[33] A. Courbot, M. Pavlova, G. Grimaud, and J.J. Vandewalle. A low-footprint Java-to-native compilation scheme using formal methods. In *proceedings of CARDIS*, pages 329–344, 2006.

[34] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198. ACM Press, 2000.

[35] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[36] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. page 169, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[37] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[38] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

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

[40] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[41] 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.

[42] Cormac Flanagan and Martin Abadi. Types for safe locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108, London, UK, 1999. Springer-Verlag.

[43] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[44] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[45] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM Press.

[46] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[47] L.-A. Fredlund. Guaranteeing correctness properties of a java card applet. In *RV'04*, ENTCS, 2004.

[48] 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.

[49] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and runtime analysis of scoped memory in java. In *RV'04*, ENCS, 2004.

[50] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Sun Microsystems, Inc., 2005.

[51] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[52] G. Grimaud and J.-J. Vandewalle. Introducing research issues for next generation Java-based smart card platforms. In *Proc. Smart Objects Conference (sOc'2003)*, Grenoble, France, 2003.

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

[54] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[55] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of 30th ACM Symp. on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.

[56] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

[57] M. Huisman, B. Jacobs, and J. van den Berg. A Case Study in Class Library Verification: Java's Vector Class. *Software Tools for Technology Transfer*, 3/3:332–352, 2001.

[58] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, 1997.

[59] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128, New York, NY, USA, 1999. ACM Press.

[60] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, Sept 2003.

[61] Bart Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programing*, 58(1-2):61–88, 2004.

[62] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, Stirling, UK, July 2004. Springer-Verlag.

[63] 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.

[64] Laurent Lagosanto. Next-generation embedded java operating system for smart cards. In *4th Gemplus Developer Conference*, 2002.

[65] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2006. to appear.

[66] 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.

[67] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. `http://www.cs.iastate.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`, 2005.

[68] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[69] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Proceedings, Formal Methods*, volume 3582 of *LNCS*, pages 26–42, 2005.

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

[71] K. Rustan M. Leino and Jan L. A. van de Snepscheut. Semantics of exceptions. In *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 447–466. North-Holland, 1994.

[72] R.K. Leino, G. Nelson, and J. B. Saxe. Esc/java user manual. Technical report, Compaq SRC, Oct 2000. 2000-002.

[73] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[74] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[76] C. Marche, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/-JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2003.

[77] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer-Verlag, August 2005.

[78] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd rev. edition, 1997.

[79] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 63–77, London, UK, 2000. Springer-Verlag.

[80] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[81] Deepak Mulchandani. Java for embedded systems. *Internet Computing, IEEE*, 2(3):30 – 39, 1998.

[82] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In *Third USENIX Conference on Object-Oriented Technologies (COOTS)*. USENIX, Portland, Oregon, June 1997.

[83] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006.

[84] David A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, pages 190–204, 2005.

[85] David A. Naumann. On assertion-based encapsulation for object invariants and simulations. *Formal Aspects of Computing*, 2006. Special issue: Applicable Research on Formal Verification and Development, to appear.

[86] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[87] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.

[88] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[89] Tobias Nipkow and Leonor Prensa Nieto. Owicki/gries in isabelle/HOL. In *Fundamental Approaches to Software Engineering*, pages 188–203, 1999.

[90] Cornelis Pierik. *Validation Techniques for Object-Oriented Proof Outlines*. PhD thesis, Proefschrift Universiteit Utrecht, 2003.

[91] Arnd Poetzsch-Heffter and Peter Muller. Logical foundations for typed object-oriented languages. In *PROCOMET '98: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, pages 404–423, London, UK, UK, 1998. Chapman & Hall, Ltd.

[92] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (wat) compiler. In *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, Oregon, June 1997. University of Arizona.

[93] Cornelia Pusch. Proving the soundness of a java bytecode verifier specification in isabelle/hol. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 89–103, London, UK, 1999. Springer-Verlag.

[94] 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.

[95] 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.

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

[97] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP*, pages 551–576, 2005.

[98] 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.

[99] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. In *Conf. version in P. D Mosses, I. Ulidowski, eds., Proc. of 2nd Wksh. on Structured Operational Semantics, SOS'05 (Lisbon, July 2005)*, pages 151–168, 2005.

[100] G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, December 2004.

[101] I. Siveroni. Operational semantics of the java card virtual machine. *Journal of Logic and Algebraic Programming*, 58:3–25, 2004.

[102] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998.

[103] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 132–143, New York, NY, USA, 1977. ACM Press.

[104] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[105] J. C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, CMU, February 2004.

[106] Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. In Mooly Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *LNCS*, pages 326–341. Springer Verlag, 2005.