Project N^o: FP6-015905

Project Acronym: MOBIUS

Project Title: Mobility, Ubiquity and Security

Instrument: Integrated Project

Priority 2: Information Society Technologies

Future and Emerging Technologies

Deliverable D3.1

Byte Code Level Specification Language and Program Logic

Due date of deliverable: 2006-09-01 (T0+12)

Actual submission date: 2006-10-09

Start date of the project: 1 September 2005 Duration: 48 months

Organisation name of lead contractor for this deliverable: WU

Revision 8093 — Final

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Byte Code Level Specification Language and Program Logic

This document describes the results of the work within Task 3.1 of the MOBIUS project. The main challenge of the task was to create a core formalism for specification and verification of bytecode programs. We present here a specification language for sequential Java bytecode (BML) and two verification condition generators. All these components are related to the associated program logic, the MOBIUS base logic, which provides the basis for the MOBIUS PCC infrastructure.

The program logic has been proved sound w.r.t. a formal operational semantics of sequential Java bytecode, which has also been developed as part of Task 3.1. The operational semantics and the MOBIUS base logic have been completely formalised (in Coq), as has the soundness proof of the MOBIUS base logic w.r.t. the operational semantics.

The specification language BML for Java bytecode is the bytecode-level cousin of the specification language JML for Java source code. It provides a syntax comprehensible to programmers, but this syntax has a lot of the syntactic sugar and thus is not the most convenient for use in a program logic or theorem prover. Therefore, the MOBIUS base logic works with a desugared format of specifications instead.

While the MOBIUS base logic is well-suited for expressing the semantics of BML specifications and proving soundness w.r.t. the operational semantics, it is not optimal for the (semi-)automated verification of programs with theorem provers. Therefore, we have also developed variants of the MOBIUS base logic, so-called verification condition generators (or VC generators for short), which are geared towards this. The first of these verification condition generators is essentially a weakest precondition calculus for the MOBIUS base logic. The second of these does not work directly on Java bytecode, but uses BoogiePL as an intermediate language for representing programs. Both VC generators have been proved sound w.r.t. the MOBIUS base logic. (In fact, the second VC generator was proved sound w.r.t. the first.) The reason to develop the second VC generator is that it is easier to support in tools. Also, it allows interoperability with the Spec# initiative at Microsoft research, which also uses BoogiePL as an intermediate language, but for C# programs.

Contents

List of Figures

Chapter 1

Introduction

The main challenge for Task 3.1 of the MOBIUS project (Byte Code Specification and Verification) was to create a core formalism for specification and proof development for bytecode programs, the MOBIUS base logic. This logic provides the foundation of the proof-carrying code (PCC) infrastructure that will be developed within the project.

Formal verification at the level of the bytecode language has several advantages over the standard approach at the source code level. In particular (1) often applications on small devices are developed directly in the low-level language to achieve better optimisation. (2) Proofs concerning bytecode programs can lead to a speed-up in JIT compilation [46]. (3) Java programs are distributed in the bytecode form, which, combined with a formal analysis framework, allows one to develop a PCC infrastructure — one of the main goals of the MOBIUS project. (4) The existence of such a framework enables software distributors to certify programs they do not have the source code for.

The most challenging aspect of the work was to provide a sound and reliable formal basis for a logic of programs for a low-level programming language of such a big size. The logic has to be expressive enough to be used to compile specifications regarding functional and non-functional properties of source code programs into specifications for the corresponding bytecode level programs, but also to prove properties of bytecode programs directly. The major challenge in developing a logic for bytecode is that the program is unstructured and can contain arbitrary jumps. In the context of MOBIUS, functional properties of high-level programs, i.e. formulae asserting the correctness of code, are expected to be primarily expressed in JML. Non-functional properties, i.e. intensional properties such as resource consumption or the absence of unintended information flow, are expressed using program analysis formalisms, (mostly type systems, but also abstract interpretation). Hence, a pragmatic design requirement of the MOBIUS base logic was to admit the representation of both kinds of specification styles in a natural way.

Within the task, two different aspects of a logic to reason about bytecode programs have been studied. The first aspect deals with the specification language used and in particular with the kind of properties that one needs to express about bytecode. This results in the proposal of the Bytecode Modeling Language (BML). The second aspect deals with the verification of bytecode programs. Different verification techniques have been developed, all of which are proved sound w.r.t. a more basic verification technique and, in the end, an operational semantics.

All critical parts of the logic and verification techniques have been proved sound with the Coq proof assistant.

1.1 Overview

This section gives a general description of the different results that have been achieved within this task. Figure 1.1 shows the different formalisms and how they relate to each other. The dashed arrows denote embeddings (i.e. translations that give the meaning for the higher level constructs in terms of lower level constructs), the solid arrows denote soundness results (i.e. they include theorems which state that proofs in the higher level formalism can be rephrased as corresponding proofs in the lower level formalism).

Bicolano is the formal description of the Java Virtual Machine semantics. The MOBIUS base logic has been proved sound w.r.t. this operational semantics.

The MOBIUS base logic allows one to write properties about bytecode as logical properties. This is expressive, but complicated. Therefore, the first line of work has concentrated on the development of an

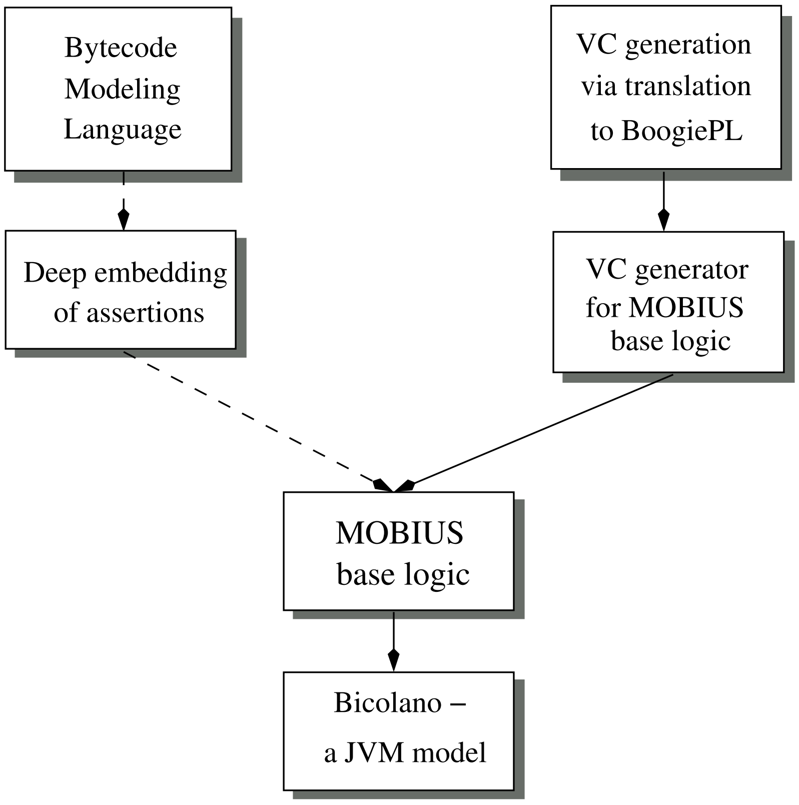
Bytecode Modeling Language
VC generation via translation to BoogiePL
Deep embedding of assertions
VC generator for MOBIUS base logic
MOBIUS base logic
Bicolano – a JVM model

Figure 1.1: The organisation of the components developed in Task 3.1. The boxes denote particular formalisms developed. The dashed edges denote embeddings, the solid edges denote soundness results.

appropriate specification language for bytecode programs. We studied the kind of properties that one would like to express about bytecode and developed a dedicated expression language for this. On top of this, we developed BML (Bytecode Modeling Language), the bytecode-level cousin of the specification language JML [34] for Java source code. While BML provides a convenient syntax for writing specifications, it is not the most convenient for use in a program logic or theorem prover. Therefore we decided to develop the MOBIUS base logic separately, so that the verification can be done on a desugared format of specifications instead (logical predicates, which are a shallow embedding of BML in the logical language of a theorem prover).

While the MOBIUS base logic is well-suited for expressing the semantics of BML specifications and proving soundness w.r.t. the operational semantics, it is not optimal for the (semi-)automated verification of programs with theorem provers. Therefore, the second line of research focused on developing variants of the MOBIUS base logic, so-called verification condition generators (or VC generators for short), which are geared towards (semi-)automated verification of programs with theorem provers. The first of these verification condition generators works directly on bytecode. It is essentially a weakest precondition version of the MOBIUS base logic. The second of these does not work directly on Java bytecode, but uses BoogiePL [21] as an intermediate language for representing programs. Both VC generators have been proved sound: the latter w.r.t. the former, and the former w.r.t. the MOBIUS base logic. The reason for this second VC generator is that it is easier to support in tools. Also, it allows to reuse several tools that are developed within the Spec# initiative at Microsoft research, which also uses BoogiePL as an intermediate language, but for C# programs.

The Bicolano operational semantics, the MOBIUS base logic, the first VC generator, and their respective soundness proofs have been completely formalised in Coq. The expression language for bytecode (section 4.2) has been formalised using the Isabelle proof assistant. The full Coq development can be found at `http://mobius.inria.fr/bicolano`.

The remainder of this chapter gives a more detailed overview of the different components that make up the results of this task.

1.1.1 Bicolano — a model of the Java Virtual Machine

The Java bytecode language is executed by a Java Virtual Machine (JVM). In order to provide an infrastructure to reason about Java bytecode programs, it is necessary to have a formal model of the machine and the bytecode language. Therefore, within the MOBIUS project, we have developed Bicolano, a Coq formalisation of the considerable part of the virtual machine semantics. At first, this semantics has been formulated as a small step semantics, closely following the official Java Virtual Machine specification [38] provided by Sun. Bicolano describes the execution of a bytecode program after the linking phase and covers directly 72 out of 147 bytecode instructions mentioned in Sun's specification. In the formalisation, we have made a few simplifications. The most significant one is that the semantics covers single-threaded behaviour only (a more detailed description of the simplifications is presented in section 2.2).

Subsequently, a big step semantics has been formulated in order to ease the reasoning based on the structure of programs. This big step semantics has been proved sound with regard to the small step semantics, to assure the correctness of further developments. Further, the consistency of the semantics description has been double-checked by a separate team within the project. All formalisations have been done with the Coq proof assistant.

The Bicolano semantics of the JVM is also used in other parts of the project to prove the correctness of the different static analyses that are developed. It has already been used to prove the correctness of an information flow analysis for bytecode in Task 2.1.

1.1.2 MOBIUS base logic

The formal verification of Java bytecode programs requires a formalism which defines a precise format of expressions to describe program properties and proof rules to manipulate them. We developed such a formalism, the MOBIUS base logic, which allows one to describe the properties of Java bytecode programs in terms of pre-, postconditions and invariants. Several of the requirements for the development of the logic came from experiences with specification and verification using JML; this resulted in particular in the need to reason about local annotations. The assertions in the logic can relate the state before and after the execution of a bytecode fragment and can use the full strength of the native logic of the proof assistant in which the proof development is carried out (in our case the extended calculus of constructions from Coq). Proof rules have been provided to establish the correctness of assertions that describe program properties.

The requirements for the development of the logic arose from specification and verification experience using JML, and from the necessity to admit natural representations of type systems and other program analysis formalisms. The first issue resulted in particular in the possibility to reason about local annotations (assert, assume), including annotations that refer to the initial state of a method invocation. The second issue resulted in a judgement form that admits proof rules that reflect the compositionality of type systems. Finally, the correct interpretation of JML annotations as well as the interpretation of many type systems apply to terminating and non-terminating computations. Extending partial-correctness logics, the MOBIUS logic therefore includes a mechanism for reasoning about infinite program executions. This mechanism takes the form of invariants that are required to be satisfied by all states that arise throughout a computation.

The logic has been formalised and proved correct w.r.t. the Bicolano semantics, using the Coq proof assistant, and represents a central component of the MOBIUS architecture. It is used to formulate the semantics of the Bytecode Modeling Language (BML) and also as a basis to prove the correctness of the verification techniques implemented in the MOBIUS tool set. Finally, it will form the basis for the generation of proof certificates.

1.1.3 Specification language for bytecode

The MOBIUS base logic provides the full power of higher order logic to express properties about bytecode programs. This is very expressive, but has the disadvantage that it is a complex language. Therefore, we developed a specification language especially tailored to bytecode. We did this development in two steps: first we identified an appropriate expression language to state properties about bytecode, and then we developed BML, a full specification language for bytecode.

Deep embedding of assertions The form of BML assertions is technically quite far from the assertions in the MOBIUS base logic. In particular, the former describe properties of programs in the class file format, while the latter describe properties after the linking of the program has been completed. In this light it is desirable to introduce a more direct translation from the specification language to the environment of a proof checker in which the syntactical constructs of BML are modelled as algebraic data types. This resulted in a deep embedding of assertions of the MOBIUS base logic, which makes it easier to map BML predicates into the MOBIUS base logic. The deep embedding thus bridges the gap between the assertions in the MOBIUS base logic and BML predicates. Still, the translation from BML to the deep embedding is not trivial, because it must perform linking, so that the program and its annotations are in a form that is appropriate for the MOBIUS base logic.

A reformulation of the MOBIUS base logic as a deep embedding makes sense not only due to more straightforward treatment of the classfiles linking. There is evidence that proofs for deep embeddings are smaller [54]. More importantly, it is more straightforward to redefine the logic in other proof assistants or prover logics when it is deeply embedded. In case a logic is formulated as the deep embedding, one can also formulate the properties of the logic at the meta-level and take advantage of meta-level operations that can be defined in a proof checker to manipulate the formulae of the logic.

Bytecode Modeling Language JML is a widely-used specification language for Java source code. The Bytecode Modeling Language (BML) is developed as a bytecode variation of most of the basic JML constructs. We believe that BML allows one to specify the behaviour of a program in such a way that it is understandable for bytecode programmers.

One of the results of this task was to define a precise description of the intuitive meaning associated with the BML constructs derived from JML. For this, we define a mapping of BML predicates to the deeply embedded expressions of the MOBIUS base logic and subsequently, a mapping for full BML specifications into judgements of the MOBIUS base logic.

This precisely defined specification language will be used by the MOBIUS tool set. It is expected that programmers will be able to write their specifications in BML and that BML will serve as a format to which JML expressions will be compiled by the proof transforming compiler developed in Workpackage 4.

1.1.4 Verification techniques for bytecode programs

VC Generator for MOBIUS base logic The basic idea of the weakest precondition calculus is that the postcondition of an instruction together with the semantics of the instruction gives rise to the weakest formula that guarantees that the postcondition holds. Thus, if one can show that the actual precondition implies this weakest precondition, one can be sure that the postcondition will be established. This idea forms the basis for a verification condition generator: using the weakest precondition calculus, one computes a set of proof obligations that ensure that the program's postcondition will be established. This VC Generator is closely linked to the MOBIUS base logic, by a straightforward transformation of the proof rules into a wp-calculus.

The soundness of the VC Generator is proved in Coq with respect to the formalisation of the MOBIUS base logic.

VC generation via translation to BoogiePL BoogiePL is a programming language in the flavour of Dijkstra's guarded command language. Additionally, it provides a labelled jump construct which makes it more appropriate for (unstructured) bytecode. It uses very few programming constructs and a rich language of expressions that controls the behaviour of the programs. This has the advantage that it is easy to develop many independent proving back-ends for the verification tool set. Moreover, BoogiePL enables the generation of verification conditions the size of which is linear in the length of the BoogiePL program [7, 26]. This is important for efficient verification.

We define a translation of annotated bytecode programs into the BoogiePL language. This non-trivial translation is shown to be sound by showing that any program that can be proved correct after translation into BoogiePL, also can be proved correct as a bytecode program directly. The proof is done by relating the expressions in BoogiePL to the expressions in the weakest precondition calculus for bytecode.

The obtained results allow one to use the BoogiePL back-end in order to obtain sound results with respect to the MOBIUS base logic.

1.2 Task 3.1 within MOBIUS

The formalisms developed within Task 3.1 will be used and extended in many other parts of the project. Figure 1.2 displays the dependencies that are currently foreseen. For each task we briefly list the particular activities that depend on this task.

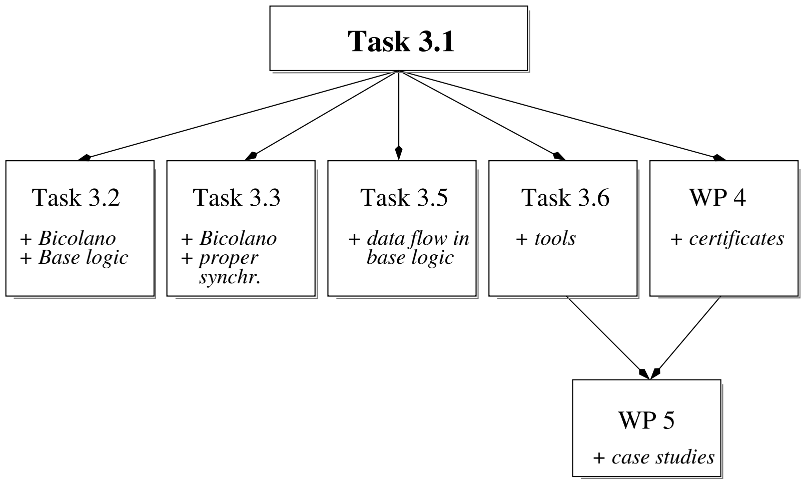

Figure 1.2: The tasks within the MOBIUS project that depend on Task 3.1.

Task 3.2 will extend the Java bytecode semantics Bicolano to take into account the resource usage for the Java Virtual Machine. This task will also extend the MOBIUS base logic with constructs that will enable specification and verification of resource policies and information flow properties. Task 3.3 plans to extend the Bicolano semantics with multi-threading. This will be used to reason about properly synchronised programs. Task 3.5 proposes to look for synergy effects that result from the combination of the type-based verification with the approach based on the logic, in particular type-based analyses will be embedded in the MOBIUS base logic. The tools developed in the Task 3.6 will be using the BML specification language and the verification formalisms developed in the current task. The certificates format as developed in Workpackage 4 will be based on the MOBIUS base logic. The tools developed in Task 3.6 and Workpackage 4 will be used in the case studies done in Workpackage 5, thus this workpackage indirectly also depends on Task 3.1.

1.3 Running example

For a better understanding of the results presented in this document, we consider a single example[1] that will be used to illustrate the different formalisms. Figure 1.3 contains the Java source code together with the JML specification of the example.

The example consists of an abstract class `Bill` that provides a universal implementation of certain billing functionality. The class has a method `produceBill`, which calculates the aggregate cost for a series of investments. The `produceBill` method uses an abstract method `roundCost` which gives the cost of a single investment round. This method is abstract, because the implementation may depend on the communication facilities that are specific to a particular mobile device on which the code will be executed.

The class has one object invariant:

```
//@ invariant sum >= 0;
```

which describes the property of the field of the class which collects the value of the bill. It says that the field `sum` is non-negative at entry or exit from each method in the class. The method `roundCost` has one postcondition:

```
//@ ensures 0 <= \result && \result <= x;
```

[1]In order to make the presentation clear, we had to make the example relatively small as the bytecode representation of programs tends to be very long and the representatnion of the bytecode in proof assistants' logics even longer.

```
abstract class Bill {

  private int sum;
  //@ invariant sum >= 0;

  //@ ensures 0 <= \result && \result <= x;
  abstract int roundCost(int x) throws Exception;

  //@ requires n > 0;
  //@ ensures sum <= \old(sum) + n * (n + 1) / 2;
  public boolean produceBill(int n) {
    int i;
    try{
      //@ loop_invariant 0 < i && i <= n + 1 &&
      //@                0 <= sum && sum <= \old(sum) + (i - 1) * i / 2;
      for (i = 1; i <= n; i++) {
        this.sum = this.sum + roundCost(i);
      }
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}
```

Figure 1.3: The source code of the class `Bill`

which expresses the fact that the cost of the round should not be greater than the sequence number of the round. The other method, `produceBill`, has both a precondition and a postcondition. The precondition:

```
//@ requires n > 0;
```

says that the parameter `n`, which describes the number of rounds for which we calculate the bill, must be positive. This is coupled with the postcondition:

```
//@ ensures sum <= \old(sum) + n * (n + 1) / 2;
```

which describes that the value of the object field `sum` after the calculation of the bill should be less than or equal to the pre-state value of `sum`, increased by the number `n * (n + 1) / 2`. In order to make possible the verification of the `for` loop in the body of the method `produceBill`, we add a loop invariant:

```
//@ loop_invariant 0 < i && i <= n + 1 &&
//@                0 <= sum && sum <= \old(sum) + (i - 1) * i / 2;
```

which is very similar to the postcondition of the method, but it describes intermediate computations in terms of the loop variable `i`. This loop invariant is valid for a particular semantics in which its value is checked at the entry to the loop (i.e. before the entry condition is checked, but in all runs of the loop except the first one after the loop control variable is increased). Notice that the loop invariant combined with the exit condition of the loop gives exactly the postcondition of the method.

Figure 1.4 presents the compiled version of the class `Bill`. First we see the code of the implicit constructor `Bill()` followed by the abstract method `roundCost` for which there is no code. The last chunk of the bytecode listing presents the `produceBill` method. Notice that the `for`-loop from the original source code is organised between the lines labelled from 2 to 24. The exit condition of the loop is checked in the lines labelled by 22–24.

Example in the task formalisms We suggest the reader to glimpse at all the pieces of the example first and then to go to the technical descriptions of particular contributions of the deliverable and study specific renderings of the example more throughly in the course of the descriptions.

```
abstract class Bill extends java.lang.Object{
Bill();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

abstract int roundCost(int);
   throws java/lang/Exception

public boolean produceBill(int);
  Code:
     0 iconst_1
     1 istore_2            //initialisation of i, at location 2, to 1
     2 goto 22 (+20)
     5 aload_0
     6 aload_0
     7 getfield #24 <Bill.sum>
    10 aload_0
    11 iload_2                               //i as a parameter to roundCost
    12 invokevirtual #26 <Bill.roundCost>  //roundCost is invoked
    15 iadd
    16 putfield #24 <Bill.sum>
    19 iinc 2 by 1         //i++
  entry loop:
    22 iload_2
    23 iload_1             //the parameter n is at location 1
    24 if_icmple 5 (-19)   //the loop exit condition is checked
    27 iconst_1
    28 ireturn
    29 astore_3            //the code to handle exception
    30 iconst_0
    31 ireturn
  Exception table:
   from   to  target type
     0    28    29   Class java/lang/Exception

}
```

Figure 1.4: The bytecode of the class `Bill`

To make this effort easier we provide here a reference to places where the example is continued: in subsection 2.3.5 on page 30 in case of Bicolano, in section 3.8 on page 42 in case of the MOBIUS base logic, in section 4.2.4 on page 60 and section 4.3 on page 62 in case of the deep embedding of assertions, in subsection 4.1.1 on page 49 in case of BML, in section 5.1.4 on page 67 in case of the Basic VC Generator, and finnally in section 5.2.3 on page 70 in case of the VC generator for BoogiePL.

Chapter 2

Bicolano — a model of the Java Virtual Machine

Java Virtual Machine (JVM) with its bytecode language is one of the main programming languages considered in the MOBIUS project. The soundness of all the formal techniques envisioned in several parts of the project requires a formal specification of JVM. This specification is formalised in the Coq proof assistant and is called Bicolano.

Bicolano is situated at the bottom of the trusted base of the MOBIUS project. It is a formal description of the Java Virtual Machine (JVM), giving a rigorous mathematical description of Java bytecode program executions. It closely follows the official description of the JVM [38] as provided by Sun. Since the correctness of Bicolano is not formally provable, the close connection with the official specification is essential to gain trust in the specification.

This requirement results in two important design decisions for Bicolano. First, we propose a small step semantics to relate consecutive JVM states during program execution, as it is done in the official description. Second, to describe the JVM we try to keep the same level of detail as in the official description. Nevertheless some simplifications have been made with respect the official documentation, some of which are motivated by the fact that we concentrate on the CLDC platform.

Section 2.1 explains the restrictions that arise from the CLDC platform, and section 2.2 then lists all simplifications that we make. The overall Bicolano architecture is presented in section 2.3. Finally, appendix A gives a more detailed overview of the Bicolano specification; this should provide the necessary information to prove correctness of a formal analysis w.r.t. the JVM.

2.1 Restrictions arising from the CLDC platform

The main focus of the MOBIUS project is on mobile devices. The primary Java configuration for such devices is the Connected Limited Device Configuration (CLDC) [52]. This configuration has several restrictions that allow us to simplify the formalisation of the JVM.

CLDC is defined as a Java 2 Micro Edition (J2ME) configuration. This has several important implications, in particular it means that the specification should apply to all Java realisations on mobile devices and the features included in the configuration must be applicable to many kinds of devices. However, for real devices extensions are often defined as profiles or platform-specific extensions.

A J2ME configuration specification always defines a subset of the Java technology features provided by J2SE. Therefore, the CLDC specification should be considered together with the J2SE specifications of the Java language [27] and the JVM [38] and whenever there is no explicit description in the CLDC specification the feature should be interpreted to be defined as in the general purpose language and machine.

Here we generally refrain from discussing issues related to the API specifications presented in the MIDP specification [51] and other J2ME profile specifications.

Finalization The Java programming language allows to define so-called finalization code, which is executed just before the object is destroyed by the garbage collector. This allows to return to the operating system all the resources that were claimed during the construction and life-time of the object. The specification of CLDC does not allow the use of the instance `Object.finalize` method, thus the applications which are written for CLDC should not rely on this feature. Since the `finalize` methods are called asynchronously, without any specified order and in an unspecified thread [?, §2.17.7], this restriction allows us to avoid many complications for the JVM specification. In addition, one also does not have to consider the synchronisation problems inherently connected with the static `System.runFinalizersOnExit` method.

Exceptions and errors The Java language specification defines so-called asynchronous exceptions. These are exceptions that can occur at any moment of the program execution, as the result of

- an invocation of the `stop` method in Thread or ThreadGroup, or
- JVM internal errors (subclasses of `VirtualMachineError`[1]).

The CLDC configuration excludes the `stop` method from the `Thread` class and the whole `ThreadGroup` class so only internal errors can trigger these asynchronous exceptions.

Virtual machine errors (defined as a subclass of the `VirtualMachineError` class) are serious errors which should never be caught by an application. For the CLDC configuration, the only specified subclass of the `VirtualMachineError` class is `OutOfMemoryError`. If other errors occur, the implementors of a JVM for a mobile device can decide to either halt in an implementation-specific manner or to throw an error of the nearest CLDC-supported class that is a superclass of the error class that would have been thrown otherwise. Thus, all errors other than `OutOfMemoryError` are signalled to the application as a `VirtualMachineError` or cause the halting of the application. Except from these errors all other asynchronous exceptions are forbidden.

The set of errors that can occur in CLDC applications possible contains in addition the following classes:

- `java.lang.Error`,
- `java.lang.NoClassDefFoundError`.

All the other errors are not allowed.

Class loaders and reflection The Java language defines reflection primitives which allow to access and manipulate the program which currently operates. These reflection features are only included in a very limited way in the CLDC/MIDP specifications.

The main restriction concerns class loaders. The specification of CLDC forbids the use of user-defined class loaders. The security requirements imply that a CLDC-conforming JVM must have a single built-in class loader that cannot be overridden, replaced, or reconfigured.

Moreover, the operation of the class loader is further restricted as described on page 21 in subsection Class file format and class loading of the current section.

Threads Virtual machines that conform to CLDC implement multi-threading in a restricted way, in particular they do not implement thread groups and daemon threads.

A thread group is a grouping primitive that allows to perform certain similar operations on multiple threads organised in blocks. On CLDC-based devices this has to be arranged by explicit use of collections.

A daemon thread is a special kind of thread which is not taken into account when the decision whether the application can be exited is taken — the application must exit when all its non-daemon threads cease their existence either by the normal or exceptional exit from the `run` method.

Minor restrictions of the CLDC configuration w.r.t. muli-threading include:

- it is not possible to print the stack of the current thread (this feature is removed to reduce the footprint),
- it is not possible to interrupt threads (no `stop` method),
- there is no nanosecond timer support,
- there is no definition of access rights for threads,
- there is no class loader field,
- it is not possible to wait for the death of another thread for a fixed amount of time, and

[1]As specified in the J2SE API specification [1], the subclasses of `VirtualMachineError` are `InternalError`, `OutOfMemoryError`, `StackOverflowError`, and `UnknownError`.

- it is not possible to change the name of a thread.

Note, that these restrictions mostly are consequences of other restrictions.

Class file verification Class file verification as is done by traditional Java bytecode verifiers is considered to be too expensive for mobile devices. Thus a new verification procedure has been developed, based on adding special `StackMap` attributes to the class files during the development stage. These additional attributes contain information about the size of the method frame and the stack, together with information about the types of the values both in the frame and on the stack. The presence of these attributes speeds up the verification process. The lack of these attributes or inconsistencies in their contents cause the machine to reject the class file.

The new verification process is supposed to give the same guarantees as the original one, namely the type system of the JVM is not violated and memory will not get corrupted. What is more, the additional attributes are simply ignored by the conventional Java Virtual Machine.

The CLDC verifier requires that all subroutines in the bytecode are inlined (which is not an easy task [3]). This means that none of the bytecode instructions `jsr`, `jsr_w`, `ret`, and `wide ret` can occur in the actual bytecode files generated for the platform.

Native functions The standard JVM with its Java Native Invocation (JNI) framework allows to extend the functionality of the Java language with new features that are available to the operating system. In fact, much of the functionality in the standard library is made available through this technology.

However, for CLDC-based devices the application programmer cannot extend the functionality of the language by downloading new libraries that access native features which are not provided by the primarily available CLDC implementation, device profiles, or by the manufacturer-specific classes.

Class file format and class loading The CLDC specification allows certain simplifications and optimisations to the class file format. The following attributes can be ignored by a CLDC implementation:

- the `Synthetic` attribute [?, §4.7.6] that mark fields that are not explicitly mentioned in the source code file,
- the `SourceFile` attribute [?, §4.7.7] that specifies how information about the source file of the class can be obtained,
- the `LineNumberTable` attribute [?, §4.7.8] that is used by debuggers to obtain the relation between line numbers in the source file and the bytecode instructions,
- the `LocalVariableTable` attribute [?, §4.7.9] that is used by debuggers to obtain the values of local variables during the execution of a method, and
- the `Deprecated` attribute [?, §4.7.10] that marks the deprecated members of an object.

In addition, also the consistency of the `InnerClass` attribute need not be checked.

As these attributes might not be available, this has certain consequences for the development of the CLDC and MIDP libraries. In particular, none of the deprecated J2SE methods is reported in the CLDC specification.

A Java application that conforms to the CLDC specification can load application classes and resources only from its own Java Archive (JAR) file. This ensures that applications from different JAR archives cannot share code or fixed data (like icons or default lists of certificates). It is also the case that a third-party application cannot access or modify the private of protected parts of the classes supplied by the manufacturer of the device (i.e. classes belonging to the CLDC, supported profiles or manufacturer-specific classes). Moreover, it is not possible to change the class file lookup order in any CLDC application.

2.2 Coverage of the specification with respect to the JVM

Bicolano makes several simplifications with respect to the official JVM semantics. Before describing the Bicolano specification in more detail, we first list these restrictions and motivate them. A detailed comparison of the Bicolano semantics with the actual JVM specification is presented in appendix A.

A post-linking view Bicolano only handles complete programs and hence is not able to deal with dynamic linking. This choice simplifies the specification considerably, because a large part of the description in the official Sun specification is related to linking.

Hence, we assume that a parser has read a bundle of class files and successfully checked that it constitutes a complete program. This check implies that each time a class name appears in the program, there exists a class with the corresponding name. The same holds for interfaces, fields and methods. Thus, Bicolano only describes the behaviour of a complete program, and any incomplete program should have been rejected by the parser.

In a class file, bytecode instructions refer to constants using symbolic information stored in a table (one per method) named constant pool. Again for simplicity, we rely on the parser to inline this symbolic table, and Bicolano does not handle constant pools. Such an inlining would not be possible if we did not consider programs post-linking.

Class initialisation Static initialisation of classes is not handled by the current version of Bicolano, since in the JVM it is done via the objects of type Class and reflection mechanisms, which are missing from Bicolano. However, some preliminary work to model static class initialisation in Bicolano as closely as possible to the JVM specification has already started. This includes adding an attribute to the class model to specify the initialisation state of the class (`Uninitialised`, `InInitialisation`, `Initialised` or `InitError`), making suitable changes to the small step semantics of instructions which could trigger class initialisation (such as `invokestatic`, `getstatic`, and `new`) and defining a mechanism to track initialisation of classes (and their superclasses) using the JVM method stack.

Currently, we restrict ourselves to a simple semantics, only allowing initialisation of a field by a constant. Class initialisation should be studied further at a later time in the MOBIUS project.

Subroutines Bicolano does not handle subroutines. Instead, we require that source code has been compiled without using subroutines. If it has not, the use of an inlining algorithm is necessary [3]. Recall that in the CLDC platform subroutines cannot be used (see section 2.1).

The semantic consequence of this choice is that there is no `returnAddress` value.

Numerical values Bicolano makes some restrictions on JVM numerical values that it specifies.

First of all, we omit 64 bits values (`double` and `long`). This is essentially for simplification of the specification: the management of 64 bits values is complex because local variables only store 32 bits values, and 64 bits values occupy two consecutive local variables. This would require to add a notion of valid index to the JVM specification. Moreover some instructions (for example `dup`) have a different meaning depending on the size of the elements on top of the dynamic operand stack.

Second, Bicolano does not model `float` numbers, as it is not a priority for the MOBIUS project to prove properties of programs that manipulate floats. However, floating numbers could be added, using an axiomatisation of the IEEE 754 standard.

2.3 Bicolano architecture

This section gives an overview of Bicolano. Figure 2.1 presents the global architecture of the development. At the core of Bicolano is the axiomatic base that describes the notion of program, and specifies the semantic domains and machine arithmetic that we use. We use the Coq module system to model these different components. The operational semantics is defined on top of this axiomatic base. We define a small step and a big step semantics, and we prove equivalence between these. Finally, to show that the axiomatisations that we use are consistent, we give possible concrete instantiations of the different modules that allow to represent particular bytecode programs.

2.3.1 Axiomatic base

As said, at the core of the Bicolano specification are the axiomatisations of program syntax (file `Program.v`), semantic domains (file `Domain.v`) and machine arithmetic (file `Numeric.v`). The latter specifies all necessary numerical operations, conversions between different numeric types etc.

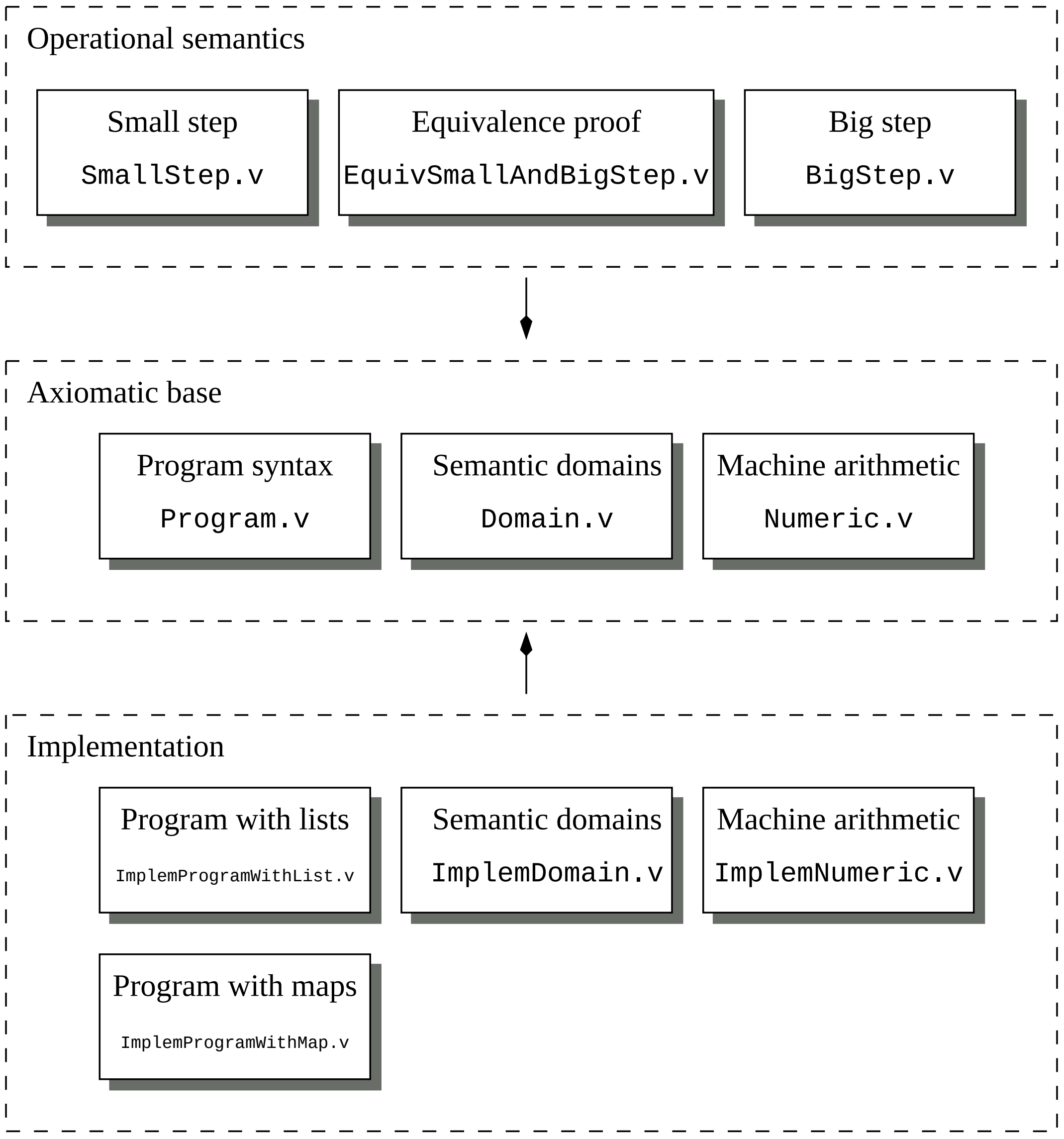

Figure 2.1: Bicolano architecture

Program syntax The syntax of a JVM program is modelled abstractly, using signatures of the Coq module system. This gives rise to cleaner separation of concerns in the formalisation of the semantics. The whole set of axioms is put in a module type named PROGRAM. This axiomatisation is based on various abstract data types:

```
Parameter Program :Set.
Parameter Class : Set.
Parameter Interface : Set.
Parameter Var : Set.
Parameter Field : Set.
Parameter Method : Set.
Parameter BytecodeMethod : Set.
Parameter ExceptionHandler : Set.
Parameter MethodSignature : Set.
Parameter FieldSignature :Set.
Parameter PC : Set.
```

Note that some notions, such as methods and fields, are modelled in two forms: the standard form and the signature form. Such a distinction is necessary because bytecode instructions do not contain direct pointers to methods or fields.

Each abstract type has a list of accessors to manipulate them. We group the accessors of a given type in the same sub-module type. Here we give an example of Program accessors:

```
(** Contents of a Java program *)
Module Type PROG_TYPE.
  (** accessor to a class from its qualified name *)
  Parameter class : Program → ClassName → option Class.
  Definition defined_Class (p:Program) (cl:Class) :=
    class p (CLASS.name cl) = Some cl.

  Parameter name_class_invariant1 : ∀ p cn cl,
    class p cn = Some cl → cn = CLASS.name cl.

  (** accessor to an interface from its qualified name *)
  Parameter interface : Program → InterfaceName → option Interface.
  Definition defined_Interface (p:Program) (i:Interface) :=
    interface p (INTERFACE.name i) = Some i.

  Parameter name_interface_invariant1 : ∀ p cn cl,
    interface p cn = Some cl → cn = INTERFACE.name cl.

End PROG_TYPE.
Declare Module PROG : PROG_TYPE.
```

Notice that the Program structure contains several internal invariants like name_class_invariant1. These are properties that we require to hold for any instantiation of the module, and that can be assumed in the operational semantics.

The axiomatisation of programs ends with a list of definitions using the previous notions such as subtyping and method lookup.

Semantic domains Semantic domains are axiomatised in a module type named SEMANTIC_DOMAIN. Each sub-domain is specified in a sub-module type. For example, the set of local variables is specified as follows.

```
Module Type LOCALVAR.
  Parameter t : Set.
  Parameter get : t → Var → option value.
  Parameter update : t → Var → value → t.
  Parameter get_update_new : ∀ l x v, get (update l x v) x = Some v.
  Parameter get_update_old : ∀ l x y v,
    x<>y → get (update l x v) y = get l y.
End LOCALVAR.
Declare Module LocalVar : LOCALVAR.
```

The most complex axiomatisation of this file concerns heap. It is based on the work of Bannwart and Müller [5]:

```
Module Type HEAP.
  Parameter t : Set.

  Inductive AddressingMode : Set :=
    | StaticField : FieldSignature → AddressingMode
    | DynamicField : Location → FieldSignature → AddressingMode
    | ArrayElement : Location → Int.t → AddressingMode.

  Inductive LocationType : Set :=
    | LocationObject : ClassName → LocationType
    | LocationArray : Int.t → type → LocationType.

  Parameter get : t → AddressingMode → option value.
  Parameter update : t → AddressingMode → value → t.
  Parameter typeof : t → Location → option LocationType.
  Parameter new : t → Program → LocationType → option (Location * t).
 ...
```

The abstract type of heaps (called `t` inside the module type `HEAP`) has two accessors `get` and `typeof` and two modifiers `update` and `new`. These functions are based on the notions of `AddressingMode` and `LocationType`. `AddressingMode` gives the kind of entry in the heap: a field signature for static fields, a location together with a field signature for field values of objects, and a location together with an integer for the element of an array. The definition `get` gives access to the value attached to an indicated address. The definition `typeof` gives the type associated with a location (if there is any). This type is either a class name for objects or a length and a type of elements for arrays. The definition `update` allows to modify a value at a given address. Finally, the definition `new` allows to allocate a new object or a new array.

2.3.2 Operational semantics

Bicolano proposes two different operational semantics: a small step and a big step. Equivalence between these two semantics is formally proved.

Small step semantics The small step semantics follows exactly the reference semantics given in the official specification. It consists of an elementary relation named `step` between states of the virtual machine. A standard state is of the form `(St h (Fr m pc s l) sf)` where `h` is a heap; `(Fr m pc s l)` is the current frame composed of the current method `m`, the current program point `pc`, the local variables `l` and the operand stack `s`; and finally `sf` is the call stack. An exceptional state is of the form `(StE h (FrE m pc loc l) sf)` where all elements are similar to those found in a standard state, except the location of the exception object `loc`, which replaces the operand stack. Exceptional states occur when an exception is thrown, but control has not yet reached the corresponding exception handler.

The `step` relation is given by an inductive relation. We give here a fragment describing the semantics of the `putfield` instruction.

```
Inductive step (p:Program) : State.t → State.t → Prop :=
...
| putfield_step_ok : ∀ h m pc pc' s l sf f loc cn v,

  instructionAt m pc = Some (Putfield f) →
  next m pc = Some pc' →
  Heap.typeof h loc = Some (Heap.LocationObject cn) →
  defined_field p cn f →
  assign_compatible p h v (FIELDSIGNATURE.type f) →

  step p (St h (Fr m pc (v::(Ref loc)::s) l) sf)
         (St (Heap.update h (Heap.DynamicField loc f) v) (Fr m pc' s l) sf)

| putfield_step_NullPointerException : ∀ h m pc s l sf f v h' loc',

  instructionAt m pc = Some (Putfield f) →
```

```
    Heap.new h p
      (Heap.LocationObject (javaLang,NullPointerException)) =  Some (loc',h') →

    step p (St h (Fr m pc (v::Null::s) l) sf)
           (StE h' (FrE m pc loc' l) sf)
  ...
```

The definition of this relation closely follows the definition given in the official specification. Together with the axiomatisation presented above, it forms the trusted base of Bicolano.

Big step semantics Since JVM states contain a frame stack to handle method invocations, it is often convenient to use an equivalent semantics where method invocations are performed in one big step transition for showing the correctness of static analyses and program logics. Therefore we introduce a new semantics relation:

```
IntraStep (p:Program) : Method →IntraNormalState →IntraNormalState + ReturnState →Prop
```

This relation denotes transitions of a method between two internal states, i.e. JVM states that only contain one frame (instead of a frame stack), or between an internal state and a return state, i.e. a pair of a heap and a final result (a JVM value or an exception object in case of termination by an uncaught exception).

The definition of `IntraStep` depends on four different relations:

- `NormalStep (p:Program) : Method →IntraNormalState →IntraNormalState →Prop` which defines the normal step relation without method calls or exception throwing;
- `ExceptionStep (p:Program) : Method →IntraNormalState →IntraExceptionState →Prop` which defines the step in which an exception is thrown;
- `CallStep (p:Program) : Method →IntraNormalState →Method*(OperandStack.t*LocalVar.t) →Prop` which defines the step in which a method is called; and
- `ReturnStep (p:Program) : Method →IntraNormalState →ReturnState →Prop` which defines the step in which a method terminates normally.

We show a fragment of `NormalStep`, specifying the behaviour of the `putfield` instruction:

```
| putfield : ∀ h m pc pc' s l f loc cn v,

    instructionAt m pc = Some (Putfield f) →
    next m pc = Some pc' →
    Heap.typeof h loc = Some (Heap.LocationObject cn) →
    defined_field p cn f →
    assign_compatible p h v (FIELDSIGNATURE.type f) →

   NormalStep p m(pc,(h,(v::(Ref loc)::s),l))
                        (pc',(Heap.update h (Heap.DynamicField loc f) v,s,l))
```

These relations are combined to express the three different kinds of steps that can occur:

- `exec_intra` for steps inside the method:

```
Inductive exec_intra (p:Program) (m:Method) : IntraNormalState →
       IntraNormalState → Prop :=
| exec_intra_normal : ∀ s1 s2,
   NormalStep p m s1 s2 →
   exec_intra p m s1 s2
| exec_exception : ∀ pc1 h1 h2 loc2 s1 l1 pc',
  ExceptionStep p m (pc1,(h1,s1,l1)) (h2,loc2) →
  CaughtException p m (pc1,h2,loc2) pc' →
  exec_intra p m (pc1,(h1,s1,l1))
               (pc',(h2,Ref loc2::OperandStack.empty,l1)).
```

- `exec_return` for steps that end the current execution of the method:

```
    Inductive exec_return (p:Program) (m:Method) : IntraNormalState →
             ReturnState → Prop :=
      | exec_return_normal : ∀ s h ov,
         ReturnStep p m s (h,Normal ov) →
         exec_return p m s (h,Normal ov)
      | exec_return_exception : ∀ pc1 h1 h2 loc2 s1 l1,
         ExceptionStep p m (pc1,(h1,s1,l1)) (h2,loc2) →
         UnCaughtException  p m (pc1,h2,loc2) →
         exec_return p m (pc1,(h1,s1,l1)) (h2,Exception loc2).
```

- `exec_call` to perform a call and directly use the corresponding result:

```
       Inductive exec_call (p:Program) (m:Method) :
       IntraNormalState → ReturnState → Method  → IntraNormalState →
              IntraNormalState+ReturnState → Prop :=
     | exec_call_normal : ∀ m2 pc1 pc1' h1 s1 l1 os l2 h2 bm2 ov,
         CallStep p m (pc1,(h1,s1,l1 )) (m2,(os,l2)) →
         METHOD.body m2 = Some bm2 →
         next m pc1 = Some pc1' →
         exec_call p m
            (pc1,(h1,s1,l1))
            (h2,Normal ov)
            m2
            (BYTECODEMETHOD.firstAddress bm2,(h1,OperandStack.empty,l2))
            (inl _ (pc1',(h2,cons_option ov os,l1)))

     | exec_call_caught : ∀ m2 pc1 pc1' h1 s1 l1 os l2 h2 loc bm2,
         CallStep p m (pc1,(h1,s1,l1 )) (m2,(os,l2)) →
         METHOD.body m2 = Some bm2 →
         CaughtException p m (pc1, h2, loc) pc1' →
         exec_call p m
            (pc1,(h1,s1,l1))
            (h2,Exception loc)
            m2
            (BYTECODEMETHOD.firstAddress bm2,(h1,OperandStack.empty,l2))
            (inl _(pc1',(h2,Ref loc::nil,l1)))
     | exec_call_uncaught : ∀ m2 pc1 h1 s1 l1 os l2 h2 loc bm2,
         CallStep p m (pc1,(h1,s1,l1 )) (m2,(os,l2)) →
         METHOD.body m2 = Some bm2 →
         UnCaughtException p m (pc1, h2, loc)  →
         exec_call p m
           (pc1,(h1,s1,l1))
           (h2,Exception loc)
           m2
           (BYTECODEMETHOD.firstAddress bm2,(h1,OperandStack.empty,l2))
           (inr _ (h2,Exception loc)).
```

`IntraStep` is then defined as a case distinction, using the previous relations.

```
 Inductive IntraStep (p:Program) :
    Method → IntraNormalState → IntraNormalState + ReturnState → Prop :=
  | IntraStep_res :∀ m s ret,
     exec_return p m s ret →
     IntraStep p m s (inr _ ret)
  | IntraStep_intra_step:∀ m s1 s2,
     exec_intra p m s1 s2 →
     IntraStep p m s1 (inl _ s2)
  | IntraStep_call :∀ m m' s1 s' ret' r,
     exec_call p m s1 ret' m' s' r →
     TransStep_l (IntraStep p m') s' (inr _ ret') →
     IntraStep p m s1 r.
```

For the method call case, a recursive call to the transitive closure of `IntraStep` (obtained with the help of `TransStep_l`) is exploited to link the initial state of the method called and its return value.

The definition of IntraStep is exploited to define several transitive closures which covey the idea of the big step semantics. IntraStepStar is the definition of the transitive closure of the internal step semantics relation IntraStep.

```
Definition IntraStepStar p m s r := TransStep_l (IntraStep p m) s r
```

The actual big step reduction definition BigStep uses the transitive closure above to relate the internal state with the return state:

```
Definition BigStep  p m s ret := IntraStepStar p m s (inr _ ret).
```

As we discuss here the transitive closures, it is worth describing the Reachable relation which holds if from one state in a particular method we can reach another state in another method.

```
Definition Reachable P M s s' :=
  exists M',  ClosReflTrans (ReachableStep P) (M,s) (M',s').
```

The definition of reachable states differs slightly from the big step definitions above as it is reflexive and uses a separate definition of a single step, ReachableStep, in case a method call is executed:

```
Inductive ReachableStep (P:Program) :
     (Method*IntraNormalState)→(Method*IntraNormalState) →Prop :=
  | ReachableIntra : ∀ M s s',
      IntraStep P M s (inl _ s') →
      ReachableStep P (M,s) (M,s')
  | Reachable_invS : ∀ M pc h os l M' os' l' bm',
      CallStep P M (pc,(h,os,l)) (M',(os',l')) →
      METHOD.body M' = Some bm' →
      ReachableStep P (M, (pc,(h,os,l)))
        (M', (BYTECODEMETHOD.firstAddress bm',(h,OperandStack.empty,l'))).
```

Proof of equivalence We have formally proved the correctness of the big step semantics with respect to the reference small step semantics, by showing that the notion “evaluation of method m in program p from states s terminates with the final value *ret*” coincides in both semantics. To achieve this result we have to relate the transitive closures of the two semantics. This result is stated in theorem Equiv_SmallStep_BigStep:

```
Theorem Equiv_SmallStep_BigStep : ∀ P m s ret sf,
  BigStep  P m s ret
    ⇔
  IntraBigStep_from_SmallStep P sf m s ret.
```

IntraBigStep_from_SmallStep is the definition of the big step judgement in terms of the small step relation: the transitive closure of step without going below the initial call stack and terminating with a return instruction or an uncaught exception in the same initial call stack.

```
Inductive IntraBigStep_from_SmallStep (P:Program) (sf:CallStack.t) (m:Method) :
  IntraNormalState → ReturnState → Prop :=
 | IntraBigStep_from_SmallStep_value : ∀ h1 pc1 l1 s1 h2 pc2 s2 l2 sr,
    step_closure_prefix_sf P
      (St h1 (Fr m pc1 s1 l1) sf)
      (St h2 (Fr m pc2 s2 l2) sf) →
    ReturnStep P m (pc2,(h2,s2,l2)) sr →
    IntraBigStep_from_SmallStep P sf m (pc1,(h1,s1,l1)) sr
 | IntraBigStep_from_SmallStep_exception : ∀ h1 pc1 l1 s1 h2 pc2 loc2 l2,
    step_closure_prefix_sf P
      (St h1 (Fr m pc1 s1 l1) sf)
      (StE h2 (FrE m pc2 loc2 l2) sf) →
    UnCaughtException P m (pc2,h2,loc2) →
    IntraBigStep_from_SmallStep P sf m (pc1,(h1,s1,l1)) (h2,Exception loc2).
```

It is worth mentioning that the equivalence theorem does not reference BigStep directly, but it uses just the expansion of its definition.

```
Module Bill.
  ...
  Definition produceBillInstructions : list (PC*Instruction) :=
    (0%N, Const INT 1%Z)::
    (1%N, Vstore Ival 2%N)::
    (2%N, Goto 20%Z)::
    (5%N, Vload Aval 0%N)::
    (6%N, Vload Aval 0%N)::
    (7%N, Getfield Bill.sumFieldSignature)::
    (10%N, Vload Aval 0%N)::
    (11%N, Vload Ival 2%N)::
    (12%N, Invokevirtual Bill.roundCostSignature)::
    (15%N, Ibinop AddInt)::
    (16%N, Putfield Bill.sumFieldSignature)::
    (19%N, Iinc 2%N 1%Z)::
    (22%N, Vload Ival 2%N)::
    (23%N, Vload Ival 1%N)::
    (24%N, If_icmp LeInt (-19)%Z)::
    (27%N, Const INT 1%Z)::
    (28%N, Vreturn Ival)::
    (29%N, Vstore Aval 3%N)::
    (30%N, Const INT 0%Z)::
    (31%N, Vreturn Ival)::
    nil
  .

  Definition produceBillHandlers : list ExceptionHandler :=
    (EXCEPTIONHANDLER.Build_t (Some java_lang_Exception.className) 0%N 29%N 29%N)::
    nil
  .

  Definition produceBillBody : BytecodeMethod := BYTECODEMETHOD.Build_t
    produceBillInstructions
    produceBillHandlers
    4                        (* max # of locals *)
    4                        (* max operand stack size *)
  .

  Definition produceBillMethod : Method := METHOD.Build_t
    produceBillSignature
    (Some produceBillBody)
    false                   (* final *)
    false                   (* static *)
    false                   (* native *)
    Public                  (* visibility *)
  .

  Definition class : Class := CLASS.Build_t
    className
    (Some java_lang_Object.className)
    nil
    (sumField::nil)
    (_init_Method::roundCostMethod::produceBillMethod::nil)
    false                   (* final *)
    false                   (* public *)
    true                    (* abstract *)
  .

End Bill.
```

Figure 2.2: A fragment of the translation to Bicolano for the class `Bill`

2.3.3 Implementations of module interfaces

To show that the axiomatisation in the core of Bicolano is consistent, we give implementations of the different parts. We currently propose two different implementations of the program syntax; one is based on lists and the other on efficient maps. The first allows a more readable program format while the second provides a more efficient manipulation of programs.

We also provide implementations of the semantic domains and machine arithmetic axiomatisation. These are currently incomplete, but should be completed during the rest of the MOBIUS project.

2.3.4 Bytecode verification in the JVM

As a part of the linking phase of bytecode, the JVM performs several additional checks to ensure several wellformedness properties of the linked program (the properties are described in [?, §4.8]). This process is called bytecode verification in the JVM[2]. Since the properties that are guaranteed by the bytecode verification in the JVM are important and simplify many proof developments, we decided to describe them as a part of Bicolano even though their checking formally belongs to the linking phase.

As in the JVM specification, the constraints that are enforced are divided into two groups: static and structural. Each of them is encapsulated in a separate Coq module. The static constraints are relatively simple syntactic restrictions, while the structural constraints require a control flow analysis of the loaded program. This control flow analysis keeps track of

- the depth of the operand stack;
- the types of the elements on the operand stack;
- whether the cells of a local array are initialised;
- the type of elements in the initialised cells of arrays; and
- the program pointer (to associate the data with particular bytecode program points).

2.3.5 Running example

Appendix B gives the full translation of the running example in the list format mentioned above. Figure 2.2 shows an interesting fragment of the translation. The class representation defined at the bottom of the figure consists of the name of the class, the superclass representation, the list of interfaces implemented by the class, the list of the class fields, the list of the class methods, the indication that the class is not final (the first false), the indication that the class is not public (the second false), and the indication that the class is abstract (true). The figure also contains the definition of the method `produceBillMethod`. The constructor for methods (`METHOD.Build_t`) takes as parameters the signature of the method, its body and several modifiers of the definition (e.g. `public`). The definition of the method body uses the list of instructions (`produceBillInstructions`), the list of exception handlers (`produceBillHandlers`), and some auxiliary information about the number of local variables and the maximal size of the operand stack.

[2]Notice that there is a terminology clash here between the standard Java community meaning of bytecode verification, and the verification whether a bytecode program adheres to its specification, as we will develop within the MOBIUS project. Therefore, in this document we explicitly use the term bytecode verification in the JVM to denote the operation done by the JVM and bytecode verification to denote the verification of a bytecode program w.r.t. its specification.

Chapter 3

MOBIUS base logic

The MOBIUS base logic is a formal program logic for expressing general functional and non-functional properties of JVM programs. Its consistency is justified with respect to the Bicolano operational semantics presented in chapter 2. We first outline the requirements imposed on the logic by the specification formalisms and proof-generating mechanisms that are envisioned to be part of the MOBIUS framework as well as by the security and resource policies which we aim to certify. We then introduce a form of judgements that extends partial-correctness logics by a mechanism for reasoning about non-terminating program executions and present a corresponding proof system. We include a description of the proof of soundness with respect to the Bicolano semantics where the formal interpretation of judgements makes the earlier informal explanation of their meaning precise. Finally, we conclude with the presentation of the running example. The technical part of the presentation (i.e. the material conveyed in sections 3.3 to 3.7) is based on a representation of the program logic in the theorem prover Coq (using a shallow embedding of assertions) and a formalisation of its soundness proof w.r.t. the Bicolano formalisation.

3.1 Requirements

Two aims governed the design of the MOBIUS logic. Firstly, logic-based verification as pursued in Workpackage 3 is based on the Java Modeling Language (JML, [34]), or its bytecode equivalent, the Bytecode Modeling language (BML, see section 4.1). Consequently, the format of assertions and judgements, and the formulation of proof rules, are required to be defined in a way that admits a smooth translation of BML specifications. Two specification constructs were identified as being of particular importance:

- code annotations at intermediate program points, interpreted as assertions that should be valid whenever the annotated program point is visited;
- the possibility to refer to the initial values of object fields in method post-conditions and intermediate program annotations through the specification keyword `\old`.

In addition to the representation of specification constructs, a further requirement consists of the ability to justify verification strategies associated with JML/BML (verification condition generators, WP-calculi, ...) w.r.t. the MOBIUS base logic formalism.

The second group of requirements stems from the work in Workpackage 2, where type systems and other program analysis frameworks are being developed that analyse intensional code properties such as information flow safety and resource consumption. In order to be able to communicate the results of these analyses from code producer to code consumer using PCC-based technology (considered in Workpackage 4), it is necessary to express the code invariants that define the soundness property of the analyses in the program logic. In particular, an efficient method to validate type-based certificates at the consumer side is obtained if the typing judgements are given interpretations in the program logic in such a way that the syntax-directedness of typing rules can be reflected in derived proof rules for the program-logic interpretations. Consequently, the judgement format of the program logic needs to satisfy two requirements:

- the interpretation of the judgement at a program point *pc* within a JVM program must refer to the program execution from *pc* onwards irrespectively of the way the program execution happened to arrive at *pc*;

- the proof rules must be formulated in such a way that the hypotheses refer to the control flow successors, as typing judgements for composite expressions depend upon typing judgements for their subexpressions.

Both requirements result from the syntax-directedness of typing systems where type judgements describe the behaviour independent of surrounding code and typing rules are oriented so that hypothetical judgements concern subexpressions.

Finally, a requirement from both logic-based verification and type-based verification concerns the ability to reason about non-terminating program executions. In fact, the above-mentioned interpretation of JML-annotations at intermediate program points applies even if the continued execution of the enclosing method fails to terminate. In case of type systems, the extension of interpretations to non-terminating executions strengthens the property that is guaranteed by a typing judgement. In fact, while many syntactic soundness proofs (e.g. proofs based on big-step operational judgements and subject-reduction theorems) fail to consider non-terminating executions, the type systems the soundness of which is proved often admit a stronger notion of soundness that does include non-terminating executions.

The constraint to consider non-terminating executions makes program logics that interpret judgements as partial or total correctness assertions insufficient. The MOBIUS base logic therefore includes invariants, i.e. assertions that constrain all executions of a program phrase.

Another point worth noting is the apparent conflict between the JML requirement that assertions should be able to refer to the initial state of method invocation and the type-motivated requirement that judgements only refer to the program continuation. The MOBIUS logic satisfied both requirements by including initial states in all assertion forms but interpreting judgements as mandated by the interpretation of type systems.

We now give a brief overview of the MOBIUS logic before describing selected technical details in the ensuing sections. All notions in the remainder of this chapter are formulated with respect to an arbitrary but fixed program, which we denote by P. Method identifiers m are of the form $m = (C, M)$ where C is a class name and M a method name, while program points $pc = (m, l)$ combine method identifiers and instruction labels. Finally, the initial label of method m is denoted by $init_m$.

3.2 Overview of the logic

Global specification and verification structure The verification of a Bicolano-represented JVM program is formulated and carried out with respect to a method specification table M that associates with each method identifier m of the program

- a method specification $\mathsf{S} = (R, T, \Phi)$, comprising a precondition R, a postcondition T, and a method invariant Φ;
- a local specification table G, i.e. a context of local proof assumptions;
- a local annotation table Q that collects the (optional) assertions associated with labels in m.

Informally, an entry $\mathsf{M}(m) = ((R, T, \Phi), \mathsf{G}, \mathsf{Q})$ is to be understood as follows.

The tuple (R, T) constitutes a partial-correctness specification, i.e. the postcondition $T(s_0, t)$ is expected to hold whenever an execution of m with initial state s_0 that satisfies $R(s_0)$ terminates, where t is the final state. As in VDM [30], the additional dependency of T on s_0 eliminates the necessity of introducing auxiliary variables. Method executions that do not terminate satisfy T trivially.

The tuple (R, Φ) constitutes a method invariant: assertion $\Phi(s_0, s)$ is expected to hold for any state s that arises during the (terminating or non-terminating) execution of m with initial state s_0 satisfying $R(s_0)$. This interpretation includes states arising in sub-frames, i.e. in invocations of further methods m' initiated during the execution of m.

The annotation table Q is a finite partial map from labels occurring in m to assertions $Q(s_0, s)$. If label l is annotated by Q, then $Q(s_0, s)$ will be expected to hold for any state s encountered at program point (m, l) during a terminating or non-terminating execution of m with initial state s_0 satisfying $R(s_0)$. Annotations are mostly of method-internal interest as the code structure is of little meaning outside the method definition.

Finally, the proof context G collects proof assumptions that may be used during the verification of the method. It consists of a finite partial map from labels in m to the components (A, B, I) of local proof judgements $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$, to be described in due course. In order to avoid faulty assumptions to be included in G, the complete verification of a program requires each entry in G to be justified, in the following way.

Given a program P with specification M, the verification task consists of showing that M is justified. For each entry $\mathsf{M}(m) = (\mathsf{S}, \mathsf{G}, \mathsf{Q})$, we need to show that:

1. The body b_m of m satisfies the method specification. This amounts to deriving the judgement $\mathsf{G}, \mathsf{Q} \vdash \{A_0\}\, m, init_m\, \{B_0\}\, (I_0)$ where the assertion A_0, B_0, and I_0 are extracted from S (see below).

2. The proof context G is justified. As entries in G may be accessed in the proof of item (1) by means of an axiom rule, the verification of G prevents the insertion of invalid proof assumptions by a malevolent proof producer.

The precise formulation of what constitutes a verified program will be given in section 3.6.3 below.

Local verification The verification of method bodies uses judgements of the form $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$. Here,

1. A is a (local) precondition, i.e. a predicate $A(s_0, s)$ that relates the state s at program point pc (i.e. the state prior to executing the instruction at that program point) to the initial state s_0 of the current method invocation.

2. B is a (local) postcondition, i.e. a predicate $B(s_0, s, t)$ that relates the state s at pc to the initial state s_0 and the final state t of the current method invocation, provided the execution of the current method invocation terminates.

3. I is a (local) invariant, i.e. a predicate $I(s, s')$ that relates the state s at pc to any future state encountered during the continued execution of the current method, including those arising in subframes.

4. G is the proof context which may be used to store recursive proof assumptions, as needed e.g. for the verification of loops.

Additionally, if $pc = (m, l)$ and $\mathsf{Q}(l) = Q$, then the judgement $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$ implicitly also mandates that $Q(s_0, s)$ holds for all states s encountered at l, where s_0 is as before.

The core of our program logic is a proof system for judgements $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$, comprising syntax-directed proof rules and logical rules. Syntax-directed rules are essentially of the form[1]:

$$\frac{\mathsf{G}, \mathsf{Q} \vdash \{A_1\}\, pc_1\, \{B_1\}\, (I_1) \quad \cdots \quad \mathsf{G}, \mathsf{Q} \vdash \{A_n\}\, pc_n\, \{B_n\}\, (I_n) \qquad \phi_1 \cdots \phi_k}{\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I),}$$

where $pc_1, \ldots pc_n$ are the control flow successors of pc and the ϕ_i are side conditions governing the applicability of a rule or relating components of the hypothetical judgements to components of the concluding judgement. In logical rules, all assumptions are side conditions.

In the remainder of this chapter, we describe some details of the logic. We describe the syntactic format of the various assertion forms, define the semantic interpretation of judgements, discuss the structure of the proof system, present some selected proof rules, and outline the soundness proof. In order to keep the text readable, we give a simplified presentation that glosses over various technicalities. The reader who is interested in the mathematical details is referred to the reference implementation of the logic, which we have carried out in the theorem prover Coq w.r.t. the Bicolano operational semantics[2].

[1]For technical reasons, the hypothetical judgements actually are of a different form, $\mathsf{G}, \mathsf{Q} \vdash \langle A_i\rangle\, pc_i\, \langle B_i\rangle\, (I_i)$, as described in section 3.6. In this brief overview (section 3.2) we gloss over this fact although the difference between the two forms is important for the soundness property.

[2]Available at `http://mobius.inria.fr/bicolano`.

3.3 Format of assertions and judgements

3.3.1 States

The above informal description of the logic mentions different kinds of states, namely initial states, states at program program points, and terminal states. Our presentation expresses the difference between these categories by using the following three types, where $\mathcal{S}$ denotes the type of stores (in JVM parlance: collection of local registers), $\mathcal{H}$ the type of object heaps, $\mathcal{O}$ the type of operand stacks, and $\mathcal{V}$ that of JVM values.

Initial states $s_0 \in \mathit{InitState} = \mathcal{S} \times \mathcal{H}$

represent states before the first instruction of a method has been executed. Initial states are used in partial method specifications, method invariants, as well as in annotations, local pre- and postconditions, and invariants.

Local states $s \in \mathit{LocalState} = \mathcal{O} \times \mathcal{S} \times \mathcal{H}$

contain all components of a JVM state and occur at intermediate program points, namely in annotations, local pre- and postconditions, and in invariants.

Terminal states $t \in \mathit{TermState} = \mathcal{V}_r \times \mathcal{H}$

contain a return value and a heap, and occur in partial-correctness method specifications and in local postconditions. Here, a return value is either a proper value $v \in \mathcal{V}$, or an exceptional return value (an address representing the location of the exception object in the heap), or the empty value (value of type void).

In addition to improving readability, the introduction of these categories introduces a basic discipline which governs the state components to which assertions may refer. For $s_0 = (S, H)$ we write $\mathit{state}(s_0) = ([\,], S, H)$ for the local state that extends s_0 with an empty operand stack.

3.3.2 Assertions

As mentioned before, judgements relate a program point $pc = (m, l)$ to a context G, a precondition A, a postcondition B, and an invariant I, and implicitly refer to an optional annotation Q. These assertion forms are of the following types.

Local assertions of type $\mathcal{A} \equiv (\mathit{InitState} \times \mathit{LocalState}) \to \mathit{Prop}$
occur as preconditions A and annotations Q, and relate the current state to the initial state of the current frame.

Postconditions $B \in \mathcal{B} = (\mathit{InitState} \times \mathit{LocalState} \times \mathit{TermState}) \to \mathit{Prop}$
relate the current state to the initial and final state of a (terminating) execution of the current frame.

Invariants $I \in \mathcal{I} = (\mathit{LocalState} \times \mathit{LocalState}) \to \mathit{Prop}$
relate the current state to a local state of the same frame or a subframe of the current frame.

3.3.3 Specifications

The behaviour of a method is specified by the following kinds of assertions

$$\begin{array}{rcl} \mathsf{S} \in \mathit{MethSpec} & = & \mathit{MethPre} \times \mathit{MethPost} \times \mathit{MethInv} \\ R \in \mathit{MethPre} & = & (\mathit{InitState}) \to \mathit{Prop} \\ T \in \mathit{MethPost} & = & (\mathit{InitState} \times \mathit{TermState}) \to \mathit{Prop} \\ \Phi \in \mathit{MethInv} & = & (\mathit{InitState} \times \mathit{LocalState}) \to \mathit{Prop} \end{array}$$

Method preconditions R are predicates on initial states. Following the partial-correctness regime, preconditions do not guarantee termination.

Method postconditions T constrain the behaviour of terminating executions of the method and thus relate their initial and final states.

Method invariants Φ specify the behaviour of terminating and non-terminating executions of a method by relating initial states and local states. Semantically, a method invariant Φ will be required to hold for all states that arise throughout the execution of a method, including those occurring in dynamically created subframes.

As was mentioned already in section 3.2, a program specification is given by a table M that associates with each method a method specification S, a local specification table G, and a local annotation table Q.

3.4 Interpretation of assertions and judgements

The interpretation of proof judgements is defined with the help of various operational notions, all formulated with respect to a method m. In order to simplify the presentation we use $(s, l) \Rightarrow^* (s', l')$ to denote the transitive and reflexive closure of the single-step relation (also used if s is an initial state), $(s, l) \Downarrow t$ to denote the big-step execution (completion of $\Rightarrow^*$ until the end of the current frame), and $(s, l) \leadsto (s', l')$ to denote the reflexive and transitive closure of the single-step relation which descends into subframes. These correspond to the Bicolano relations *IntraStepStar*, *BigStep*, and *Reachable* as follows[3]:

$$\begin{aligned}
(s, l) \Rightarrow^* (s', l') &::= \exists n.\ \mathit{IntraStepStar}\ P\ m\ n\ (s, l)\ \mathit{inl}(s', l'),\\
(s_0, l) \Rightarrow^* (s', l') &::= (\mathit{state}(s_0), l) \Rightarrow^* (s', l'),\\
(s, l) \Downarrow t &::= \mathit{BigStep}\ P\ m\ (s, l)\ t,\\
(s, l) \leadsto (s', l') &::= \mathit{Reachable}\ P\ m\ n\ (s, l)\ \mathit{inl}(s', l').
\end{aligned}$$

Definition 3.4.1 A proof judgement is valid, notation $\models_{\mathsf{Q}} \{A\}\, m, l\, \{B\}\, I$, if $s_0, \mathit{init}_m \Rightarrow^* (s, l)$ and $A(s_0, s)$ implies

- if $(s, l) \Downarrow t$ then $B(s_0, s, t)$;
- if $(s, l) \leadsto (s', l')$ then $I(s, s')$;
- if $(s, l) \Rightarrow^* (s', l')$ and $\mathsf{Q}(l') = Q$ then $Q(s_0, s')$.

A program is valid for program specification M if all method specifications in M are valid, i.e. if for all m, R, T, Φ, and G, and Q, $\mathsf{M}(m) = ((R, T, \Phi), \mathsf{G}, \mathsf{Q})$ implies $\models_{\mathsf{Q}} \{A\}\, m, \mathit{init}_m\, \{B_T\}\, I_\Phi$, where

$$\begin{aligned}
A &= \lambda\ (s_0, s).\ R(s_0) \wedge s = \mathit{state}(s_0),\\
B_T &= \lambda\ (s_0, s, t).\ s = \mathit{state}(s_0) \to T(s_0, t),\ \text{and}\\
I_\Phi &= \lambda\ (s, r).\ \forall\ s_0.\ s = \mathit{state}(s_0) \to \Phi(s_0, r).
\end{aligned}$$

3.5 Assertion transformers

In order to simplify the presentation of the proof rules, we define operators that relate assertions occurring in judgements of adjacent instructions. These assertion transformers resemble WP-operators, but are separately defined for preconditions, postconditions, and invariants.

For basic instructions (arithmetic operations, object manipulations, etc.) the operators for normal (i.e. non-exceptional) execution relate the assertions for a single step from label l to its successor label l'[4]:

$$\begin{aligned}
\mathit{PRE}(m, l, l', A)(s_0, r) &= \exists\ s.\ \mathit{NormalStep}(m, (l, s), (l', r)) \wedge A(s_0, s)\\
\mathit{POST}(m, l, l', B)(s_0, r, u) &= \forall\ s.\ \mathit{NormalStep}(m, (l, s), (l', r)) \to B(s_0, s, u)\\
\mathit{INV}(m, l, l', I)(r, u) &= \forall\ s.\ \mathit{NormalStep}(m, (l, s), (l', r)) \to I(s, u)
\end{aligned}$$

Roughly speaking, the precondition $\mathit{PRE}(m, l, l', A)$ holds for a state r at label l' if r can be reached from l, s in a single step for some s satisfying A. Similar readings may be given to the transformers for postconditions and invariants.

[3]The relations *IntraStepStar*, *BigStep* and *Reachable* are explained in section 2.3.2 on page 27.
[4]The explanation of the NormalStep relation is on page 26.

Further operators $PRE^{\mathsf{e}}(\cdot,\cdot,\cdot,\cdot)$, $POST^{\mathsf{e}}(\cdot,\cdot,\cdot,\cdot)$, and $INV^{\mathsf{e}}(\cdot,\cdot,\cdot,\cdot)$ capture exceptional behaviour of basic instructions, but have been omitted from this presentation.

For the instruction Invokestatic, the non-exceptional transformers mediate between the assertions that enclose the execution of a method call.

$$
\begin{array}{l}
PRE_{sinv}(R,T,A,N) = \\
\quad \lambda\ s_0\ s.\ \exists\ O\ S\ h\ H\ O'\ w. \\
\qquad\qquad |O'| = N\ \wedge\ s = (w :: O, S, h)\ \wedge\ A\ s_0\ (O'{+}{+}O, S, H)\ \wedge \\
\qquad\qquad (R\ (frame(O'{+}{+}O, N), H) \\
\qquad\qquad\quad \to T\ (frame(O'{+}{+}O, N), H)\ (h, w)) \\
POST_{sinv}(R,T,B,N) = \\
\quad \lambda\ s_0\ s\ t. \forall\ O\ S\ h\ H\ O'\ w. \\
\qquad\qquad |O'| = N \to s = (w :: O, S, h) \to \\
\qquad\qquad (R\ (frame(O'{+}{+}O, N), H) \to T(frame(O'{+}{+}O, N), H)\ (h, w)) \\
\qquad\qquad\quad \to B\ s_0\ (O'{+}{+}O, S, H)\ t \\
INV_{sinv}(R,T,I,N) = \\
\quad \lambda\ s\ t. \forall\ O\ S\ h\ H\ O'\ w. \\
\qquad\qquad |O'| = N \to s = (w :: O, S, h) \to \\
\qquad\qquad (R\ (frame(O'{+}{+}O, N), H) \to T(frame(O'{+}{+}O, N), H)\ (h, w)) \\
\qquad\qquad\quad \to I\ (O'{+}{+}O, S, H)\ t
\end{array}
$$

In contrast to the transformers for the basic instructions, the effect of the instruction of interest is calculated from the method specification, not the operational semantics. The expression $frame(O'{+}{+}O, N)$ denotes the initial register store of the new frame. This is obtained by popping the arguments O' from the operand stack $O'{+}{+}O$ and assigning them to the local variables $1, \dots, N$, where N is the number of formal parameters of the invoked method. The implication $R(\cdot,\cdot) \to T(\cdot,\cdot)$ represents the satisfaction of the (partial-correctness) method specification by the initial and final states of the invoked method. Again, we elide the definition of assertion transformers for exceptional behaviour.

Similar assertion transformers have been defined for the Invokevirtual instruction, for normal behaviour, exceptional behaviour that arises during the execution of the invoked method and cannot be handled locally, and exceptional behaviour that prevents the method invocation from being initiated due to the presence of a Null reference in the argument position for the object reference (NullPointerException).

3.6 Proof rules

We now turn to the proof rules. The proof system has two judgement forms, $\mathsf{G}, \mathsf{Q} \vdash \{A\}\ pc\ \{B\}\ (I)$ and $\mathsf{G}, \mathsf{Q} \vdash \langle A\rangle\ pc\ \langle B\rangle\ (I)$. The former one can be considered the more fundamental one — this is also the one used in the definition of verified programs (section 3.6.3 below), the syntactic counterpart to valid programs as defined in section 3.4 above. Both judgement forms associate a program point with a precondition, a postcondition, and an invariant, relative to a proof context G. Contexts are finite maps which associate triples (A', B', I') with further program points pc', and are primarily used to store recursive proof assumptions during the verification of loops. The motivation for using two judgement forms stems from the interaction between the rules that alter the flow of control inside a method frame (principally conditional and unconditional jumps, but also all instructions that may throw exceptions) and the rule AX that extracts such assumptions from G. We will motivate the distinction between the two formats after having introduced the rules.

3.6.1 Syntax-directed rules

We start with the syntax-directed rules. These are oriented in such a way that the conclusion is an unconstrained judgement and proof hypotheses refer to successor instructions. Thus, their application during proof search or verification follows the flow of control.

Basic instructions We define a single rule for specifying the behaviour of basic instructions, such as instructions that access local variables or merely manipulate the operand stack (including arithmetic operations) as well as instructions for object creation and manipulation, exception raising, and unconditional jumps:

$$basic(m,l) \equiv P(m,l) \in \left\{ \begin{array}{l} \mathsf{Iload}\ x, \mathsf{Istore}\ x, \ldots \mathsf{Const}\ t\ z, \mathsf{Ibinop}\ o, \ldots, \mathsf{New}\ C, \\ \mathsf{Getfield}\ F, \mathsf{Putfield}\ F, \mathsf{Getstatic}\ F, \mathsf{Putstatic}\ F, \ldots, \mathsf{Athrow}, \mathsf{Goto}\ l' \end{array} \right\}.$$

Ignoring exceptions for a moment, and denoting the immediate control flow successor of l in m by $next_m(l)$, the preliminary rule

$$\frac{\begin{array}{c} basic(m,l) \qquad \forall s_0\ s.\ A\ s_0\ s \to I\ s\ s \\ \forall Q.\ \mathsf{Q}(l) = Q \to (\forall s_0\, s.\ A\ s_0\ s \to Q\ s_0\ s) \\ \forall l'.\ next_m(l) = l' \to\ \mathsf{G},\mathsf{Q} \vdash \langle PRE(m,l,l',A)\rangle\, m,l'\, \langle POST(m,l,l',B)\rangle\, (INV(m,l,l',I)) \end{array}}{\mathsf{G},\mathsf{Q} \vdash \{A\}\, m,l\, \{B\}\, (I)}$$

contains as hypothesis a judgement for the successor instruction the assertions of which are related to the main assertions by the assertion transformers for normal termination. In addition, side conditions ensure that the invariant I and any local annotation Q are satisfied in any state reaching label l. Satisfaction of I in later states and satisfaction of local annotations Q' of later program points are guaranteed by the judgement for $next_m(l)$.

In order to capture the exceptional behaviour of basic instructions the implemented rule extends the above preliminary rule by the hypothesis

$$\forall l'\ e.\ l' \in handle_m(l,e) \to\ \mathsf{G},\mathsf{Q} \vdash \langle PRE^{\mathsf{e}}(m,l,e,A)\rangle\, m,l'\, \langle POST^{\mathsf{e}}(m,l,e,B)\rangle\, (INV^{\mathsf{e}}(m,l,e,I)) \tag{3.1}$$

and the side condition

$$\begin{array}{l} \forall s_0\, s\, h\, a\, e. \quad ExceptionStep(m,(s,l),(h,a)) \to typeOf(h,a) = e \to \\ \qquad (\forall l'. \neg l' \in handle_m(l,e)) \to A\ s_0\ s \to B\ s_0\ s\ (h,a). \end{array} \tag{3.2}$$

Hypothesis (3.1) covers exceptions that can be handled locally. Similarly to the hypothesis for normal behaviour, the derivability of a judgement is required for the label l' at which the execution continues. Dynamically, this label l' is determined by the (dynamic) type of the thrown exception object e. In order to capture all possible labels of this kind, the hypothesis is applied to all labels that are associated with l in the handler table for m, which we denote by $handle_m$. As the flow of control remains in the same frame, the assertions in the judgement for l' are related to those in the judgement for l by some appropriate assertion transformers. In this case, the transformers model the effect of raising and handling the exception.

Exceptions that cannot be handled locally are propagated to the enclosing method frame, i.e. the current method frame is terminated. Therefore, the side condition (3.2) requires the method specification B to hold with respect to any state in which the (dynamic) type of the exception has no entry in the handler table, where the dynamic type is obtained by retrieving the object at location a in the heap that resulted from the attempt to execute the instruction at l.

Thus, the final rule for basic instructions is

$$\frac{\begin{array}{c} basic(m,l) \qquad \forall s_0\ s.\ A\ s_0\ s \to I\ s\ s \\ \forall Q.\ \mathsf{Q}(l) = Q \to (\forall s_0\, s.\ A\ s_0\ s \to Q\ s_0\ s) \\ \forall l'\ e. \quad l' \in handle_m(l,e) \to \\ \mathsf{G},\mathsf{Q} \vdash \langle PRE^{\mathsf{e}}(m,l,e,A)\rangle\, m,l'\, \langle POST^{\mathsf{e}}(m,l,e,B)\rangle\, (INV^{\mathsf{e}}(m,l,e,I)) \\ \forall s_0\, s\, h\, a\, e. \quad ExceptionStep(m,(s,l),(h,a)) \to typeOf(h,a) = e \to \\ (\forall l'. \neg l' \in handle_m(l,e)) \to A\ s_0\ s \to B\ s_0\ s\ (h,a) \\ \forall l'.\ next_m(l) = l' \to\ \mathsf{G},\mathsf{Q} \vdash \langle PRE(m,l,l',A)\rangle\, m,l'\, \langle POST(m,l,l',B)\rangle\, (INV(m,l,l',I)) \end{array}}{\mathsf{G},\mathsf{Q} \vdash \{A\}\, m,l\, \{B\}\, (I).} \quad \text{(INSTR)}$$

Conditional jumps In Bicolano, jump destinations are calculated from an offset and the current label, where the offset is encoded in the instruction. For simplicity, we omit this detail in the presentation (though not in the formalisation) and use the jump label directly.

$$\frac{\begin{array}{c} P(m,l) = \mathsf{If}\ comp\ l' \qquad \forall s_0\ s.\ A\ s_0\ s \to I\ s\ s \\ \forall Q.\ \mathsf{Q}(l) = Q \to (\forall s_0\, s.\ A\ s_0\ s \to Q\ s_0\ s) \\ \mathsf{G},\mathsf{Q} \vdash \langle PRE(m,l,l',A)\rangle\, m,l'\, \langle POST(m,l,l',B)\rangle\, (INV(m,l,l',I)) \\ \forall l''.\ next_m(l) = l'' \to \mathsf{G},\mathsf{Q} \vdash \langle PRE(m,l,l'',A)\rangle\, m,l''\, \langle POST(m,l,l'',B)\rangle\, (INV(m,l,l'',I)) \end{array}}{\mathsf{G},\mathsf{Q} \vdash \{A\}\, m,l\, \{B\}\, (I).} \quad \text{(IF)}$$

No exceptions can be raised by these instructions.

Method invocation and return We present the rules for static and virtual method invocation and $\mathsf{Ireturn}$.

The second side condition of the rule for the invocation of static methods,

$$\frac{\begin{array}{c} P(m,l) = \mathsf{Invokestatic}\ m' \quad |params(m')| = N \\[1ex] \mathsf{M}(m') = ((R,T,\Phi),\mathsf{G}',\mathsf{Q}') \qquad \forall\, s_0\ s.\ A\ s_0\ s \to I\ s\ s \\[1ex] \forall\, Q.\ \mathsf{Q}(l) = Q \to (\forall\, s_0\, s.\ A\ s_0\ s \to Q\ s_0\ s) \\[1ex] \begin{array}{ll} \forall\ s_0\ O'\ O\ S\ H\ t. & |O'| = N \to A\ s_0\ (O'{+}{+}O,S,H) \to \\ & (R(frame(O'{+}{+}O,N),H) \to \Phi\ (frame(O'{+}{+}O,N),H)\ t) \to \\ & I\ (O'{+}{+}O,S,H)\ t \end{array} \\[1ex] \mathsf{G},\mathsf{Q} \vdash \langle PRE_{sinv}(R,T,A,N)\rangle\, m, next_m(l)\, \langle POST_{sinv}(R,T,B,N)\rangle\, (INV_{sinv}(R,T,I,N)) \\[1ex] \begin{array}{ll} \forall\ l'\ e.\ l' \in handle_m(l,e) \to \mathsf{G},\mathsf{Q} \vdash & \langle PRE^{\mathsf{e}}_{sinv}(R,T,A,N,l,e)\rangle \\ & m, l' \\ & \langle POST^{\mathsf{e}}_{sinv}(R,T,B,N,l,e)\rangle \\ & (INV^{\mathsf{e}}_{sinv}(R,T,I,N,l,e)) \end{array} \\[1ex] \begin{array}{ll} \forall\ s_0\ O\ O'\ S\ H\ h\ a\ e. & A\ s_0(O'{+}{+}O,S,H) \to |O'| = N \to \\ & (R\ (frame(O'{+}{+}O,N),H) \to T\ (frame(O'{+}{+}O,N),H)\ (a,h)) \to \\ & typeOf(h,a) = e \to (\forall\, l'.\neg l' \in handle_m(l,e)) \to \\ & B\ s_0\ (O'{+}{+}O,S,H)(a,h) \end{array} \end{array}}{\mathsf{G},\mathsf{Q} \vdash \{A\}\, m, l\, \{B\}\ (I),} \quad \text{(INVS)}$$

extracts the length N of the formal list of parameters of the invoked method m' from the program representation, while the third side condition extracts the specification for m' from the table M. The forth and fifth side conditions are as before, while side condition

$$\begin{array}{ll} \forall\ s_0\ O'\ O\ S\ H\ t. & |O'| = N \to A\ s_0\ (O'{+}{+}O,S,H) \to \\ & (R(frame(O'{+}{+}O,N),H) \to \Phi\ (frame(O'{+}{+}O,N),H)\ t) \to \\ & I\ (O'{+}{+}O,S,H)\ t \end{array}$$

enforces that the caller's invariant I is satisfied w.r.t. the state prior to the method invocation and any internal state t. The invariant is required to be satisfied whenever the local precondition of the invoking instruction holds and the satisfaction of the callee's method invariant Φ w.r.t. the callee's initial state and t follows from the satisfaction of the callee's method precondition.

The hypothetical judgement

$$\mathsf{G},\mathsf{Q} \vdash \langle PRE_{sinv}(R,T,A,N)\rangle\, m, next_m(l)\, \langle POST_{sinv}(R,T,B,N)\rangle\, (INV_{sinv}(R,T,I,N))$$

relates the assertions associated with the program point of the invocation and its successor using the transformers defined in the previous section, while the remaining two side conditions deal with exceptions that arise during the execution of the invoked method but are not handled inside the callee's frame. The first side condition treats the case where such an exception is handled by the caller, i.e. the handler table of m associates a continuation label l' with the method invocation's label l and the type of the raised exception. The second side condition treats the case where the caller's handler table does not provide a means for dealing with the exception and the caller's frame is terminated.

The rule for virtual method invocations,

$$
\frac{\begin{array}{c}
P(m,l) = \mathsf{Invokevirtual}\ m' \quad |params(m')| = N \\[1ex]
\mathsf{M}(m') = ((R,T,\Phi),\mathsf{G}',\mathsf{Q}') \qquad \forall\, s_0\ s.\ A\ s_0\ s \to I\ s\ s \\[1ex]
\forall\, Q.\ \mathsf{Q}(l) = Q \to (\forall\, s_0\, s.\ A\ s_0\ s \to Q\ s_0\ s) \\[1ex]
\begin{array}{ll}
\forall\ s_0\ O'\ O\ S\ H\ t\ a. & |O'| = N \to A\ s_0(O'{+}{+}(\mathsf{Ref\ a}) :: O,S,H) \to \\
& \left(\begin{array}{l} R(frame(O'{+}{+}(\mathsf{Ref\ a}) :: O, 1+N), H) \\ \quad \to \Phi(frame(O'{+}{+}(\mathsf{Ref\ a}) :: O, 1+N), H)\ t \end{array}\right) \to \\
& I\ (O'{+}{+}(\mathsf{Ref\ a}) :: O,S,H)\ t
\end{array} \\[1ex]
\mathsf{G},\mathsf{Q} \vdash \langle PRE_{vinv}(R,T,A,N)\rangle\, m, next_m(l)\, \langle POST_{vinv}(R,T,B,N)\rangle\ (INV_{vinv}(R,T,I,N)) \\[1ex]
\begin{array}{ll}
\forall\ l'\ e.\ l' \in handle_m(l,e) \to \mathsf{G},\mathsf{Q} \vdash & \langle PRE^{\mathsf{e}}_{vinv}(R,T,A,N,l,e)\rangle \\
& m, l' \\
& \langle POST^{\mathsf{e}}_{vinv}(R,T,B,N,l,e)\rangle \\
& (INV^{\mathsf{e}}_{vinv}(R,T,I,N,l,e))
\end{array} \\[1ex]
\begin{array}{ll}
\forall\ s_0\ O\ O'\ S\ H\ a'\ h\ a\ e. & A\ s_0\ (O'{+}{+}(\mathsf{Ref\ a'}) :: O,S,H) \to |O'| = N \to \\
& \left(\begin{array}{l} R\ (frame(O'{+}{+}(\mathsf{Ref\ a'}) :: O, 1+N), H) \\ \quad \to T\ (frame(O'{+}{+}(\mathsf{Ref\ a'}) :: O, 1+N), H)\ (a,h) \end{array}\right) \to \\
& typeOf(h,a) = e \to (\forall\, l'.\neg l' \in handle_m(l,e)) \to \\
& B\ s_0\ (O'{+}{+}(\mathsf{Ref\ a'}) :: O,S,H)(a,h)
\end{array} \\[1ex]
\begin{array}{ll}
\forall\ s_0\ O'\ O\ S\ H\ h\ a. & A\ s_0\ (O'{+}{+}\mathsf{Null} :: O,S,H) \to |O'| = N \to \\
& \mathsf{new}(H, \mathsf{NullPointerException}) = (a,h) \to \\
& \left(\begin{array}{l}
\left(\begin{array}{l} (\forall\, l'.\neg l' \in handle_m(l, \mathsf{NullPointerException})) \\ \quad \to B\ s_0\ (O'{+}{+}\mathsf{Null} :: O,S,H)(a,h) \end{array}\right) \wedge \\
\left(\begin{array}{ll} \forall\ l'. & l' \in handle_m(l, \mathsf{NullPointerException}) \\
& \to \mathsf{G},\mathsf{Q} \vdash \begin{array}{l} \langle PRE^{\mathsf{NullPointerException}}_{vinv}(A,N)\rangle \\ m.l' \\ \langle POST^{\mathsf{NullPointerException}}_{vinv}(B,N)\rangle \\ (INV^{\mathsf{NullPointerException}}_{vinv}(I,N)) \end{array} \end{array}\right)
\end{array}\right)
\end{array}
\end{array}}
{\mathsf{G},\mathsf{Q} \vdash \{A\}\, m, l\, \{B\}\ (I),}
\qquad \text{(INVV)}
$$

is similar but contains a further hypothesis that deals with a NullPointerException. The specification extracted from the table M is the one associated with the statically identified method. The soundness result therefore requires that specifications of overriding methods refine the specifications of the overridden methods conservatively (behavioural subtyping).

Finally, the rule for Ireturn essentially links the local precondition A with the local postcondition B, mediated by the effect of executing the return instruction:

$$
\frac{\begin{array}{c}
P(m,l) = \mathsf{Ireturn} \qquad \forall\, s_0\ s.\ A\ s_0\ s \to I\ s\ s \\
\forall\, Q.\ \mathsf{Q}(l) = Q \to (\forall\, s_0\, s.\ A\ s_0\ s \to Q\ s_0\ s) \\
\forall\ s_0\ s.\ A\ s_0\ s \to \forall t.\ ReturnStep(m,(s,l),t) \to B\ s_0\ s\ t
\end{array}}
{\mathsf{G},\mathsf{Q} \vdash \{A\}\, m, l\, \{B\}\ (I).}
\qquad \text{(IRETURN)}
$$

The rules for the other return instructions are similar.

3.6.2 Logical rules

The logical rules terminate the verification of a loop by accessing a recursive assumption (rule AX), limit the growth of assertions (rules of consequence, one for each judgement form), and mediate between the judgement forms (rule INJECT). As is the case in traditional program logics, the rules of consequence allow preconditions to be strengthened, while postconditions and invariants may be weakened.

$$
\frac{\begin{array}{c}
\mathsf{G},\mathsf{Q} \vdash \langle A'\rangle\, l\, \langle B'\rangle\ (I') \qquad \forall\ s_0\ s\ A\ s_0\ s \to A'\ s_0\ s \\
\forall\ s_0\ s\ t.B'\ s_0\ s\ t \to B\ s_0\ s\ t \qquad \forall\ s\ r.\ I'\ s\ r \to I\ s\ r
\end{array}}
{\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m, l\, \langle B\rangle\ (I),}
\qquad \text{(CONSEQ-T)}
$$

$$\frac{\begin{array}{cc}\mathsf{G},\mathsf{Q}\vdash\{A'\}\,l\,\{B'\}\,(I') & \forall\, s_0\ s\ A\ s_0\ s\rightarrow A'\ s_0\ s\\ \forall\, s_0\ s\ t.B'\ s_0\ s\ t\rightarrow B\ s_0\ s\ t & \forall\, s\ r.\ I'\ s\ r\rightarrow I\ s\ r\end{array}}{\mathsf{G},\mathsf{Q}\vdash\{A\}\,m,l\,\{B\}\,(I),}\qquad\text{(CONSEQ-F)}$$

$$\frac{\mathsf{G},\mathsf{Q}\vdash\{A\}\,m,l\,\{B\}\,(I)}{\mathsf{G},\mathsf{Q}\vdash\langle A\rangle\,m,l\,\langle B\rangle\,(I),}\qquad\text{(INJECT)}$$

$$\frac{\begin{array}{c}\mathsf{G}(m,l)=A,B,I\qquad\forall\, s_0\ s.\ A\ s_0\ s\rightarrow I\ s\ s\\ \forall\, Q.\ \mathsf{Q}(l)=Q\rightarrow(\forall\, s_0\, s.\ A\ s_0\ s\rightarrow Q\ s_0\ s)\end{array}}{\mathsf{G},\mathsf{Q}\vdash\langle A\rangle\,m,l\,\langle B\rangle\,(I).}\qquad\text{(AX)}$$

3.6.3 Verified programs

The verification of P with respect to a method specification table M and an annotation table Q proceeds by proving the following property *VerifiedProg*.

$$\begin{array}{lll}
\mathit{VerifiedProg} & \equiv & \forall\, m\ R\ T\ \Phi\ \mathsf{G}\ \mathsf{Q}.\ \mathsf{M}(m)=((R,T,\Phi),\mathsf{G},\mathsf{Q})\rightarrow\\
& & \mathsf{G},\mathsf{Q}\vdash\ \{\lambda\ (s_0,s).\ R(s_0)\wedge s=\mathit{state}(s_0)\}\\
& & \qquad m,\mathit{init}_m\\
& & \qquad \{\lambda\ (s_0,s,t).\ s=\mathit{state}(s_0)\rightarrow T(s_0,t)\}\\
& & \qquad (\lambda\ (s,r).\ \forall\ s_0.\ s=\mathit{state}(s_0)\rightarrow\Phi(s_0,r))\\
& & \wedge\ \forall\ l\ A\ B\ I.\ \mathsf{G}(l)=(A,B,I)\rightarrow\mathsf{G},\mathsf{Q}\vdash\{A\}\,m,l\,\{B\}\,(I)\\
& & \wedge\ \forall\ m'\ S'\ \mathsf{G}'\ \mathsf{Q}'.\ \ \mathsf{M}(m')=((R',T',\Phi'),\mathsf{G}',\mathsf{Q}')\rightarrow\\
& & \qquad\qquad \mathit{classOf}(m')\leq\mathit{classOf}(m)\rightarrow\mathit{overrides}(m',m)\rightarrow\\
& & \qquad\qquad \left(\begin{array}{l}((\forall\ s_0\ s.R'\ s_0\rightarrow T'\ s_0\ s)\rightarrow(\forall\ s_0\ s.R\ s_0\rightarrow T\ s_0\ s))\ \wedge\\ ((\forall\ s_0\ s.R'\ s_0\rightarrow\Phi'\ s_0\ s)\rightarrow(\forall\ s_0\ s.R\ s_0\rightarrow\Phi\ s_0\ s))\end{array}\right).
\end{array}$$

The first condition requires that all method specifications in M are justified by derivations for (the initial labels of) their method bodies. Note the similarity of this clause with the assertions in the definition of valid programs. Similarly, the second condition mandates that all entries of G be backed up by derivations for the respective code blocks. Typically, the proof contexts G will be provided as part of the PCC certificate communicated together with the code. Finally, the third clause enforces that the specification table obeys the restrictions of behavioural subtyping (abbreviated: BST), i.e. the specifications of methods in subclasses imply the specifications of identically named methods in super-classes. By the virtue of bytecode verification, the dynamic type of an object on which a virtual method will be invoked is guaranteed to be a subclass of the statically identified type. In the definition, the notation $\mathit{classOf}(m)$ denotes the (Bicolano-defined) class name identifying the class in which method m is defined and $\leq$ denotes the sub-class relationship.

3.6.4 Discussion of the judgement forms

In order to motivate the introduction of two judgement forms, suppose for a moment that all judgements are of the form $\mathsf{G},\mathsf{Q}\vdash\langle A\rangle\,pc\,\langle B\rangle\,(I)$. For an instruction at program point pc suppose that G contains the entry $\mathsf{G}(pc)=(A,B,I)$ and suppose that when the verification process reaches pc, we indeed wish to justify $\mathsf{G},\mathsf{Q}\vdash\langle A\rangle\,pc\,\langle B\rangle\,(I)$. Then, the verification may conclude by rule AX. For a complete verification, however, the assumptions in G also need to be justified. What this justification consists of is expressed in the definition of verified programs: discharging an assumption $\mathsf{G}(pc)=(A,B,I)$ requires us to prove $\mathsf{G},\mathsf{Q}\vdash\langle A\rangle\,pc\,\langle B\rangle\,(I)$. Thus, in a proof system with only one form of judgements, any assumed context entry could be discharged trivially, by using the axiom rule, and the code block the initial instruction of which is referred in the assumption would never be inspected! Clearly, such a logic would be unsound. We see three possible solutions:

Solution 1: Instead of introducing a general axiom rule, one could have more specific axiom rules. As one of the main places where assumptions are used are jumps, one might introduce, for example, the rule

$$\frac{P(m,l) = \mathsf{Goto}\ l' \qquad \mathsf{G}(m,l') = (A,B,I)}{\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m,l\, \langle B\rangle\, (I)} \qquad \text{(AX-GOTO)}$$

where the context entry refers to the target of the jump instruction. Thus, the justification of contextual assumptions would be forced to inspect the code block starting at the destination of the jump instruction. However, we would need similar proof rules for at least all instruction forms that may alter the flow of control and all their possible control flow successors. In the case of the JVM bytecode language, this would increase the number of rules significantly as all instructions that may throw an exception would need to be considered, each one with all its handlers.

Solution 2: Alternatively, one could alter the definition of verified programs by requiring that $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m,l'\, \langle B\rangle\, (I)$ be derivable whenever $\mathsf{G}(m,l) = (A,B,I)$ holds and $P(m,l) = \mathsf{Goto}\ l'$. Similarly, ignoring assertion transformers for a moment, one would require $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m,l'\, \langle B\rangle\, (I)$ to be derivable whenever $\mathsf{G}(m,l) = (A,B,I)$ holds and $P(m,l) = \mathsf{If0}\ \mathsf{Eq}\ l'$, and likewise for all cases where an instruction may transfer the flow of control from l to l'. Like the first solution, this proposal appears perfectly possible, but not well suited in the presence of exceptions, as nearly all instruction forms may transfer the flow of control in such a fashion.

Solution 3: Finally, the chosen solution uses two judgement forms to separate the usage of an assumption from its justification. The axiom rule can only be used to derive judgements of the form that is required in the hypothesis of the syntax-directed rules, $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, pc\, \langle B\rangle\, (I)$. In contrast, the definition of verified programs requires us to discharge an assumption $\mathsf{G}(pc) = (A,B,I)$ by exhibiting a proof of $\mathsf{G},\mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$. Such a proof cannot simply consist of an application of the rule AX, but will necessarily end (modulo applications of the rule CONSEQ-F) in a syntax-directed rule. Consequently, the justification of an assumption is forced to inspect the corresponding code block, eliminating the possibility to insert arbitrary (incorrect) assumptions. In order to chain together a sequence of syntax-directed rules, we introduce a further rule, INJECT, that turns a derivation of $\mathsf{G},\mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$ into one of $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, pc\, \langle B\rangle\, (I)$ — but of course, no rule is given for the conversion in the opposite direction.

3.7 Proof of soundness

The soundness property of the program logic guarantees that verified programs are valid. The core of the proof of this property is a soundness result for local judgements $\mathsf{G},\mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$, i.e. a proof that the derivability of such a judgement entails the validity $\models_{\mathsf{Q}} \{A\}\, pc\, \{B\}\, I$.

Traditionally, formalised soundness proofs of program logics have been performed using an auxiliary notion of relativised validity, where a judgement is only required to hold for operational judgements up to a certain derivation height. The soundness proof then proceeds by induction on the axiomatic semantics, where in the case of recursive program structures (procedure calls, jumps) the usage of the height index ensures that hypothetical assumptions become available. General validity is then obtained by universally quantifying over the height index [4, 42].

The soundness proof of the MOBIUS base logic follows a different approach that avoids the notion of relativised validity. Instead, a syntax-directed family of judgement forms is introduced in such a way that for each instruction form there is only one matching judgement form. Each judgement form comes with precisely one introduction rule the hypotheses and side conditions of which are identical to those of the rules presented in section 3.6. Additionally, the rule of consequence is inlined. For example, the rule for the judgement form for basic instructions, $\mathsf{G},\mathsf{Q} \vdash_{\mathsf{Basic}} \langle A\rangle\, m,l\, \langle B\rangle\, (I)$, is

$$\frac{\begin{array}{c} \mathit{basic}(m,l) \qquad \forall s_0\ s.\ A\ s_0\ s \rightarrow I\ s\ s \\ \forall Q.\ \mathsf{Q}(l) = Q \rightarrow (\forall s_0\, s.\ A\ s_0\ s \rightarrow Q\ s_0\ s) \\ \begin{array}{ll} \forall l'. & \mathit{next}_m(l) = l' \rightarrow \\ & \mathsf{G},\mathsf{Q} \vdash \langle \mathit{PRE}(m,l,l',A)\rangle\, m,l'\, \langle \mathit{POST}(m,l,l',B)\rangle\, (\mathit{INV}(m,l,l',I)) \\ \forall l'\ e. & l' \in \mathit{handle}_m(l,e) \rightarrow \\ & \mathsf{G},\mathsf{Q} \vdash \langle \mathit{PRE}^{\mathsf{e}}(m,l,e,A)\rangle\, m,l'\, \langle \mathit{POST}^{\mathsf{e}}(m,l,e,B)\rangle\, (\mathit{INV}^{\mathsf{e}}(m,l,e,I)) \end{array} \\ \forall s_0\, s\, h\, a\, e.\quad \mathit{ExceptionStep}(m,(s,l),(h,a)) \rightarrow \mathit{typeOf}(h,a) = e \rightarrow \\ \quad (\forall l'.\neg l' \in \mathit{handle}_m(l,e)) \rightarrow A\ s_0\ s \rightarrow B\ s_0\ s\ (h,a). \\ \forall\ s_0\ s\ A'\ s_0\ s \rightarrow A\ s_0\ s \qquad \forall\ s_0\ s\ t.A\ s_0\ s \rightarrow B\ s_0\ s\ t \rightarrow B'\ s_0\ s\ t \\ \forall\ s\ r.\ I\ s\ r \rightarrow I'\ s\ r \end{array}}{\mathsf{G},\mathsf{Q} \vdash_{\mathsf{Basic}} \langle A'\rangle\, m,l\, \langle B'\rangle\, (I'),} \quad \text{(INSTR-SD)}$$

and the rule for judgement form for conditionals, $\mathsf{G},\mathsf{Q} \vdash_{\mathsf{If}} \langle A\rangle\, m,l\, \langle B\rangle\, (I)$, is

$$\frac{\begin{array}{c} P(m,l) = \mathsf{If}\ \mathit{comp}\ l' \qquad \forall s_0\ s.\ A\ s_0\ s \rightarrow I\ s\ s \\ \forall Q.\ \mathsf{Q}(l) = Q \rightarrow (\forall s_0\, s.\ A\ s_0\ s \rightarrow Q\ s_0\ s) \\ \mathsf{G},\mathsf{Q} \vdash \langle \mathit{PRE}(m,l,l',A)\rangle\, m,l'\, \langle \mathit{POST}(m,l,l',B)\rangle\, (\mathit{INV}(m,l,l',I)) \\ \forall\, ''.\ \mathit{next}_m(l) = l'' \rightarrow \mathsf{G},\mathsf{Q} \vdash \langle \mathit{PRE}(m,l,l'',A)\rangle\, m,l''\, \langle \mathit{POST}(m,l,l'',B)\rangle\, (\mathit{INV}(m,l,l'',I)) \\ \forall\ s_0\ s\ A'\ s_0\ s \rightarrow A\ s_0\ s \qquad \forall\ s_0\ s\ t.A\ s_0\ s \rightarrow B\ s_0\ s\ t \rightarrow B'\ s_0\ s\ t \qquad \forall\ s\ r.\ I\ s\ r \rightarrow I'\ s\ r \end{array}}{\mathsf{G},\mathsf{Q} \vdash_{\mathsf{If}} \langle A'\rangle\, m,l\, \langle B'\rangle\, (I').} \quad \text{(IF-SD)}$$

In both cases, the rules differ from their counterparts in section 3.6.1 only by the inclusion of the side conditions of the consequence rule.

The following auxiliary lemma may now be proved by induction on the axiomatic semantics.

Lemma 3.7.1 Let P satisfy *VerifiedProg* w.r.t. M, $\mathsf{M}(m) = (\mathsf{S},\mathsf{G},\mathsf{Q})$, and $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m,l\, \langle B\rangle\, (I)$. Then, there is an instruction *ins* such that $P(m,l) = \mathit{ins}$ and $\mathsf{G},\mathsf{Q} \vdash_{\mathsf{kind}} \langle A\rangle\, m,l\, \langle B\rangle\, (I)$ hold, where kind is the unique kind of *ins*, i.e. Basic for basic instructions, If for conditionals etc.

From this, the soundness result can be proved by induction on the operational judgement.

Theorem 3.7.1 (Soundness) Let P satisfy *VerifiedProg* w.r.t. M. Then $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m,l\, \langle B\rangle\, (I)$ implies $\models_{\mathsf{Q}} \{A\}\, m,l\, \{B\}\, I$.

By rule INJECT, Lemma 3.7.1 and Theorem 3.7.1 also hold if $\mathsf{G},\mathsf{Q} \vdash \langle A\rangle\, m,l\, \langle B\rangle\, (I)$ is replaced by $\mathsf{G},\mathsf{Q} \vdash \{A\}\, m,l\, \{B\}\, (I)$.

3.8 Running example

We conclude the exposition of the MOBIUS logic by describing the verification of the example program, based on a formalised verification carried out in a proof assistant. We first give the assertions corresponding to the specification formulae given in Figure 1.3. Then we outline the structure of the verification.

3.8.1 Translation of the specification

The formulae defining the specification of the class `Bill` are collected in Figure 3.1, and are motivated as follows.

All methods of class `Bill` are required to satisfy the field invariant of `sum`. The fact that the content of this field w.r.t. the current object is non-negative prior to executing a method is expressed by the predicate $\mathit{ObjPre}_{\mathit{Bill}}$. The definition of this formula makes use of the auxiliary formula $\mathit{ObjInv}_{\mathit{Bill}}$ and the shorthand $H(a.\mathtt{sum})$ that stands for the Bicolano-defined heap access to the field with signature `sum` of the object at location a in H. The object post-condition is $\mathit{ObjPost}_{\mathit{Bill}}$. Together, $\mathit{ObjPre}_{\mathit{Bill}}$ and $\mathit{ObjPost}_{\mathit{Bill}}$ represent a skeleton that forms the basis of all (partial-correctness) method specifications of methods in class `Bill`.

The abstract method `roundCost` extends the generic method specification by asserting that the result value is bounded by the input value, which in turn may be assumed to be non-negative. The specification consists of three parts, namely a (trivial) method pre-condition, a method post-condition that links initial state and final state by universally quantifying over the location of the parent object and the input value, and a (trivial) method invariant.

Formula	Definition
$\mathit{ObjPre}_{\mathit{Bill}}\ a\ \ (S, H)$	$S(\mathtt{0\%N}) = \mathsf{Ref}(a) \wedge \mathit{ObjInv}_{\mathit{Bill}}\ a\ H$
$\mathit{ObjInv}_{\mathit{Bill}}\ a\ H$	$\exists\ \mathit{sum}.\ H(a.\mathtt{sum}) = \mathsf{Int}(\mathit{sum}) \wedge 0 \leq \mathit{sum}$
$\mathit{ObjPost}_{\mathit{Bill}}\ a\ (v, H)$	$\mathit{ObjInv}_{\mathit{Bill}}\ a\ H$
$\mathit{rcPre}\ a\ x\ (S, H)$	$\mathit{ObjPre}_{\mathit{Bill}}\ a\ (S, H) \wedge S!\mathtt{1\%N} = x \wedge 0 \leq x$
$\mathit{rcPost}\ a\ x\ (v, H)$	$\mathit{ObjPost}_{\mathit{Bill}}\ a\ (v, H) \wedge (\exists\ \mathit{res}.v = \mathsf{Int}(\mathit{res}) \wedge 0 \leq \mathit{res} \wedge \mathit{res} \leq x)$
$\mathit{roundCostSpec}$	$\begin{pmatrix} \lambda\ s_0.\ \mathit{True}, \\ \lambda\ s_0\ t.\ \forall\ a\ x.\ \mathit{rcPre}\ a\ x\ s_0 \to \mathit{rcPost}\ a\ x\ t, \\ \lambda\ s_0\ s. \forall\ a\ x.\ \mathit{rcPre}\ a\ x\ s_0 \to \mathit{True} \end{pmatrix}$
$\mathit{pbPre}\ a\ n\ N\ (S, H)$	$\mathit{ObjPre}_{\mathit{Bill}}\ a\ (S, H) \wedge S(\mathtt{1\%N}) = \mathsf{Int}(n) \wedge 0 < n \wedge H(a.\mathtt{sum}) = \mathsf{Int}(N)$
$\mathit{pbPost}\ a\ n\ N\ (v, H)$	$\mathit{ObjPost}_{\mathit{Bill}}\ a\ (v, H)\ \wedge$ $(\exists\ \mathit{sum}.\ H(a.\mathtt{sum}) = \mathsf{Int}(\mathit{sum}) \wedge 2 * \mathit{sum} \leq 2 * N + n * (n + 1))$
$\mathit{produceBillSpec}$	$\begin{pmatrix} \lambda\ s_0.\ \mathit{True}, \\ \lambda\ s_0\ t.\ \forall\ a\ n\ N.\ \mathit{pbPre}\ a\ n\ N\ s_0 \to \mathit{pbPost}\ a\ n\ N\ t, \\ \lambda\ s_0\ s.\ \forall a\ n\ N.\ \mathit{pbPre}\ a\ n\ N\ s_0 \to \mathit{True} \end{pmatrix}$
$\mathit{loopInv}\ a\ n\ N\ (O, S, H)$	$\exists\ i\ \mathit{sum}.\quad S(\mathtt{2\%N}) = \mathsf{Int}(i) \wedge S(\mathtt{1\%N}) = \mathsf{Int}(n)\ \wedge$ $S(\mathtt{0\%N}) = \mathsf{Ref}(a) \wedge H(a.\mathtt{sum}) = \mathsf{Int}(\mathit{sum})\ \wedge$ $0 < i \wedge i \leq n + 1 \wedge 0 \leq \mathit{sum} \wedge 2 * \mathit{sum} \leq 2 * N + (i - 1) * i$
SpecEL	$\begin{pmatrix} \lambda\ s_0\ s.\ \forall\ a\ n\ N.\ \mathit{pbPre}\ a\ n\ N\ s_0 \to \mathit{loopInv}\ a\ n\ N\ s, \\ \lambda\ s_0\ s\ t.\ \forall\ a\ n\ N.\ (\mathit{pbPre}\ a\ n\ N\ s_0 \wedge \mathit{loopInv}\ a\ n\ N\ s) \to \mathit{pbPost}\ a\ n\ N\ t, \\ \lambda\ s\ r.\ \mathit{True} \end{pmatrix}$
G_{pb}	$[((\mathtt{Bill}, \mathtt{produceBill}), 22) \mapsto \mathit{SpecEL}]$
G_{rc}	emp
Q_{pb}, Q_{rc}	emp
M	$[\mathtt{produceBill} \mapsto (\mathit{produceBillSpec}, \mathsf{G}_{pb}, \mathsf{Q}_{pb}),$ $\mathtt{roundCost} \mapsto (\mathit{roundCostSpec}, \mathsf{G}_{rc}, \mathsf{Q}_{rc})]$

Figure 3.1: Translation of the specification for the class `Bill` into formulae of the base logic

The structure of the main method of interest, `produceBill`, is similar. We first define formulae representing pre- and postconditions. These are parametrised by values that are subsequently universally quantified over in the method specification. In case of the method `produceBill`, the parameters are the location of the parent object, the value of the input parameter, and the initial value of the field `sum`.

The loop invariant asserts that the loop counter, i, is non-negative and bounded by $n+1$ as well as that the current value *sum* of the field `sum` is non-negative and satisfies the functional condition stipulated in Figure 1.3. The association of this loop invariant with instruction 22 (label `entry loop`) is achieved by the definition of a corresponding entry, *SpecEL*, in the table of proof assumptions for the method `produceBill`, G_{pb}. The first component of *SpecEL* ensures that the loop invariant is proved whenever we extract the assumption from G. This is (see below) the case at the end of the verification of the initial code fragment (loop header) and at the end of the verification of the code fragment starting at label 22, in the branch where the loop exit condition is not yet validated after an iteration of the loop. The occurrence of *loopInv* in the second component allows us to exploit the invariant during the verification of the segment with the initial label 22.

The proof context for the abstract method `roundCost` and the tables of further annotations for both methods are defined to be the empty maps. Finally, the method specification table M associates both specifications with their respective methods.

3.8.2 Verification process

The overall aim of the verification process is to establish the verified-program property. Unfolding the definition given in section 3.6.3, we see that we need to justify all entries in the two proof contexts (trivial in the case of G_{rc}, and G_{pb} has only one entry), justify both method specifications (trivial in the case of `roundCost` as this method does not have a method body), and verify the behavioural-subtyping condition (trivial as there are no further classes). Essentially, we are thus left with just two subgoals:

- $$\begin{aligned}\mathsf{G}_{pb}, \mathsf{Q}_{pb} \vdash\ & \{\lambda\ (s_0, s).\ \mathit{produceBillSpec}_1(s_0) \wedge s = \mathit{state}(s_0)\} \\ & (\mathtt{Bill}, \mathtt{produceBill}), 0 \\ & \{\lambda\ (s_0, s, t).\ s = \mathit{state}(s_0) \to \mathit{produceBillSpec}(s_0, t)\} \\ & (\lambda\ (s, r).\ \forall\ s_0.\ s = \mathit{state}(s_0) \to \mathit{produceBillSpec}(s_0, r))\end{aligned}$$

and

- $\mathsf{G}_{pb}, \mathsf{Q}_{pb} \vdash \{\mathit{SpecEL}_1\}\,(\mathtt{Bill}, \mathtt{produceBill}), 22\,\{\mathit{SpecEL}_2\}\,(\mathit{SpecEL}_3)$,

where the indices point out to the respective components of the specification triples *produceBillSpec* and *SpecEL*.

We commence with the second goal. The verification is initiated by an application of the rule INSTR, as the first instruction to be verified is the basic instruction `iload_2`. The first three resulting side conditions may be discharged easily (the fact that the instruction is a basic instruction has already been observed, the other two conditions are trivial by the definition of $\mathit{produceBillSpec}_3$ and Q_{pb}). Hence, only[5] the side condition

$$\mathsf{G}_{pb}, \mathsf{Q}_{pb} \vdash \langle \mathit{PRE}(m, 22, 23, \mathit{SpecEL}_1)\rangle\, m, 23\, \langle \mathit{POST}(m, 22, 23, \mathit{SpecEL}_1)\rangle\, (\mathit{INV}(m, 22, 23, \mathit{SpecEL}_1))$$

remains, where m abbreviates (`Bill`, `produceBill`). Unfolding neither the formulae SpecEL_i nor the assertion transformers, we immediately apply the rule INJECT and then INSTR again, as the instruction at label 23 is another basic instruction. Continuing in this syntax-directed manner, we finally arrive at a goal referring again to the instruction 22, in the positive branch of the conditional. At this point, we apply the rule CONSEQ-T and then the rule AX, thus extracting the specification *SpecEL* from the context and closing the loop. Of the three verification conditions resulting from the application of the consequence rule, the first one requires us to establish SpecEL_1 for the final state of an iteration, assuming the satisfaction of the transformed version of SpecEL_1 for its initial state. Here, the transformation consists of the repeated application of the *PRE* operator and one application of the PRE_{vinv} operator due to the instruction at label 12. In the other side conditions, the implications to be proved are in the opposite direction, due to

[5] In order to simplify the presentation, the verification was carried out using a formalisation that does not include the modelling of exceptions. We thus disregard all issues relating to exceptions in our discussion, and in particular instructions 29 to 31.

the covariance of postconditions and invariants. For example, we have to prove the postcondition for the massaged version of $SpecEL_2$ from the satisfaction of the unmodified form of the same formula w.r.t. the loop-final (and method-final) state. All side conditions may be discharged by unfolding the specification formulae given in Figure 3.1 and by using some basic arithmetic properties. The same applies to the side conditions resulting from the path involving the negative branch of the conditional. As this path does not visit any instruction with an entry in G_{pb}, it continues until the end of method is reached and the side conditions refer to a final state together with the formula $pbPost$.

Having thus verified the code segment with the initial label `22`, we verify the method body (the last remaining subgoal) in a similar way. Starting with an application of the rule INSTR (the instruction at the label `0` is basic), we again obtain two paths. The first path follows the positive branch of the conditional and is terminated by the axiom rule (protected by an application of CONSEQ-T) at the instruction labelled with `22`, while the second path follows the negative branch of the conditional and terminates with the instruction at `28`. An alternative to the above translation of the JML specification is to attach the (translation of the) loop invariant to the instruction at the label `5`. This requires a slight modification of the formula $loopInv$, namely the replacement of the term $i \leq n + 1$ by $i \leq n$, and results in a less efficient verification process as instructions `22` to `31` are visited in both verifications. Under the PCC perspective, the choice to attach formulae to this or that program point (and how to obtain the formulae from JML specifications) is up to the proof producer, as the proof context G would naturally form a part of the producer-communicated certificate that guides the verification process.

Chapter 4

Specification language for bytecode

To express properties of bytecode programs it is convenient to employ a language that uses the notations and keywords the programmer is used to. Bytecode Modeling Language (BML) is a proposal of such a language. BML is based on another specification language — the Java Modeling Language (JML) — which has proved to be useful in specifying realistic industrial examples and finding errors in them. But where JML allows to specify source code level programs, BML is designed to specify bytecode level programs.

To define the semantics of a BML specification in terms of the MOBIUS base logic, as an intermediate step we propose a deeply embedded assertion language which encloses both the syntax of BML and the syntax of assertions in the MOBIUS base logic. This intermediate step allows us to provide a smooth translation between the BML specifications embedded in the Java class files and the post-linking representation of programs in the MOBIUS base logic and Bicolano. The deep embedding of assertions also makes it easier to adapt the MOBIUS base logic to another proving environment and to use meta-level facilities of a theorem prover to ease the verification process.

Section 4.1 presents BML, and it explains how BML specifications can be stored in the class file. Section 4.2 then defines the deep embedding of the assertion language. Finally, section 4.3 shows how the deep embedding is used to map BML specifications into judgements of the MOBIUS base logic.

4.1 The Bytecode Modeling Language

Over the last few years, JML has become the de facto specification language for Java source code programs. Different tools exist that allow to validate, verify or generate JML specifications (see [16] for an overview). 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. [15]). One of the reasons for its success is that JML uses a Java-like syntax. Specifications are written using preconditions, postcondition, class invariants and other annotations, where the different predicates are side-effect-free Java expressions, extended with specification-specific keywords (e.g. logical quantifiers and a keyword to refer to the return value of a method). Other important factors for the success of JML are its expressiveness and flexibility. In Deliverable D1.2 we show how JML can be used (and extended) to express the different security requirements that have been identified to be of interest for the project.

Therefore, to specify Java bytecode programs, we define a variation of JML, which is especially tailored to bytecode. This section presents this specification language, which we call BML, short for Bytecode Modeling Language. BML is designed to be the counterpart of JML at bytecode level. It allows to compile specifications at source code level into specifications at bytecode level, basically by compiling the source code predicates into bytecode predicates and leaving all other parts of the specification unchanged.

Below, we will first present the syntax of BML, discuss its relation with JML and give a specification example. We will also propose a format in which BML specifications can be stored in the appropriate class files (just as JML specifications can be written as special comments in the Java source code).

The semantics of BML specifications can be described in terms of the MOBIUS base logic. The next section proposes a deep embedding of an assertion language. This deep embedding gives a meaning to BML predicates, which is then used in section 4.3 to define a translation of BML specifications into judgements of the MOBIUS base logic.

We assume the reader is familiar with JML, its syntax and its semantics. For a detailed overview of JML we refer to its reference manual [34]. Where necessary, we refer to the appropriate sections of this manual.

```
primary-suffix := ( [expression-list] )
        | [ expression ]
primary-expr ::= #natural                       % reference in the constant pool
        | lv[natural]                           % local variable
        | length(expression)                    % array length
        | cntr                                  % counter of the operand stack
        | st(additive-expr)                     % stack expressions
        | constant | super
        | true | false | this | null
        | (expression)
        | jml-primary

store-ref-expression := store-ref-name [store-ref-name-suffix]
store-ref-name := #natural                      % reference in the constant pool
        |super | this
store-ref-name-suffix := (store-ref-expression)
        | [spec-array-ref-expr]
```

Figure 4.1: Grammar for BML predicates and specification expressions

4.1.1 A short overview of the Bytecode Modeling Language

Syntax of BML Basically, BML has the same syntax as JML with two exceptions:

1. specifications are not written directly in the program code, they are added as special attributes to the bytecode; and

2. the grammar for expressions only allows bytecode expressions.

With respect to the expression syntax, this means concretely that field names, class names etc. are replaced by constants, using the constant pool, while registers are used to refer to local variables. In addition, we can use stack expressions and the stack counter to describe intermediate states of a computation. These will typically only appear in intermediate assertions, we do not use them in method specifications. Finally, we add a special expression `length(`a`)`, denoting the length of array a. Since the source code expression a`.length` is compiled into a special bytecode instruction `arraylength`, we also need a special specification construct for this at bytecode level.

BML contains equivalent constructs for all specification constructs of JML Level 0 (see [?, §2.9]), which defines the features that should be understood and checked by all JML tools. In addition, it contains several constructs from JML level 1, that we find important to be able to write meaningful specifications for the example applications studied in the MOBIUS project. These constructs are:

- static invariants;
- object and static history constraints; and
- loop variants (using the `decreasing` keyword).

At the moment, the use of pure methods is not part of the BML grammar, as there is still ongoing research on the exact semantics of method calls used in specifications. However, we believe that if the theoretical issues have been settled, eventually the MOBIUS tool set should support this, both at source code and at bytecode level. Finally, experiences with verification of realistic case studies have shown that it is beneficial to have a special clause `loop-modifies`, which is specified together with the loop invariant. This clause specifies which variables may be modified by a loop (as an `assignable` clause does for a method). This `loop-modifies` clause allows to write concise specifications, and to efficiently generate proof obligations using a weakest precondition calculus.

Since the bytecode and BML specifications are two separate entities, they should be parsed independently. Concretely this means that the grammar of BML is similar to the grammar of type specifications, method specifications and data groups of JML [?, §A.5, A.6, A.7], restricted to the constructs in JML level 0, plus the constructs of JML level 1 mentioned, but with the changes to the grammar for predicates and

```
{| requires lv[1] > 0
   ensures #24 <= \old(#24) + lv[1] * (lv[1] + 1) / 2 |}
 0 iconst_1
 1 istore_2            //initialisation of i, at location 2, to 1
 2 goto 22 (+20)
 5 aload_0
 6 aload_0
 7 getfield #24 <Bill.sum>
10 aload_0
11 iload_2                            //i as a parameter to roundCost
12 invokevirtual #26 <Bill.roundCost> //roundCost is invoked
15 iadd
16 putfield #24 <Bill.sum>
19 iinc 2 by 1         //i++
{| loop_invariant 0 < lv[2] && lv[2] <= lv[1] + 1 &&
                  0 <= #24 && #24 <= \old(#24) + (lv[2] - 1) * lv[2]/2 |}
entry loop:
22 iload_2
23 iload_1             //the parameter n is at location 1
24 if_icmple 5 (-19)   //the loop exit condition is checked
27 iconst_1
28 ireturn
29 astore_3            //the code to handle exception
30 iconst_0
31 ireturn
```

Figure 4.2: Bytecode together with BML specification for method `produceBill` in class `Bill`

specification expressions, as mentioned above. Figure 4.1 displays the most interesting part of this grammar for predicates and specification expressions, defining the syntax for primary expressions, primary suffixes, store-ref expressions and store-ref expressions (see appendix C for a short explanation of the syntax notation and the full grammar of predicates and specification expressions). Primary expressions, followed by zero or more primary suffixes, are the most basic form of expressions, formed by identifiers, bracketed expressions etc. Store ref expressions (followed by zero or more store ref suffixes) are the expressions that can be used in an assignable clause.

As mentioned above, all identifiers are replaced by references to the constant pool (a number, preceded by the symbol #) or to local registers. The local register `lv[0]` of a non-static method always contains the implicit argument `this`, the other registers contain the parameters and the local variables declared inside a method body. As explained above, we add special keywords to be able to express properties about the length of an array, the current stack counter (`cntr`), and to refer to an element on the stack (`st(`e`)`, where e is some arithmetic expression). In BML, a field access is written as a function application. For example, suppose we have the source code qualified expression `obj.f`, where `obj` is the first parameter of a method, and `f` is a field of this object. This becomes `#`n`(lv[1])` in BML, where n is the index in the constant pool to the field constant reference denoting the field `f`, while `lv[1]` is the register in the local variable array in which the parameter `obj` is stored. Therefore the grammar for primary-suffixes and store-ref-name-suffixes does not provide any grammar for qualified expressions.

In JML, many special keywords are preceded by the symbol \, to ensure that they will not clash with variable names. For BML, we do not have to worry about this: all variable names are replaced by references to the constant pool or local variable registers. Therefore, the new keywords are written without a special preceding symbol. However, for convenience, we keep the symbol for keywords that are also JML keywords.

Finally, statement annotations are described as a special attribute, mapping line numbers to annotations. To parse these annotations, we reuse the relevant parts of the grammar for statements and annotation statements [?, §A.9].

Type checking of the specification can be done in the obvious way, using the type information stored in the constant pool.

Example To show a typical BML specification, Figure 4.2 presents the BML version of the specification of method `produceBill` of the running example (see Figure 1.3 on page 16 for the JML specification).

Notice that the field `sum` has been assigned the number 24 in the constant pool. Further, `lv[1]` denotes the parameter `n`, and `lv[2]` denotes the local variable `i`.

The class invariant gives rise to the following BML specification (stored in the class file as a special user-specific attribute, as explained below):

```
invariant:  #24 >= 0
```

Evaluation of BML expressions When defining the evaluation of BML expressions, a subtle point that has to be taken into account is the fact that at bytecode level no explicit boolean values are used, they are encoded as integers (but variables can still be of type `boolean` — this information is used by the BCV). Thus, to make sure that expressions such as `\result == \exists i.  i >= 0` are correctly evaluated, the evaluation of the quantified expression is wrapped up by a conditional function, returning 1 if the condition is true, 0 otherwise.

4.1.2 Encoding BML specifications in the class file format

To store BML specifications together with the bytecode that it specifies, we need a way to encode them in the class file format. We do this using so-called user-specific attributes for Java class files.

Recall that a class file contains all the information related to a single class or interface, i.e. its class name, interfaces implemented by the class, its super class and the methods and fields it declares. The Java Virtual Machine Specification [38] prescribes the mandatory elements of the class file: the constant pool, the field information and the method information. The constant pool is the table which is used to construct the runtime constant pool upon class or interface creation. This will serve for loading, linking and resolution of references used in the class. The JVM specification allows to add user-specific information to the class file ([?, §4.7.1]), by defining user-specific attributes, following the structure prescribed by the JVM specification. We use these to encode the BML specifications. For each class, we add the following global user-specific attributes:

- lists of the model and ghost fields used in the specification[1]; if a model or a ghost field is dereferenced in the specification, then a constantFieldRef is added to the constant pool as the Java compiler would do for any dereferenced Java field;
- a list of the class invariants (both static and object); and
- a list of the history constraints (both static and object).

In order to describe the binary format of BML attributes we follow The Java Virtual Machine Specification [?, §4] and use a C-like structure notation. The lists of model and ghost fields have the following definition:

```
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];
}
```

This should be understood as follows: the name of the attribute is given as an index into the constant pool. This constant pool entry will be representing a string (either `"Model_Field"` or `"Ghost_Field"`). Next we have the length of the attribute, which should be 2 + 6*fields_count (the number of fields stored in the list). The fields table then stores all ghost and model fields. For each field we store its access flags (e.g. `public` or `private`), and the name index and descriptor index, both referring to the constant pool. The first must be a string, representing the (unqualified) name of the variable, the latter is a field descriptor,

[1]Note that the current version of the MOBIUS base logic does not support model and ghost fields, however, this will be included in Task 3.2.

containing e.g. type information. The information as u2 and u4 specifies the size of the attribute, 2 and 4 bytes, respectively.

In a similar way, we specify the format for the attributes containing the list of class invariants and history constraints. The type of invariants and history constraints is specified by the type entry: when it is 1 the invariant (or history constraint) is defined over objects, when it is 0 the invariant (or constraint) is static.

```
JMLClassInvariant˙attribute {
  u2 attribute˙name˙index;
  u4 attribute˙length;
  u2 invariant˙count;
  { u1 type;
    formula invariant;
  } invariants[invariant˙count];
}
```

```
JMLHistoryConstraints˙attribute {
    u2 attribute˙name˙index;
    u4 attribute˙length;
    u2 history˙constr˙count;
    { u1 type;
      formula constraint;
    } history˙constr[history˙constr˙count];
}
```

The JVM specification prescribes that the table with method information at least contains the code of each method. We add attributes for the method specification, a table with set statements, a table with assert statements, a table with assume statements and a table with loop specifications. The attribute with the lightweight behaviour specifications is formatted as follows (heavyweight behaviour specifications are handled similarly):

```
JMLMethod˙attribute {
    u2 attribute˙name˙index;
    u4 attribute˙length;
    formula requires˙formula;
    u2 spec˙count;
    { formula spec˙requires˙formula;
      u2 assignable˙count;
      formula assignable[assignable˙count];
      formula ensures˙formula;
      u2 exsures˙count;
      { u2 exception˙index;
        formula exsures˙formula;
      } exsures[exsures˙count];
    } spec[spec˙count];
}
```

The global requires formula is the disjunction of all preconditions in the different specification cases of the method. For each specification case, we then have a precondition (spec˙requires˙formula), a list of assignable expressions, a postcondition (ensures˙formula) and a list of exceptional postconditions (stored in the exsures attribute). If a clause is not explicitly specified, its default value will be stored here. Notice that for each list of elements we get two attributes: one to store the number of elements, and one attribute actually containing the elements.

The tables with set, assert and assume statements are very similar. For each statement we use index to denote the point in the bytecode to which the statement is associated. For the set statement, expression e1 is a ghost variable, e2 denotes the expression that will be assigned to e1. For the assert and assume statements, the formula predicate is the predicate that is supposed to hold at this point in the program execution. We only give the format for the assert statement table here, the assume statement table is similar.

```
Set˙attribute {
    u2 attribute˙name˙index;
    u4 attribute˙length;
    u2 set˙count;
    { u2 index;
      expression e1;
      expression e2;
    } set[set˙count];
}
```

```
Assert˙attribute {
  u2 attribute˙name˙index;
  u4 attribute˙length;
  u2 assert˙count;
  { u2 index;
    formula predicate;
  } assert[assert˙count];
}
```

Finally, loop specifications consist of the following elements: an index to the bytecode instruction that corresponds to the entry of the loop, a list of variables that may be modified by the loop, a loop invariant, and a decreases clause, which is the loop variant, i.e. the expression that allows to prove termination of the loop. If the specification does not contain a loop variant, we indicate this, using a special tag for the decreases clause. This gives the following attribute format.

```
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];
}
```

4.2 Deep embedding of assertions

In this section we introduce a language of assertions which encompasses both BML predicates and the syntax of assertions of the MOBIUS base logic, as presented in chapter 3. Expressions defined by the grammar of our language are called deep or deeply embedded, whereas higher order logic expressions of the MOBIUS base logic encoded in a theorem prover are called shallow.

We have formalised the deep expression language as a datatype in Isabelle [43], and our presentation of the grammar and derived constructs is influenced by Isabelle's style of definitions. We will use syntax translations whenever it seems reasonable.

4.2.1 Motivation

There are at least three reasons for which the language of deep expressions has been designed. First, the deep embedding language serves as a bridge between BML and the (shallow) MOBIUS base logic, formalised in a theorem prover. If one wishes to verify a BML specification, one can translate this specification into a specification in the MOBIUS base logic and then verify this specification using the proof rules of the logic. The deep embedding makes this translation smooth, since it encompasses both constructs from BML and the MOBIUS base logic. Given a BML specification and a constant pool, one obtains the deeply embedded assertions using a simple, almost one-to-one, translation procedure. The semantic evaluation functions, as we will define, then map the deeply embedded assertions into higher order logic expressions for the shallow base logic.

Second, the deep embedding language is richer than BML. In particular, it allows to construct expressions over stacks, stores, and heaps as entities, whereas in BML one accesses them by single elements — the n-th element in the case of stacks, variables in the case of stores and fields in the case of heaps. For instance, in the deep embedding language one can express that the current heap is not empty by `Not (Eq currentHeap emptyHeap)`. Such a property cannot be expressed in BML. Moreover, we plan to extend our deep embedding language with constructs such as a fixed point operator, that will enable us to express the reachability relation for addresses in a given heap.

Third, the deep embedding approach can help for on-device proof checking. So far we have not defined a language of proof rules, that is we do not consider a deeply embedded version of the MOBIUS base logic. However, developing such a program logic might be of interest for on-device proof checking. This will require a "small" system of proof rules, designed especially for MOBIUS verification, rather than installation of a general purpose theorem prover. The deep embedding language is the first step towards a standalone MOBIUS prover.

The rest of this section describes the syntax and semantics of deep expressions, respectively. We also show how the predicates in the BML specification of the running example (see Figure 4.2) are mapped into assertions of the deep embedding language.

When defining the syntax, we first discuss typing of expressions and the way we refer to state components. The latter are first related with the MOBIUS base logic, by recalling the definition of the different kinds of states (initial, at-a-program-counter, terminal) and assertions, and then defined in the deep embedding language. We also discuss differences in the base logic and BML interpretations of "temporal" operations, such as the `\old` operator and lookup of stacks, stores, and heaps. We show how we handle this in the deep embedding language. Then we define the substitution-into-state-parameter operation, which models application of an assertion to state parameters. Finally, we define deep assertion transformers which are the syntactic counterparts of the assertion transformers of the shallow base logic.

The semantics of deep expressions is defined as an evaluation function. We define the main evaluation function and its versions for different sorts of assertions: pre- and postconditions, invariants, etc. We prove that evaluation commutes with substitution. We also briefly sketch how one could prove soundness of a program logic with deep assertions.

4.2.2 Syntax of the deep embedding language

Types To design a language of deep expressions, one needs to take into account the types coming from the MOBIUS base logic, such as booleans, integers, operand stacks, stores, heaps, addresses, objects, program values, return values (which are either program values or references to exception handlers). In the deep embedding language one could have considered a collection of mutually recursive grammars, where each grammar defines expressions of a particular type above. However, large mutually recursive definitions handicap proof search.

Instead, we consider one main grammar of the language. It defines raw deep expressions which are either of one of the types above (and thus well-typed) or ill-typed. The grammar does not perform any type-checking service itself. Well-typedness of an expression is checked on the semantical level by the evaluation function, which only returns Some value of the appropriate type if the expression is well-typed, and returns not defined, i.e. None, otherwise (see section 4.2.3). Additionally, we can also define a pure type-checking function for expressions and contexts, which is an abstraction of the evaluation function.

The types above are referred to not only in the semantics of expressions but also in the syntax. In particular, the universal and existential quantifiers in the language have to be typed, otherwise it is not possible to define their evaluation correctly. Therefore, we define an auxiliary grammar for type tags:

$$\begin{array}{rcl} \mathit{type\_tag} & := & \mathit{boolean} \mid \mathit{integer} \mid \mathit{address} \mid \mathit{object} \\ & \mid & \mathit{val} \mid \mathit{stack} \mid \mathit{store} \mid \mathit{heap} \\ & \mid & \mathit{exception} \mid \mathit{retval} \mid \mathit{class} \mid \mathit{lab} \end{array}$$

State components Recall that an assertion in the MOBIUS base logic is a predicate over states and a state may be of one of three sorts:

- initial $s_0 \in \mathit{InitState} = \mathcal{S} \times \mathcal{H}$,
- local $s \in \mathit{LocalState} = \mathcal{O} \times \mathcal{S} \times \mathcal{H}$,
- terminal $t \in \mathit{TermState} = \mathcal{V}_r \times \mathcal{H}$,

where $\mathcal{S}$, $\mathcal{H}$, $\mathcal{O}$, $\mathcal{V}_r$ denote state components — stores, heaps, operand stacks — and return values, respectively. Assertions can be of any of the following types:

- local assertions A of type $\mathcal{A} = (\mathit{InitState} \times \mathit{LocalState}) \to \mathit{Prop}$,
- postconditions $B \in \mathcal{B} = (\mathit{InitState} \times \mathit{LocalState} \times \mathit{TermState}) \to \mathit{Prop}$,
- invariants $I \in \mathcal{I} = (\mathit{LocalState} \times \mathit{LocalState}) \to \mathit{Prop}$,
- method preconditions $R \in \mathit{MethPre} = (\mathit{InitState}) \to \mathit{Prop}$,
- method postconditions $T \in \mathit{MethPost} = (\mathit{InitState} \times \mathit{TermState}) \to \mathit{Prop}$,
- method invariants $\Phi \in \mathit{MethInv} = (\mathit{InitState} \times \mathit{LocalState}) \to \mathit{Prop}$.

Thus, tuples of state components are parameters of assertions. To present these parameters in the deep embedding language we introduce standard names for state components:

$$
\begin{array}{rcll}
\mathit{state\_component} & := & \mathit{st0} \mid \mathit{h0} & \text{(initial state)} \\
 & \mid & \mathit{ops1} \mid \mathit{st1} \mid \mathit{h1} & \text{(state at some p. counter)} \\
 & \mid & \mathit{ops11} \mid \mathit{st11} \mid \mathit{h11} & \text{(state at some next p. counter)} \\
 & \mid & \mathit{rv} \mid \mathit{th} & \text{(terminal state)}
\end{array}
$$

where *st0* and *h0* denote the initial store and heap at an entry point of a given method; *ops1*, *st1*, and *h1*, are the operand stack, store and heap at some program counter inside the method body; *ops11*, *st11*, and *h11* are the operand stack, store and heap at a next program counter; and *rv*, and *th* are the return value and the heap at a return point of the method.

Note, that when a state component occurs in a base logic assertion as λ-bound, in the deep embedding it will be replaced, according to its position in the binding, by the standard name from the grammar above, while the λ-binder itself will be ignored.

As we will show below, this allows us to define a deep version of the assertion transformers (as presented in section 3.5). This will be a major ingredient for defining a deep version of the MOBIUS base logic, as we plan to do as future work[2].

Logical variables Logical variables are used when quantifiers are introduced. Note, that since BML specifications do not have free logical (non-program) variables, they are mapped onto deep expressions without free logical variables.

We use de Bruijn levels to represent them. With this scheme the variable bound by the outermost quantifier is denoted by 0. A variable bound by the quantifier which is in the scope of the outermost one, is called 1, etc. For instance, the formula $\forall\ x.\ (\exists\ y.\ x > y)\ \wedge\ (\exists\ z.\ x < z)$ is presented as $\forall\ 0.\ (\exists\ 1.\ 0 > 1)\ \wedge\ (\exists\ 1.\ 0 < 1)$. The advantage that this scheme has over the more popular of de Bruijn's schemes (de Bruijn indices) is that the variable bound by a given quantifier is represented in the same way wherever it appears in the scope of this quantifier. The disadvantage of de Bruijn levels (which de Bruijn indices do not have) is that the bound variables in a term may need to be renumbered when it is substituted into a different term.[3]

In the grammar of the language logical variables are declared by the constructor `LV`nat. We have not defined substitutions for logical variables yet. However, one will need them once we extend the language of deep expressions to a deep logic.

Evaluation time of expressions The MOBIUS base logic has more general expressions over stacks, stores and heaps than BML. In BML the operators over stacks and stores always refer to the current stack, store or heap. In contrast, operands — stacks, stores and heaps — in the MOBIUS base logic are arbitrary, in particular empty, or constructed from others.

In addition, the base logic and BML also have different approaches to express evaluation depending on time, i.e. looking up the content of a variable, heap, or stack either "now", in the current state, or "before", in some earlier state. The MOBIUS base logic approach may be called an absolute time scale, since one has direct access to the current, previous and initial state via parameters ops1, h1, s0, h0 etc.

BML has a special construct to refer to earlier time ticks: `\old`, which indicates that an expression is evaluated in the initial state of the method (as mentioned at the beginning of the previous chapter).

We use the MOBIUS base logic point of view, but to make the translation from BML predicates into the language of deep expressions transparent, the language contains an `Old` constructor, and explicit constructors `currentStack`, `currentStore` and `currentHeap`. The latter allow us to define several derived lookups that correspond to the BML semantics. The evaluation of these constructors will depend on an argument representing the time (indicating in which state is the expression to be evaluated). Based on the absolute time scale of the MOBIUS base logic, we define the following grammar for time:

$$\mathit{time} := \text{init} \mid \text{middle\_1} \mid \text{middle\_11} \mid \mathit{ret}$$

[2]Eventually one might even be interested in a full deeply embedded logic, that is a logic where proof rules are syntactic objects themselves.

[3]See, for instance, http://math.boisestate.edu/∼ holmes/babydocs/node26.html for more detail.

The full grammar of the deep expression language Let java-literal denote strings of symbols representing boolean and integer constants in Java. The grammar of the language of deep expressions is defined below. Note that this is an Isabelle style grammar definition, where n-ary constructors are used to define each language construct.

```
expr  :=  (* Boolean and integer constants *)
       |  TT | FF
       |  Number java-literal
       |  ProgramVariable java-literal
       |  ClassName java-literal
       |  FieldName java-literal

          (* Constants for lambda-bound in the MOBIUS base logic stacks, stores, heaps *)
       |  MV state_component

          (* Logical Variables *)
       |  LV nat

          (* Operations and relations over integers *)
       |  Unary UnOp expr
       |  Binary BinOp expr expr
       |  Rel BinRel expr

          (* Polymorphic equality *)
       |  Eq expr expr

          (* Logical connectives *)
       |  Not expr | And expr expr
       |  Or expr expr | Impl expr expr
       |  ForAll type_tag expr | Exists type_tag expr

          (* Constructs arising from BML *)
       |  Typeof expr | Old expr
       |  Fresh expr | This

          (* Constructs for adaptation to BML *)
       |  currentStack | currentStore  | currentHeap

          (* Operations over stacks, stores, heaps *)
       |  Cntr expr | LookUpStack expr expr | Push expr expr
       |  emptyStore | LookUpStore expr expr | UpdateStore expr expr expr
       |  emptyHeap | LookUpHeap expr expr expr | UpdateHeap expr expr expr expr expr

          (* Get the current program counter *)
       |  currentLab

          (* Get the next program counter *)
       |  nextLab

          (* The MOBIUS base logic NormalStep (M, (l, (ops, s, h)), (l1, (ops1, s1, h1))) *)
       |  NormalStepL Method expr expr expr expr expr expr expr expr

          (* Construtors for program and return values *)
       |  nulladdr | ProgVal expr | noexc | ExVal expr
```

In addition, we have the following auxiliary grammars:

```
 UnOp   :=  uminus
 BinOp  :=  plus | minus | mult | div | remainder
 BinRel :=  less | greater | lesseq | greatereq
```

Note that the language constructs marked as arising from BML are specific for BML. The operations and relations and logical connectives are in the intersection of the MOBIUS base logic and BML.

There are several constructs in BML the semantics of which is less general than the semantics of the related deep embedding constructs. In BML expressions such as `cntr`, `st(`e`)` and lookup of fields or local variables is always evaluated w.r.t. the current stack, store and heap. In contrast, in the deep embedding language, the related constructors `Cntr`, `LookUpStack`, `LookUpStore` and `LookUpHeap` can be evaluated w.r.t. any stack, store or heap — this is an explicit parameter of the constructor.

For the sake of transparency, we define these BML constructs as abbreviations in the language of deep expressions, using the corresponding deep constructors:

- `CntrBML` := `Cntr` (`currentStack`),
- `LookUpStackBML` exnat := `LookUpStack` (`currentStack`) exnat,
- `LookUpStoreBML` exx := `LookUpStore` (`currentStore`) exx,
- `LookUpHeapBML` exa f := `LookUpHeap` (`currentHeap`) exa f.

For example, the translation "BML × Constant_pool ⟶ Deep Expressions" maps the BML local variable lookup of x to `LookUpStoreBML` (`ProgramVariable` x), and the field lookup of sum of class `Bill` to `LookUpHeapBML This` (`FieldName` Bill.sum).

Recall the different kinds of shallow base logic assertions: local assertions, postconditions and invariants, method preconditions, postconditions and invariants. The syntactic structure of expressions of these sorts and their semantic evaluation functions are essentially the same, but they differ in the types of state components to which they are applicable. We define these different sorts of assertions on top of the main grammar:

exprLocAssn	:=	`LocAssnTag` expr
exprPost	:=	`PostTag` expr
exprInv	:=	`InvTag` expr
exprMethPre	:=	`MethPreTag` expr
exprMethPost	:=	`MethPostTag` expr
exprMethInv	:=	`MethInvTag` expr

Evaluation functions for the assertion kinds above will be defined using one main evaluation function. We also introduce explicit tags for the different kinds of assertions:

assn_tag := `locAssn` | `post` | `inv` | `methPre` | `methPost` | `methInv`

Substitutions In the base (shallow) logic assertions can be applied to state components. We use a syntactic substitution of state components to represent this in the language of deep expressions. The substitution function subst: expr → state_components → expr → expr is defined as a primitive recursive function in the usual way.

In addition, we need an auxiliary function incDeBruijn: expr → nat → expr, which when applied to a deep expression deepA and natural number n increments the de Bruijn levels of the logical variables in deepA by n.

For example, the shallow expression

$$\lambda\ (s_0,\ h_0)\ (ops_1,\ s_1,\ h_1).\ \exists\ h.\ A\ (s_0,\ h)\ (ops_1,\ s_1,\ h_1)\ \longrightarrow (...)$$

has the following deep representation:

`Exists` heap (`Impl` (subst (incDeBruijn deepA 1) h0 (`LV 0`)) (...)).

As mentioned above, there is no explicit λ-binding of the state components in the deep embedding language. One logical variable of type heap is introduced and bound by the existential quantifier. This variable automatically gets the name `LV 0` as it is the outermost bound variable. It corresponds to h in the shallow example. To model the application of h to the expression A, h0 is substituted by `LV 0` in the deep expression deepA. The introduction of this fresh variable requires increments of all the variables in the original expression deepA by 1.

Deep assertion transformers In the shallow base logic, assertion transformers are defined to relate assertions for adjacent instructions. This allows the rules of the logic to be formulated so that assertions in the conclusions are unconstrained, i.e. a rule always can be applied directly to derive a judgement in a backward-style manner. Judgements occurring as premises involve assertions that are notationally constrained, and relate to the conclusions' assertions via uniform constructions.

These assertion transformers can be encoded in the deep expression language. For example, the assertion transformer PRE: Method $\times$ Label $\times$ Label $\times \mathcal{A} \to \mathcal{A}$, which relates an assertion to the weakest precondition from a judgement of the operational semantics, is defined as follows (spelling out the complete definition):

$$\begin{aligned} PRE(m,\ l,\ l',\ A)\big((s0,\ h0),\ (ops1,\ s1,\ h1)\big) \equiv\ & \exists\ (ops,\ s,\ h). \\ & \text{NormalStep}\,(m,\ (l,\ (ops,\ s,\ h)), (l',\ (ops1,\ s1\ ,h1))) \wedge \\ & A\,((s0, h0),\ (ops, s, h)) \end{aligned}$$

In the language of deep expressions it is defined as follows:

```
deepPRE : Method → exprLocAssn → exprLocAssn

deepPRE MB (LocAssnTag expr) =
    (LocAssnTag (Exists stack (Exists store (Exists heap
                (And (NormalStepL MB currentLab (LV 0) (LV 1) (LV 2)
                                    nextLab (MV ops1) (MV s1) (MV h1))
                     (subst (subst (subst (incDeBruijn expr 3) ops1 (LV 0)) s1 (LV 1)) h1 (LV 2))
                )))))))
```

4.2.3 Semantics of the deep embedding language

To define the meaning of expressions in the language of assertions in terms of the MOBIUS base logic, we define appropriate evaluation functions. The evaluation function for an expression of the main grammar has the following arguments:

- an assertion type,
- a current program counter,
- the next program counter,
- a current time point,
- different state components,
- a de Bruijn logical environment.

The de Bruijn logical environment le of type env is a list of logical values of the sum type LVal, where:

$$\begin{array}{rcl} \text{LVal} & := & \text{BO bool} \mid \text{INTG int} \mid \text{AD (Addr option)} \mid \text{OB object} \\ & \mid & \text{PV } \mathcal{V} \mid\ \text{OS } \mathcal{O} \mid \text{ST } \mathcal{S} \mid \text{HP } \mathcal{H} \\ & \mid & \text{EC Ref} \mid\ \text{RV } \mathcal{V}_r \mid \text{CL class} \mid \text{LB Label} \\ & \mid & \text{JV java-literal} \end{array}$$

where EC Ref denotes a reference to an exception handler.

Such an environment represents the list of values for the free variables, where the head of the list is the value of the variable `LV 0`, the next element is the value of `LV 1`, etc.

The main evaluation function

$$\begin{array}{rl} \text{eval} & : \ \text{expr} \to \text{assn\_tag} \to \text{Label} \to \text{Label} \to \text{time} \to \mathcal{S} \to \mathcal{H} \to \mathcal{O} \to \mathcal{S} \to \mathcal{H} \to \mathcal{O} \to \\ & \quad \mathcal{S} \to \mathcal{H} \to \mathcal{V}_r \to \mathcal{H} \to \text{env} \to \text{(LVal option)} \end{array}$$

is defined in the usual way as a primitive recursive function on expressions. We will give several of its defining equations, to illustrate the most interesting aspects of its definition. For boolean constants the evaluation is trivial:

```
eval TTassnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
    (Some (BO True))
```

The explanation about state components above gives the intuition for the evaluation of the state component constructors:

```
eval (MV c) assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
    case c of
          s0 ⇒  (case assnType of
                   inv ⇒ None
                 | _ ⇒ Some (ST S0 ))
          ...
       |  th ⇒  (case assnType of
                   post ⇒ Some (HP TH )
                 | methPost ⇒ Some (HP TH )
                 | _ ⇒ None )
```

The evaluation of logical variables is straightforward, one should just keep in mind the order of logical variables in the environment (see also the evaluation of the universal quantifier later):

```
eval (LV i) assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
    (if  (i¡(length le ))
     then  (Some (le[i] ))
     else None )
```

The evaluation of the `currentLab` constructor is trivial:

```
eval currentLab assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
     Some (LB l )
```

The `NormalStepL` constructor is evaluated using the NormalStep relation from the MOBIUS base logic, that is used to define the assertion transformers in section 3.5.

```
eval (NormalStepL Mb exl exl' exos exs exh exll exoss exss exhh )
    assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
        (case (eval exl assnType l l' tick S0 H0 OPS1 S1 H1
                          OPS11 S11 H11 RV TH le ) of Some vl ⇒
            (case vl of LB l ⇒
                ...
                (case (eval exos assnType l l' tick S0 H0 OPS1 S1 H1
                              OPS11 S11 H11 RV TH le ) of Some vos ⇒
                  (case vos of OS STACK ⇒
                    (case (eval assnType exs l l' tick S0 H0 OPS1 S1 H1
                              OPS11 S11 H11 RV TH le ) of Some vst ⇒
                      (case vst of ST STORE ⇒
                        ...
                        (Some (BO (NormalStep  (Mb, (l, (STACK, STORE , HEAP )),
                                               (l', (STACK1, STORE1, HEAP1 ))))))))
```

An expression with the universal quantifier `ForAll` tp ex evaluates to Some boolean value if for all well-typed values of the bound variable the expression expr evaluates to some boolean value. Otherwise `ForAll` tp ex evaluates to None. The expression `ForAll` tp ex evaluates to Some (BO True), if and only if for all well-typed values of the bound variable the expression expr evaluates to Some (BO True):

```
eval (ForAll tp ex )assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
    if (∀ lval. ∃ b.((get_type lval = tp ) ⟶
        (eval ex assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH (le@[lval] ) =
            Some (BO b ))))
then (if (∀ lval.(get_type lval = tp ) ⟶
        (eval ex assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH (le@[lval] ) =
            Some (BO True ))))
    then Some (BO True ))
    else Some (BO False )))
else None
```

The auxiliary function get_store: time → Store → Store → Store → (Store option) shows how the time parameter tick is used.

```
get_store init S0 S1 S11 = Some S0
get_store middle_1 S0 S1 S11 = Some S1
get_store middle_11 S0 S1 S11 = Some S11
get_store ret S0 S1 S11= None
```

This function is used, for instance, to evaluate `currentStore`:

```
eval currentStore assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 W TH le =
    (case(get_store tick S0 S1 S11 )of
        Some S ⇒ Some (ST S )
    |   None ⇒ None)
```

Thus, nullary constructors `currentStack`, `currentStore` and `currentHeap` evaluate to the corresponding state parameters from the list $S0, \ldots, TH$ depending on the value of the argument tick.

Next we show how the time argument is used to evaluate `Old` ex; the recursive call ensures that ex is evaluate in the initial state:

```
eval (Old ex )assnType l l' tick S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le =
    eval ex assnType l l' init S0 H0 OPS1 S1 H1 OPS11 S11 H11 RV TH le
```

The evaluation functions for the different assertions are defined via the main evaluation function. For instance, for local assertions we define:

```
eval_localassn: exprLocAssn → Label → Label → (LocalAssn option)

eval_localassn (LocAssnTag assn) l l' =
    if (∀s h opss ss hh. ∃ b.
            (eval assn locAssn l l' middle_1 s h opss ss hh [] [] [] (RV (RVal nullref )) [] [] =
              Some (BO b )))
    then Some (λ(s, h) (opss, ss, hh).
                    (case eval assn locAssn l l' middle_1 s h opss ss hh
                                [] [] [] (RV (RVal nullref ))[] []) of
                            Some (BO b) ⇒ b
                        |   _ ⇒ False))
    else None
```

Towards soundness Similarly to the proof rules of the shallow logic with judgements $\{A\}\ C, M, l\ \{B\}\ (I)$, one might define rules where preconditions, postconditions and invariants are in the language of assertions: $\{deepA\}\ C, M, l\ \{deepB\}\ (deepI)$. These rules would involve the deep assertion transformers as deepPRE, defined above, to describe the premises of the proof rules.

We present here a possible way the soundness theorem could be proved for program logic rules with deeply embedded Hoare quadruples. First, evaluate the deep triples in a given rule to their shallow counterparts.

Suppose for instance, that to prove the rule for basic bytecode instructions one has

eval_localassn deepA = Some A
eval_post deepB = Some B
eval_inv deepI = Some I

for the conclusion and

eval_localassn (deepPRE MB deepA) = Some A'
eval_post (deepPOST MB deepB) = Some B'
eval_localassn (deepINV MB deepI) = Some I'

for the hypothesis. Next, apply the shallow rule in backward manner, where $\{A\}\ C,\ M,\ l\ \{B\}\ (I)$ is in the conclusion; this rule can be applied if one has

$$\{\text{PRE (M, l, l', A)}\}\ C,\ M,\ l'\ \{\text{POST (M, l, l', B)}\}\ (\text{INV (M, l, l', I)})$$

as a hypothesis. To obtain this hypothesis one proves the following three lemmas:

PRE (M, l, l', A) $\equiv$ A'
POST (M, l, l', B) $\equiv$ B'
INV (M, l, l', B) $\equiv$ I'

Alternatively one can use the rule of consequence, in which case it would suffice to prove the following weaker statements:

PRE (M, l, l', A) $\to$ A'
B' $\to$ POST (M, l, l', B)
I' $\to$ INV (M, l, l', B)

4.2.4 Running example

The predicates in the BML specification of the running example (see Figure 4.2) can be translated into expressions in the deep expression language. The method specification of the method `produceBill` reads as follows:

```
{| requires lv[1] > 0
   ensures #24 <= \old(#24) + lv[1] * (lv[1] + 1) / 2 |}
```

Recall, that the BML local variable `lv[0]`, denoting `this`, and `lv[1]`, denoting the method parameter, are mapped onto expressions lv0, lv1 respectively which are the abbreviations for (`ProgramVariable` lv[0]) and (`ProgramVariable` lv[1]). The translation reads the entry `#24` from the given constant pool and finds that it corresponds to the field `sum` of the class `Bill`.

So, the BML `requires`-clause is represented as

$$\texttt{MethPreTag}\,((\texttt{LookUpStoreBML}\ lv1\,) > 0) \tag{4.1}$$

where we use $e > e'$ instead of `Rel greater` e e' to improve readability.

In JML method parameters in the postcondition are by default considered to be evaluated in the initial store and heap. The ensures-clause of the example is therefore represented as:

$$\begin{aligned}\texttt{MethPostTag}\ \ &((\texttt{LookUpHeapBML}\,(\texttt{Old(This)})\ (\texttt{FieldName}\,\text{Bill.sum}\,)\\ &\leq\\ &(\texttt{Old}\,(\texttt{LookUpHeapBML This}\ (\texttt{FieldName}\,\text{Bill.sum}\,)) + (\texttt{Old}\,(\texttt{LookUpStoreBML}\ \text{lv1}\,))*\\ &((\texttt{Old}\,(\texttt{LookUpStoreBML}\ \text{lv1}\,))+\ 1\,)\ \texttt{div}\ 2\,)\end{aligned} \tag{4.2}$$

Again, to improve readability we use some syntax translations — "1" and "2" are in fact `Number` 1 and `Number` 2, the infix ex + ex' denotes `Binary plus` ex ex', and similarly for the comparisons $<, \leq$, multiplication $*$, and division div.

Finally, the method's loop invariant reads as follows:

```
        loop_invariant 0 < lv[2] && lv[2] <= lv[1] + 1 &&
                       0 <= #24 && #24 <= \old(#24) + (lv[2] - 1) * lv[2]/2
```

As will be explained in the next section, loop invariants will give rise to appropriate local assertions. Therefore, this expression gets translated into the following assertion in the deep expression language (again using obvious syntax translations):

$$\begin{array}{ll}
\texttt{LocAssnTag} & 0 \text{ ¡ } (\texttt{LookUpStoreBML}\ \mathrm{lv2}\,) \\
 & \wedge \\
 & ((\texttt{LookUpStoreBML}\ \mathrm{lv2}\,) \leq (\texttt{Old}\,(\texttt{LookUpStoreBML}\ \mathrm{lv1}\,)) + 1\,) \\
 & \wedge \\
 & (0 \text{ ¡ } (\texttt{LookUpHeapBML}\ \texttt{This}\ (\texttt{FieldName}\ \mathrm{Bill.sum}))) \\
 & \wedge \\
 & ((\texttt{LookUpHeapBML}\ \texttt{This}\ (\texttt{FieldName}\ \mathrm{Bill.sum})) \leq \\
 & \quad (\texttt{Old}(\texttt{LookUpHeapBML}\ \texttt{This}\ (\texttt{FieldName}\ \mathrm{Bill.sum}\,)) + \\
 & \quad (((\texttt{LookUpStoreBML}\ \mathrm{lv2}) \text{ - } 1\,) * (\texttt{LookUpStoreBML}\ \mathrm{lv2}\,))\ \texttt{div}\ 2\,))
\end{array} \tag{4.3}$$

4.3 Embedding BML in the MOBIUS base logic

In the previous section, we showed how BML predicates can be mapped into expressions of the deep expression language and in turn, how these expressions can be mapped into assertions for the (shallow) MOBIUS base logic. Given these mappings, we are now ready to define how complete BML specifications can be translated into judgements of the MOBIUS base logic.

As in JML, a BML specification can contain the following constructs:

- class specifications, namely invariants and history constraints;
- method specifications, consisting of preconditions (`requires` clause), normal postconditions (`ensures` clause), exceptional postconditions (`signals` clause), and frame conditions (`assignable` clause);
- statement annotations, in particular the `assert`, `assume`, and `set` annotations, that can occur at any point in the program text, and loop invariants, loop frame conditions, and loop variants, that are associated to loop blocks.

Above we have described how BML predicates can be embedded into expressions of the deeply embedded expression language. Here we define how the different specification constructs can be embedded into a judgement of the form $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$ of the MOBIUS base logic.

Class specifications For invariants and history constraints, the BML semantics corresponds to the JML semantics. Thus, any class invariant or constraint can be translated into appropriate pre- and postconditions for the relevant methods (taking the visibility modifiers into account). We follow this semantics for JML [?, §8] and do not discuss it further in this document.

Method specifications BML preconditions directly correspond to the preconditions of the MOBIUS base logic judgements (using the embedding in the deeply embedded expression language). The normal and exceptional postconditions are combined into a single postcondition, specifying with a case distinction which conditions should hold if the state is normal or exceptional, respectively. Frame conditions should also be added to the postconditions, specifying explicitly which variables are allowed to be changed. Since in the deep embedding (and the shallow predicates in the MOBIUS base logic) one can specify properties over the whole heap, this can be expressed directly: all locations that are not mentioned in the frame condition of the method (evaluated in the prestate of the method) should be unchanged.

In JML and also in BML it is possible to write several method specifications for a single method. A standard procedure exists for desugaring these into a single method specification [47]. We use this approach in our translation.

Statement annotations The treatment of `assert` annotations is simple, they are transformed directly into program annotations.

For `assume` annotations, different interpretations are possible: either the `assume` statement adds some additional information that is supposed to be added to the properties that are already known for a particular program point or the `assume` statement corresponds to a local precondition for the remaining statements. In the latter case they have to be sufficiently strong to prove correctness of all the remaining program code. We are currently investigating these different interpretations and which would be the most appropriate for our work.

At the moment, the MOBIUS base logic does not have the ghost variables; it is future work to add these. To specify assignments to ghost variables, `set` annotations are used. Therefore, it only makes sense to map `set` annotations into the MOBIUS base logic, once it has ghost variables. In JML, `set` annotations occur in the program text. In BML, they are described in a special set annotation table, which associates them to a particular instruction. The latter approach will probably also be followed in the MOBIUS base logic, i.e. there will be a set annotations table. It is still to be decided whether the set annotations associated to a particular instruction are an unordered set or whether we will actually specify a sequence of instructions (currently in BML it is described as an unordered set).

Similar to `assert` annotations, loop invariants also directly become program annotations of the appropriate program point.

Loop variants can be transformed into a sequence of assert and set annotations, after introducing appropriate ghost variables. This transformation can be done at the level of BML, after which we can use the mapping of set and assert annotations into the MOBIUS base logic. The transformation of the BML specification basically proceeds as follows. Let `variant` be the expression declared in the decreases clause. We declare ghost variables `loop_init` (initially set to true) and `loop_variant` (the preliminary initialisation of which is not essential). If l is the program point where we enter the loop, then at that point we add an assertion

```
//@ assert !loop_init ==> (0 <= variant && variant < loop_variant);
```

followed by:

```
//@ set loop_init = false;
//@ set loop_variant = variant;
```

This ensures that every time the loop entry point l is reached again, the decrease of the loop variant is checked. Only a path that goes through the loop can set `loop_init` to false.

For mapping loop frame conditions, we use again the fact that in the deep expression language we can express properties of the heap. Thus, we again make a transformation into a sequence of assert and set statements. We declare ghost variables to remember the old heap and all locations mentioned in the loop frame condition, and a ghost variable `loop_init` as above. Then we assert at the entry point of the heap that if `loop_init` does not hold, any location that is not mentioned in the loop frame condition should remain unchanged. Notice that this assertion cannot be directly expressed in BML, but it can be expressed in the deep expression language. Finally, in the MOBIUS base logic we add appropriate ghost variable updates to remember the old heap and the locations of the loop frame condition when the loop was first entered.

Running example Consider again the BML specification of the class `Bill` in Figure 4.2. We have already shown above how the different predicates can be translated into the deep expression language. Let us use γ_{PRE}, γ_{POST} and γ_{INV} for expressions (4.1), (4.2) and (4.3), respectively. Let us further assume that $\mathcal{I}$ is the deep expression encoding the class invariant `#24 >= 0`, i.e.

$$\mathcal{I} = \texttt{LookUpHeapBML This (FieldName } \mathit{Bill.sum}) \leq 0.$$

If we map the BML specification into the MOBIUS base logic, this will result in the following judgement:

$$\mathsf{G}, \mathsf{Q} \vdash \{\texttt{MethPreTag}(\mathcal{I}) \wedge \gamma_{\text{PRE}}\}\, pc\, \{\texttt{MethPostTag}(\mathcal{I}) \wedge \gamma_{\text{POST}}\}\, (\mathsf{true}).$$

Finally, the local annotation table Q will contain a single entry: from instruction 22 (loop entry point) to γ_{INV}.

Chapter 5

Verification techniques for bytecode programs

The use of verification condition (VC) generators makes the automation of proof easier. They compute formulae using the specification of a program in such a way that the validity of these formulae implies the validity of the program with respect to its specifications.

We present here two VC generators. The first one, described in section 5.1, generates verification condistions in a direct way, i.e. it works on the bytecode programs and transforms formulae directly using the semantic rules of the bytecode language expressed in a weakest precondition calculus derived from MOBIUS base logic. The second VC generator, described in section 5.2, works with BoogiePL as an intermediate programming language and program logic. The main advantages of the second VC generator are that it makes possible to reuse the tools developed within the Spec# platform and gives a clear method for generation of verification conditions the size of which is proportional to the size of the original program. The second VC generator is related to the MOBIUS base logic by a soundness proof w.r.t. the first verification generator.

5.1 Verification Condition Generator for MOBIUS base logic

In this section we present a formalisation of a verification condition generator (VC generator) using shallowly embedded assertions. We first define the VC generator and prove the correctness of the VC generator up to the MOBIUS base logic (i.e. if all conditions generated by the VC generator are satisfied then there exists a derivation in the MOBIUS base logic for the program and its specification). The soundness proof of the MOBIUS base logic implies the soundness of the VC generator. We also give a direct soundness proof of the VC generator.

The major differences with the MOBIUS base logic are:

- The VC generator does not deal with method invariants Φ, we plan to add methods invariants in the VC generator later. This should normally be done without any major difficulty.
- In the MOBIUS base logic a proof context associates a triple (A, B, I) to a program point, where A is a local precondition B a local postcondition and I a local invariant. The local specification table of a method can be seen as the proof context G of the MOBIUS base logic, but for the VC generator the local specification table contains only local preconditions (the local postcondition is always the postcondition of the method, and the VC generator does not currently deal with global invariants).
- In the MOBIUS base logic at least one derivation rule is used for each instruction of the program, to be able to prove its correctness. The VC generator is simply a strategy to automatically apply rules of the MOBIUS base logic. It generates a condition corresponding to the conjunction of the side-conditions needed by the MOBIUS base logic rules. Furthermore, it tries to automatically remove some side-conditions when they are trivially provable.

5.1.1 Overview of the VC generator

The verification condition generator is defined for annotated programs.

- An annotated program is a triple $(p, \mathit{subclass}, \mathit{MST})$, where p is a JVM program (as represented in Bicolano), *subclass* is a boolean function over pairs of class names of p (i.e. a decidable version of the subclass relation of p), and MST is the method specification table of p.

- A method specification table is a partial mapping from methods to pairs of global method specifications (the precondition and the postcondition of the method) and local specification tables of the method.
- A local specification table is a partial mapping from program points to assertions (predicates over initial state and current state) that should be understood as the precondition of the program point.

$$\boxed{\begin{array}{rcl} \mathit{AnnotProg} & = & p * \mathit{subclass} * \mathit{MST} \\ \mathit{subclass} & = & \mathit{ClassName} * \mathit{ClassName} \rightarrow \mathit{bool} \\ \mathit{MST} & = & \mathcal{M} \mapsto \mathit{MethodPre} * \mathit{MethodPost} * \mathit{LMS} \\ \mathit{LMS} & = & \mathit{PC} \mapsto \mathit{LocalPre} \\ \mathit{MethodPre} & = & \mathit{InitState} \rightarrow \mathit{Prop} \\ \mathit{MethodPost} & = & \mathit{InitState} * \mathit{TermState} \rightarrow \mathit{Prop} \\ \mathit{LocalPre} & = & \mathit{InitState} * \mathit{LocalState} \rightarrow \mathit{Prop} \end{array}}$$

In the sequel we write p for an annotated program $(p, \mathit{subclass}, \mathit{MST})$, and we will write $p.\mathit{subclass}$ for the second component of $(p, \mathit{subclass}, \mathit{MST})$ and $p.\mathit{MST}$ for the third one.

The definition of the VC generator is based on two basic, mutually dependent functions:

$$\begin{array}{rcl} \mathsf{wp_i} & : & \mathit{AnnotProg} \rightarrow \mathcal{M} \rightarrow \mathit{PC} \rightarrow \mathit{LocalPre} \\ \mathsf{wp_l} & : & \mathit{AnnotProg} \rightarrow \mathcal{M} \rightarrow \mathit{PC} \rightarrow \mathit{LocalPre} \end{array}$$

The function $\mathsf{wp_i}\ p\ m\ pc$ computes the weakest precondition of a program point pc in an annotated program p and a method m without using the local specification table (at that point). Intuitively the function computes the weakest precondition of all successors of pc (using the function $\mathsf{wp_l}$) and then uses the different results and the semantics of the instruction at pc to compute the weakest precondition of the instruction at pc. The function $\mathsf{wp_l}\ p\ m\ pc$ does the same, but also uses the local specification table of the method m directly: if the program point pc is annotated with the precondition A then the weakest precondition of pc (computed by $\mathsf{wp_l}$) is A, else the weakest precondition of pc is $\mathsf{wp_i}\ p\ m\ pc$. We will define these two functions in the next section.

Essentially, the function $\mathsf{wp_i}$ assumes the role of the judgement $\mathit{MST}, \mathit{LMS} \vdash \{\bullet\}\, pc\, \{\bullet\}\, (\bullet)$ in the MOBIUS base logic, and the function $\mathsf{wp_l}$ assumes the role of $\mathit{MST}, \mathit{LMS} \vdash \langle\bullet\rangle\, pc\, \langle\bullet\rangle\, (\bullet)$.

With the use of these two functions, we can define the notion of certified methods and certified programs.

Definition 5.1.1 (Certified methods) Given an annotated program p, a precondition R, a postcondition T and a local specification table $\mathcal{S}$, a method m is certified if $p.\mathit{MST}(m) = (R, T, \mathcal{S})$ and the following property holds:

$$\begin{array}{l} \texttt{certifiedMethod}\ p\ \mathcal{S}\ m \equiv \\ \quad (\forall (s_0, s).\ R\ s_0 \rightarrow \mathsf{wp_l}\ p\ m\ \mathsf{init}_m\ (s_0, \mathit{state}(s_0))) \wedge \bigwedge_{\mathcal{S}(pc)=A} \forall (s0, s).\ A(s_0, s) \rightarrow \mathsf{wp_i}\ p\ m\ pc\ (s_0, s). \end{array}$$

In other words, a method is certified if its precondition implies the weakest precondition of the starting point of the method and for all annotations A of an annotated point pc in m, A implies the weakest precondition of the instruction at pc. Assume that only pc_1 and pc_2 are annotated (with A_1 and A_2) then $\texttt{certifiedMethod}\ p\ \mathcal{S}\ m$ reduces to

$$\begin{array}{rl} & (\forall (s_0, s).\ R\ s_0 \rightarrow \mathsf{wp_l}\ p\ m\ \mathsf{init}_m(s_0, \mathit{state}(s_0))) \\ \wedge & (\forall (s0, s).\ A_1(s_0, s) \rightarrow \mathsf{wp_i}\ p\ m\ pc_1\ (s_0, s)) \\ \wedge & (\forall (s0, s).\ A_2(s_0, s) \rightarrow \mathsf{wp_i}\ p\ m\ pc_2\ (s_0, s)). \end{array}$$

Definition 5.1.2 (Certified programs) An annotated program p is certified whenever the following property holds:

$$\texttt{certifiedProg}\ p \equiv \bigwedge_{p.\mathit{MST}(m)=(R,T,\mathcal{S})} \texttt{certifiedMethod}\ p\ \mathcal{S}\ m.$$

This definition, similarly as the definition of certified methods, is reducible which has the advantage that the Coq system can compute the result of the VC generator. This can be observed when a proof of correctness for a particular bytecode method is conducted in Coq — the formulae generated by the WP generator show up automatically in appropriate proof steps.

5.1.2 Weakest precondition of instructions

The weakest precondition (WP) of instructions can be seen as a symbolic execution. Instead of computing a result, the symbolic execution computes a weakest precondition which intuitively corresponds to the conjunction of the side-conditions of the MOBIUS base logic. The WP tries to simplify and to reduce the number of side-conditions of the MOBIUS base logic, in particular the side-conditions corresponding to exception cases.

In the MOBIUS base logic, the rule (INSTR), corresponding to basic instructions, contains two side-conditions:

- (3.1) an exception is thrown and caught by the handler,
- (3.2) an exception is thrown and uncaught.

These two side-conditions quantify over all exceptions that can be handled or not at that position. But most of the basic instructions cannot throw exceptions. In that case, these two conditions can simply be removed. Furthermore, the exceptions that are possibly thrown by basic instructions as well as their exception handlers are known statically (runtime exceptions). So, we can statically decide if the exception is caught by a handler or uncaught. This means that at least one of these two side-conditions can be removed.

A problem is that in the Bicolano semantics whether an exception is caught or not is defined by a (inductive) predicate

$$pc' \in handle_m(pc, e)$$

which is not executable. Here, we need a decidable function, this is why we require a subclass relation in an annotated program. This allows us to write a function *lookuphandlers m pc cn* returning pc', when $pc' \in handle_m(pc, e)$, or $\perp$ otherwise.

Assertion transformers

In order to simplify the presentation of the weakest precondition rules, we first define some operators. The definitions are implicitly parametrised by an annotated program p, a method m and a function wp_l.

- $\mathsf{wp}_\mathrm{next}\ pc$: computes the WP of the successors of pc if they exist (in Bicolano, the next function is a partial function, some program points do not have any successors).

$$\mathsf{wp}_\mathrm{next}\ pc = \begin{cases} \mathsf{wp}_\mathrm{l}\ pc' & \text{if } next_m(pc) = pc', \\ \lambda\,(s_0, s).\ \mathsf{True} & \text{if } next_m(pc) = \perp. \end{cases}$$

  The case $next_m(pc) = \perp$ never occurs if the program is well formed.

- $\mathsf{wp}_\mathrm{Exc}\ pc\ loc\ cn\ h\ l\ s_0$ computes the WP of a program point pc when an exception stored at location loc of a class cn is thrown, h is the current heap and l are local variables.

$$\mathsf{wp}_\mathrm{Exc}\ pc\ loc\ cn\ h\ l\ s_0 = \begin{cases} m.T(s_0, (h, loc)) & \text{if } lookuphandlers\ m\ pc\ cn = \perp, \\ \mathsf{wp}_\mathrm{l}\ pc'\ (s_0, (h, loc :: \emptyset, l)) & \text{if } lookuphandlers\ m\ pc\ cn = pc'. \end{cases}$$

  If the exception is not caught (*lookuphandlers* returns $\perp$) then the current method returns the exception. In that case the WP is the postcondition of m ($m.T$) applied to the initial state s_0 and the return state (h, loc). If the exception is caught (*lookuphandlers* returns pc'), the next state is $(pc', (h, loc :: \emptyset, l))$ (i.e. the code pointer becomes pc', the heap and local variables are unchanged, the stack is cleared). So the WP resulting is the WP of pc' applied to the initial state s_0 and the state at pc': $(h, loc :: \emptyset, l)$.

- $\mathsf{wp}_\mathrm{JvmExc}\ pc\ e\ h\ l\ s_0$ computes the WP of a program point pc when the JVM exception e is thrown.

$$\begin{aligned} &\mathsf{wp}_\mathrm{JvmExc}\ pc\ e\ h\ l\ s_0 = \\ &\quad \forall\, h'\ loc.\ new\ h\ p\ (javaLang, e) = (loc, h') \to \mathsf{wp}_\mathrm{Exc}\ pc\ loc\ (javaLang, e)\ h'\ l\ s_0. \end{aligned}$$

  When an instruction throws a JVM exception e, a fresh object of class name $(javaLang, e)$ is allocated, after this allocation the heap is h' and loc is a reference to the newly allocated object representing the exception. In these cases, we enrich the WP with two fresh variables h' and loc together with a hypothesis characterising these two variables as a function of the current heap (h).

- $\mathsf{wp}_{\text{Null}}\ pc = \mathsf{wp}_{\text{JvmExc}}\ pc\ \mathit{NullPointerException}$.
- $\mathsf{wp}_{\text{Cond}}\ C\ T\ F = (C \to T) \wedge (\neg C \to F)$. This definition is used when the semantics performs a test.
- $\mathsf{wp}_{\text{JvmCond}}\ C\ T\ pc\ e\ h\ l\ s_0 = \mathsf{wp}_{\text{Cond}}\ C\ T\ (\mathsf{wp}_{\text{JvmExc}}\ pc\ e\ h\ l\ s_0)$. Most of the instructions that throw a runtime exception e perform a test C, if the test is true then the execution continues normally (T is morally the WP of the next instruction), else the runtime exception e is thrown.

  Remark: It is very simple to modify this definition so that the WP excludes runtime exceptions, just use the following definition:

  $$\mathsf{wp}_{\text{JvmCond}}\ C\ T\ pc\ e\ h\ l\ s_0 = C \wedge (C \to T)$$

  It is trivially correct up to the first definition since if this proposition is satisfied then the false branch of the conditional can never appear.
- $\mathsf{wp}_{\text{Bound}}\ i\ \mathit{length}\ T\ pc = \mathsf{wp}_{\text{JvmCond}}(0 \leq i < \mathit{length})\ T\ pc\ \mathit{ArrayIndexOutOfBoundsException}$. This definition is used for instructions that access an array (load or store values).

Weakest precondition of instructions: wp_{i}

The function wp_{i} expects three arguments: the program counter pc, the initial state s_0 and the "symbolic" current state s. As above, the definition is implicitly parametrised by an annotated program p, a method m and by a function wp_{l}. The goal of this function is to compute the WP of a program point pc in the program p and method m.

The function proceeds by case analysis on the instruction at label pc and on the state s. Intuitively the function computes the WP of all possible successors of pc (that is a predicate over initial state and local state) and then applies each result to their corresponding state (i.e. the state derived from s after the execution of the instruction).

We do not give the entire definition of the function wp_{i}, but just some examples to give the intuition:

- If the current instruction is *AconstNull* and the current state is $s = (h, os, l)$, the JVM pushes the $Null$ value on top of the operand stack and jumps to the next instruction. The function wp_{i} does the same: first it computes the WP of the next instruction and then applies this to the corresponding state (expressed as a function of s).

  $$\mathsf{wp}_{\text{i}}\ pc\ (s_0, (h, os, l)) = \mathsf{wp}_{\text{next}}\ pc\ (s_0, (h, Null :: os, l)) \qquad \text{if } m.(pc) = \mathit{AconstNull}.$$
- If the current instruction is *Getfield* f, we have to distinguish two cases:
  - $s = (h, Null :: os, l)$, in that case the object we try to access is a null pointer, the semantics of the JVM throws a *NullPointerException*, so we use the predefined function $\mathsf{wp}_{\text{Null}}$:

    $$\mathsf{wp}_{\text{i}}\ pc\ (s_0, (h, Null :: os, l)) = \mathsf{wp}_{\text{Null}}\ pc\ h\ l\ s_0.$$
  - $s = (h, loc :: os, l)$, the object is not a null pointer, the semantics accesses the field f of the reference loc, pushes it on top of the operand stack and jumps to the next instruction. So the resulting state is $(h, v :: os, l)$ where v is value of the field f of loc. The WP simply introduces a new variable v, adds the hypothesis characterising v and proceeds as previously:

    $$\begin{aligned}&\mathsf{wp}_{\text{i}}\ pc\ (s_0, (h, loc :: os, l)) = \\ &\qquad \forall (i,t).\ \mathit{typeof}(h, loc) = \mathit{LocationArray}(i,t) \to \mathsf{wp}_{\text{next}}\ pc\ (s_0, (h, i :: os, l)).\end{aligned}$$
- If the instruction is *If0 cmp o*, the function applies the function $\mathsf{wp}_{\text{Cond}}$ to the condition and to the WP of both successors:

  $$\mathsf{wp}_{\text{i}}\ pc\ (s_0, (h, i :: os, l)) = \mathsf{wp}_{\text{Cond}}\ (i\ \mathit{cmp}\ 0)\ (\mathsf{wp}_{\text{l}}\ (pc + 0)\ (s_0, (h, os, l)))\ (\mathsf{wp}_{\text{next}}\ pc\ (s_0, (h, os, l))).$$
- If the instruction is *Return*, the WP is simply the postcondition of m applied to the return state:

  $$\mathsf{wp}_{\text{i}} pc\ (s_0, (h, v :: os, l)) = m.\mathit{post}\ (s_0, (h, v)).$$

It is now easy to define the function wp_l:

$$\mathsf{wp}_\mathsf{l}\ pc = \begin{cases} A & \text{if } \mathcal{S}.(pc) = A, \\ \mathsf{wp}_\mathsf{i}\ pc & \text{if } \mathcal{S}.(pc) = \bot \end{cases}$$

with $p.MST(m) = (R, T, \mathcal{S})$.

5.1.3 Correctness of the VC generator

Theorem 5.1.3 (Correctness of the VC generator) For each annotated program p and method m, such that the proof obligation generated by the VC generator is satisfied (i.e. a proof of `certifiedProg` p exists) and $p.MST(m) = (R, T, \mathcal{S})$ we have that if the proposition $\mathsf{wp}_\mathsf{l}\ p\ m\ pc\ (s_0, s)$ is valid and the state (pc, s) evaluates to:

- the state (pc', s'), then the proposition $\mathsf{wp}_\mathsf{l}\ p\ m\ pc'\ (s_0, s')$ is valid,
- the return state (h, rv), then the proposition $T(s_0, (h, rv))$ is valid.

The first item should be understood as a "subject reduction" property, while the second is the property we want about the VC generator: if the evaluation of the method terminates then the postcondition is satisfied. Using the definition of `certifiedProg` and `certifiedMethod`, it is trivial to prove that if we start the evaluation of a method in a initial state satisfying its precondition, the result (if the function terminates) satisfies the postcondition.

Correctness of the VC generator up to the MOBIUS base logic

To prove the correctness of the VC generator up to the MOBIUS base logic, we first need to translate the annotation table of the VC generator to the annotation table of the MOBIUS base logic. The differences between the two are that annotation tables of the MOBIUS base logic contain information for the method invariants and a local annotation table. To complete the annotation table of the VC generator, we use the trivial "true" proposition and we set the local annotation table to empty.

Theorem 5.1.4

- If the proposition $\forall(s_0, s).\ \mathsf{wp}_\mathsf{l}\ p\ m\ pc\ (s_0, s)$ is valid then there exists a derivation of
$$MST, LMS \vdash \langle(\mathsf{wp}_\mathsf{l}\ p\ m\ pc)\rangle\ pc\ \langle m.post\rangle\ (\mathsf{True}).$$
- If the proposition $\forall(s_0, s).\ \mathsf{wp}_\mathsf{i}\ p\ m\ pc\ (s_0, s)$ is valid then there exists a derivation of
$$MST, LMS \vdash \{(\mathsf{wp}_\mathsf{l}\ p\ m\ pc)\}\ pc\ \{m.post\}\ (\mathsf{True}).$$
- If the proposition `certifiedProg` p is provable then p is verifiable in the MOBIUS base logic.

5.1.4 Running example

With the help of the running example we present now the way the VC generator works. To use the VC generator we first must define the method specification table of the Bill program. We start with the method specification for the `roundCost` method:

```
Definition roundCostPre (s0:InitState) := True.
Definition roundCostPost (s0:InitState) (t:ReturnState) :=
  let (l0, h0) := s0 in
  let (ht,vt) := t in
      (forall (am:Heap.AdressingMode) (v:value),
        Heap.get h0 am = Some v -> Heap.get ht am = Some v)
   /\
      match vt with
      | Normal res =>
```

```
         forall x, LocalVar.get l0 1%N = Some (Num (I x)) ->
          exists n, res = Some (Num (I n)) /\ 0 <= n <= x
       | Exception loc =>
         Heap.typeof ht loc = Some (Heap.LocationObject java_lang_Exception.className)
       end.
 Definition roundCostSpec := (roundCostPre, roundCostPost, empty).
```

The `roundCost` method has no precondition which is modelled by a predicate function (`roundCostPre`) which associates to any initial state `s0` the proposition "true". What does the postcondition of the method say? It is a predicate over the initial state and return state: it decomposes the initial state into the initial local variables (`l0`) and the initial heap (`h0`) (resp.). Similarly, the terminal state (`t`) is decomposed into the final heap (`ht`) and the result (`vt`). The first proposition says that the method does not modify values contained in the initial heap. The second one describes the result: if the result is a value then the value (`n`) is between 0 and the initial value of the argument of the method (`x`). Otherwise, if the method returns with an exception, then the exception has the type `Exception`. Finally, the specification of the virtual method is a triple composed of its precondition, postcondition and the empty local method specification (since the method is abstract and its body need not be specified).

Now, let us define the precondition of the `produceBill` method:

```
 Definition mk_produce_bill_pre (s0:R.InitState) loc (n sum:Int.t) :=
   let (l0,h0) := s0 in
      l0 = mk_bill_var loc n
   /\ Heap.get h0 (Heap.DynamicField loc Bill.sumFieldSignature) = Some (Num (I sum))
   /\ 0 <= sum
   /\ sum + (n*(n+1))/2 < Int.half_base.

 Definition produce_bill_pre (s0:R.InitState) :=
   exists loc, exists n, exists sum, mk_produce_bill_pre s0 loc n sum.
```

The precondition says that there exist three values (`loc`, `n`, `sum`) such that the initial local variables contain the `loc` at position 0 and the integer `n` at position 1. The identifier `sum` represents the value of the sum field of the reference `loc`. This value should be positive and the last condition ensures that there is no overflow (this is important for a proof of the postcondition).

The postcondition of the method is the following:

```
 Definition produce_bill_post (s0:R.InitState) (t:ReturnState) :=
   let (l0, h0) := s0 in
   let (ht,vt) := t in
   forall loc n sum, mk_produce_bill_pre s0 loc n sum ->
      (forall (am:Heap.AdressingMode) (v:value),
         am <> (Heap.DynamicField loc Bill.sumFieldSignature) ->
         Heap.get h0 am = Some v -> Heap.get ht am = Some v)
   /\ (exists sum',
           Heap.get ht (Heap.DynamicField loc Bill.sumFieldSignature) =
               Some (Num (I sum'))
        /\ 0 <= sum' <= sum + (n*(n+1))/2)
        /\ match vt with
           | Normal _ => True
           | Exception _ => False
           end.
```

It expresses that for all triples (`loc`, `n`, `sum`) satisfying the precondition of the method, the method does not modify the heap (except the sum field of the reference `loc`), after execution of the method this field contains a positive value `sum'` less than the initial value of the field plus the expression $(n * (n + 1))/2$. The last condition ensures that the method does not raise an exceptions.

Since the body of the method contains a loop we need to give its invariant. This invariant is a local annotation of the method (a predicate over the initial state and the current state):

```
Definition produce_bill_invariant (s0:R.InitState) (s:R.LocalState) :=
   let (l0,h0) := s0 in
   let (h,s,l) := s in
   exists loc, exists n, exists sum0, exists sum, exists i,
        mk_produce_bill_pre s0 loc n sum0
    /\ (forall (am:Heap.AdressingMode) (v:value),
           am <> (Heap.DynamicField loc Bill.sumFieldSignature) ->
           Heap.get h0 am = Some v -> Heap.get h am = Some v)
    /\ l = LocalVar.update (mk_bill_var loc n) 2 (Num (I i))
    /\ 1 <= i <= n + 1
    /\ Heap.get h (Heap.DynamicField loc Bill.sumFieldSignature) = Some (Num (I sum))
    /\ 0 <= sum <= sum0 + ((i-1)*i)/2.
```

It tells that there exists some value satisfying the precondition of the method, a current value (`sum`) for the sum field and a current value (`i`) of the variable i from the source program (at position 2 in the bytecode). The current heap is the same as the initial heap except that the sum field contains now the value `sum`, which is specified. The values of the local variables at positions 0 and 1 have not been modified, the value `i` is between 0 and the initial value of the variable `n` plus one.

Finally, the specification of the `produceBill` method is a triple composed of its precondition, postcondition, and the local annotation table which associates the invariant with the program point 22 (the entry point of the loop).

To obtain the correctness of the `produceBill` method the following Coq lemma should be proved:

```
Lemma produce_bill_correct:
   certifiedMethod annoProg Bill.produce_billMethod produce_bill_spec.
```

where `annoProg` is the annotated version of the Bill program (associating the two specifications to the two methods). A script with a Coq proof of the lemma is presented in Appendix D.

5.2 VC generation via translation to BoogiePL

In this section we present a translation of annotated sequential bytecode to the BoogiePL language. With an implementation of such a translation it is possible to generate small verification conditions that can be checked efficiently with automatic theorem provers like Simplify.

In Task 3.6 of the Mobius project, BoogiePL will be used as an intermediate step when generating verification conditions. A translator from source code to BoogiePL as well as a translator from bytecode to BoogiePL will be part of the Mobius tool suite.

First, we give a brief overview of BoogiePL and sketch a logic for it. We then introduce the translation and illustrate it by a presentation of how it works for the running example. Finally, we present the soundness theorem of the translation with respect to the direct VC generation from annotated bytecode. As the formalisation of this translation is carried out on paper only, we present the description of the translation in a more detailed way.

5.2.1 Overview of BoogiePL

BoogiePL [21] is an intermediate language for imperative programming languages that enables generation of efficiently checkable verification conditions. It has a simple type system containing the basic types int, bool, ref, and any, together with a predefined literal null of type ref. Additionally, BoogiePL has a basic type name which can be used to type field-names for example. It is also possible to define new types in BoogiePL.

In the global name space we can define variables, constants, axioms, uninterpreted functions, and the implementations of procedures, where the BoogiePL commands can be used. The background theory (e.g. the axiomatic heap model presented in subsection 5.2.4) is described with the use of the first four kinds of definitions.

Procedures of BoogiePL can contain specifications for preconditions, postconditions, and modifies clauses. In our translation, we do not make use of this feature as you can formalise those specifications directly with the use of `assume` and `assert` constructs. The implementation of a procedure contains a declaration of all local variables at the beginning, followed by blocks of BoogiePL commands. A block has an ID and a non-deterministic `goto` at the end, which specifies all possible blocks that can follow in the control flow graph of the BoogiePL program.

The idea of BoogiePL is to have a very small set of operations that are used to transform an imperative language with specifications to verification conditions. Namely, we have an assignment statement, a call statement and most importantly, we have an `assume`, `assert`, and a `havoc` statement. For example, "`havoc var`" removes any information about variable `var` by assigning an arbitrary value to it. In other words: after the command "`havoc var`", `var` can contain any value that is allowed by its type.

5.2.2 A logic for BoogiePL

Barnett and Leino defined a WP-calculus for a passive BoogiePL without back-edges in it's control flow graph [7]. Here, "passive" means that the program is converted to a single assignment form and all assignments are replaced by assumptions. Additionally, all back-edges are removed and the loop invariant is appropriately asserted or assumed depending on the position in the BoogiePL code.

We define a logic for an non-passive version of BoogiePL without back-edges. It means that we still have assignment statements as well as `havoc` statements, but in the course of the translation we eliminate the back-edges from the control flow graph. We assume that the statements of the resulting BoogiePL representation are numbered. We use the term "position" to refer to the number of a BoogiePL statement (this corresponds to *pc* for bytecode).

We define the following operations on positions:

- $\mathsf{init}_{m_{bpl}}$ yields the first position of a method body.
- $position(m, pc)$ yields the position of the first BoogiePL statement in the translation of the bytecode instruction at program counter *pc*. If *pc* is None, this function returns `post_X`. This allows us to use position together with the function lookuphandlers which is described in the VC generator section (5.1). to get the desired behaviour. This informatin can be gathered during translation.
- $stmt(bpl, m, pos)$ yields the BoogiePL statement at the position *pos* for a given BoogiePL program *bpl* and a method *m*.

We use positions to define a wp function for BoogiePL. This function computes the weakest precondition from a given position to the end of the BoogiePL code chunk in the first argument:

$$\mathsf{wp_b} : BoogiePLProg \times Method \times Position \to LocalPre$$

The function is defined as follows:

```
$\mathsf{wp_b}(bpl, m, pos) \equiv$
  match $stmt(bpl, m, pos)$ with
  — assume P:
    $P \implies \mathsf{wp_b}(bpl, m, pos + 1)$
  — assert P:
    $P \wedge \mathsf{wp_b}(bpl, m, pos + 1)$
  — x := e:
    $\mathsf{wp_b}(bpl, m, pos + 1)[e/x]$
  — havoc x:
    $\forall v.\mathsf{wp_b}(bpl, m, pos + 1)[v/x]$
  — return:
    true
  — goto $pos_1..pos_n$:
    $\bigwedge_{i:=1..n} \mathsf{wp_b}(bpl, m, pos_i)$
```

5.2.3 Running example in BoogiePL

In the following sections, we will use the running example to clarify the translation of bytecode into BoogiePL. Listing 5.1 shows the translation of the method `produceBill()`. The variable declaration part starting in the line 5 is abbreviated to present only one variable declaration. The original bytecode instructions are shown as comments on the right hand side of the first line of the corresponding BoogiePL text.

```
procedure Bill.produceBill(param0: ref, param1: int) returns (result:int);
    modifies heap;

implementation Bill.produceBill(param0: ref, param1: int) returns (result:int){
    var stack0r: ref ... // further variable declarations omitted

init:
    //Save old heap
    old_heap := heap;

    reg0r := param0;
    reg1i := param1;


    //Free requires
    assume param0 ≠ null;
    assume typ(rval(param0)) == Bill;
    assume alive(rval(param0),heap);

    goto pre;

pre:
    //Class Invariant:
    assume toint(get(heap, instvar(param0, Bill.sum))) ≥ 0;
    //Precondition:
    assume param1i > 0;

//Translation of first instruction starts here
                                                               //bytecode program:
    stack0i := 1;                                              //0:  iconst_1
    reg2i := stack0i;                                          //1:  istore_2
    assert 0 < reg2i ∧ reg2i ≤ reg1i + 1 ∧ //Loop invariant    //2:  goto 22
           0 ≤ toint(get(heap, instvar(param0, Bill.sum))) ∧
           toint(get(heap, instvar(param0, Bill.sum))) ≤
           toint(get(old_heap, instvar(param0, Bill.sum))) + (reg2i−1)∗reg2i/2;
    goto block_22;

block_5:
    stack0r := reg0r;                                          //5:  aload_0
    stack1r := reg0r;                                          //7:  aload_0
    assert stack1r ≠ null;                                     //9:  getfield sum
    stack1i := toint(get(heap, instvar(stack1r, Bill.sum)));
    stack2r := reg0r;                                          //10: aload_0
    stack3i := reg2i;                                          //11: iload_2
    arg0r := stack2r;                                          //12: invokevirtual roundCost
    arg1i := stack3i;                                          //p.MST(target) = (target_R,target_T,_)
    pre_heap := heap;
    havoc heap;
    assert true  //target_R (pre_heap, args)
    goto block_12_Normal, block_12_Exception;

block_12_Exception:
    havoc stack0r;
    assume alive(rval(stack0r), heap);
    assume typ(rval(stack0r)) ⪯ java.lang.Exception
    assume true; //target_T (pre_heap, args) (heap, (Exception stack0r))
    goto block_29;

block_12_Normal:
    havoc stack2i;
    assume (stack2i ≤ arg1i); //target_T (pre_heap, args) (heap, (Normal stack2i))
    stack1i := stack1i + stack2i;                              //15: iadd
    assert stack0r ≠ null;                                     //16: putfield sum
    heap := update(heap, instvar(stack0r, Bill.sum), ival(stack1i));
    reg2i := reg2i + 1;                                             //19: iinc 2, 1
    assert reg2i ≤ reg1i + 1 ∧ //Loop Invariant
          toint(get(heap, instvar(param0, Bill.sum))) ≤
          toint(get(old_heap, instvar(param0, Bill.sum))) + (reg2i−1)∗reg2i/2;
    return; //Backedge removed

block_22:
    havoc stack0r, ... //all variables of local state
    assume reg2i ≤ reg1i + 1 ∧ //Loop Invariant
          toint(get(heap, instvar(param0, Bill.sum))) ≤
          toint(get(old_heap, instvar(param0, Bill.sum))) + (reg2i−1)∗reg2i/2;
    stack0i := reg2i;                                               //22: iload_2
    stack1i := reg1i;                                               //23: iload_1
    goto block_24_True, block_24_False;                             //24: if_icmple 5

block_24_True:
    assume stack0i > stack1i;
    goto block_5;

block_24_False:
    assume stack0i ≤ stack1i;
    stack0i := 1;                                                   //27: iconst_1
    result := stack0i;                                              //28: ireturn
    goto post;

block_29: //Catch block
    reg3r := stack0r;                                               //29: astore_3
    stack0i := 0;                                                   //30: iconst_0
    result := stack0i;                                              //31: ireturn
    goto post;

post:
    //Postcondition
    assert toint(get(heap, instvar(param0, Bill.sum))) ≤
          toint(get(old_heap, instvar(param0, Bill.sum))) + param1i ∗ (param1i+1) / 2;
    //Class Invariant
    assert toint(get(heap, instvar(param0, Bill.sum))) ≥ 0;
    return;

post_X:
    //Exceptional Postcondition
    assert true
    //Class Invariant
    assert toint(get(heap, instvar(param0, Bill.sum))) ≥ 0;
    return;
}
```

Listing 5.1: The running example translated to BoogiePL

5.2.4 Foundations of the translation

In this section we describe the general setting for the translation of bytecode to BoogiePL. This includes a brief description of the way method specifications are handled, the information about bytecode properties that we assume to know, and a description of the way the operand stack, the registers, and the heap are modelled in BoogiePL.

Method specifications

We consider translating a sufficiently richly annotated program. Especially, we have loop-invariants defined as a local assertion at the first instruction of the loop header. The form of annotations corresponds to the form of annotations used in the previous chapters.

The Method Specification Table (MST) gives us the pre- and postcondition as well as the local annotations for each label of the bytecode program. In this translation we use the following mapping for the specifications:

$$p.MST(m) = (R, T, S).$$

Information on the bytecode

We can safely assume that our input bytecode passed the bytecode verifier and thus we can rely on the following:

1. For every instruction, the height of the operand stack is known.
2. For every program point, the height of the possible operand stack at the point is delimited by MaxStackSize, which is fixed for every method separately.
3. For every program point and possible stack, the smallest common supertype of all possible values in a stack position is known.
4. The control flow graph of the method is known.

Using this information, we can introduce helper functions which support our translation. The functions isEdge and isEdgeTarget allow us to translate the bytecode sequentially, while getStackHeight and getStackType free us from keeping track of the stack contents. The functions isBackEdge and isBackEdgeTarget give us information about the location of back-edges in the control flow graph. In the BoogiePL translation, back-edges are eliminated by the use of local annotations (see [7]).

- $getStackHeight : Program \rightarrow Method \rightarrow PC \rightarrow Int$
  Returns the height of the operand stack at the given program counter location.
- $getStackType : Program \rightarrow Method \rightarrow PC \rightarrow Int \rightarrow Type$
  Returns the smallest common super-type of all possible values at a given program counter location and a given stack position.
- $isEdgeTarget : Program \rightarrow Method \rightarrow PC \rightarrow Boolean$
  Returns true iff the given program counter location is the start of a block in the control flow graph.
- $isEdge : Program \rightarrow Method \rightarrow PC \rightarrow PC \rightarrow Boolean$
  Returns true iff there is an edge between two given program counter locations in the control flow graph.
- $isBackEdgeTarget : Program \rightarrow Method \rightarrow PC \rightarrow Boolean$
  Returns true iff there is a back-edge to the given program counter location.
- $isBackEdge : Program \rightarrow Method \rightarrow PC \rightarrow PC \rightarrow Boolean$
  Returns true iff there is a back-edge from a given program counter location to a given program counter location.

Operand stack

The JVM operand stack is modelled by a set of BoogiePL variables of the form `stackit`, where i denotes the depth of the stack and t is the type of the stack element, which can be either `i` for integers or `r` for object references. If the type is not known (e.g. for the instruction dup, the operation is done on both integer and reference variables.

Line 62 of the running example shows how `iadd` is translated to BoogiePL when we have 3 elements on the stack.

```
62    stack1i := stack1i + stack2i;                    //15: iadd
```

Registers

Registers (which represent the local variables) are treated similarly to stack elements. We use variables **regi**t to represent the i-th register when its type is primitive or object reference.

These two lines from the running example show how register and stack operations are translated. One can see that the stack grows from 0 to the current height of the stack.

```
76    stack0i := reg2i;                                   //22:   iload_2
77    stack1i := reg1i;                                   //23:   iload_1
```

Heap

We use the heap model described in [44], translated to a BoogiePL representation. In this model, the heap is described by a variable of the type Store and accessed through functions. Its behaviour is given by a set of axioms over these functions.

In case of the operand stack and the registers, it is convenient to use the basic types `int` and `ref` of BoogiePL. In case of the heap axiomatisation, this is not sufficient, as those types are not translated into the verification conditions. Therefore, we have to find a way not to loose the information about the type of a variable passed to the heap.

This can be done by introducing a type Value. A value represents either an integer (type $int) or a reference type. Functions are provided to map these types to BoogiePL`int` and `ref` types and vice versa.

Instance variables (locations) are qualified through an object reference and a field identifier. BoogiePL`name` constants are used to model field identifiers, as we will see in an example later on.

```
type Store;

function IsClassType(name) returns (bool); //Boogie 'ref' type
function IsValueType(name) returns (bool); //Boogie 'int' type

//Define $int as the only non reference type
const $int: name;
axiom IsValueType($int);

type Value;

function ival(int) returns (Value);
axiom (∀ x1: int, x2: int :: ival(x1) == ival(x2) ⟺ x1 == x2);
axiom (∀ v: Value :: ival(toint(v)) == v);

function toint(Value) returns (int);
axiom (∀ x: int :: toint(ival(x)) == x);

//... Accordingly for functions 'rval' and 'toref'
```

Listing 5.2: The main types of the heap model

Access to the heap is provided through five functions. Their meanings are given in the comments.

```
// Return the heap after storing a value in a field.
function update(Store, InstVar, Value) returns (Store);

// Returns the heap after an object of the given type has been allocated.
function add(Store, name) returns (Store);

// Returns the value stored in a field.
function get(Store, InstVar) returns (Value);

// Returns true if an object referenced by a value is alive in a given heap.
function alive(Value, Store) returns (bool);

// Returns a newly allocated object of the given type.
function new(Store, name) returns (Value);
```

Listing 5.3: The accessing functions of the heap model

The rules governing the behaviour of the heap are expressed by axioms, in the global space of a BoogiePL file that results from the translation.

We can see these rules in action in lines 41 and 42 of the running example that correspond to the translation of the getfield instruction. The reference in `stack1r` is fed together with the field identifier Bill.sum to `instvar`. This function gives us the location which is used to get the content of it with the function `get`. In the last step, the returned value has to be converted to an integer by `toint`.

```
41    assert stack1r ≠ null;                                //9:   getfield sum
42    stack1i := toint(get(heap, instvar(stack1r, Bill.sum)));
```

5.2.5 The main concepts of the translation

Conditional branches

Conditional branches generate a non-deterministic `goto` to two successor blocks that assume the condition to be true or false. The true-block is then connected to the branch target while the false-block is connected to the block that starts with the instruction immediately following the conditional branch instruction.

We can see this setup in the running example in lines 78 to 85.

```
78    goto block_24_True, block_24_False;                   //24:   if_icmple 5
79
80 block_24_True:
81    assume stack0i > stack1i;
82    goto block_5;
83
84 block_24_False:
85    assume stack0i ≤ stack1i;
```

It is worth pointing out that the values of stack0i and stack1i are not destroyed by the `havoc` instruction after the `assume` statements. It is possible to omit this step as the translation function guarantees that the values in the stack cannot be read before they are overwritten.

Method calls

Method calls are translated so that the precondition of the method is asserted before the call and the postconditions are assumed after that. The latter is done in such a way that the normal and exceptional method termination is taken into account. This is realized by a `goto` instruction which jumps to the code which assumes the normal postcondition, in case the method returned in a normal way, to the code that handles the exception, in case the exception can be handled locally, or to the block that asserts the exceptional postcondition of the calling method, if no handler is installed.

Lines 45 to 61 of the running example show a method call that can throw an exception that is caught in the same frame. The variable `pre_heap` is used to make possible the use of `\old` expressions in the postcondition of the called method. In the exceptional block on lines following 52, we assume that we have a reference to an alive exception object on `stack0r`. In the normal block, we perform `havoc` on the stack element, that will contain the return value and then assume the normal postcondition.

```
45    arg0r := stack2r;                                     //12:  invokevirtual roundCost
46    arg1i := stack3i;                                     //p.MST(target) = (target_R,target_T,_)
47    pre_heap := heap;
48    havoc heap;
49    assert true  //target_R (pre_heap, args)
50    goto block_12_Normal, block_12_Exception;
51
52 block_12_Exception:
53    havoc stack0r;
54    assume alive(rval(stack0r), heap);
55    assume typ(rval(stack0r)) ⪯ java.lang.Exception
56    assume true;  //target_T (pre_heap, args) (heap, (Exception stack0r))
57    goto block_29;
58
59 block_12_Normal:
60    havoc stack2i;
```

```
assume (stack2i ≤ arg1i); //target_T (pre_heap, args) (heap, (Normal stack2i))
```

Exceptions

We distinguish between JVM errors, runtime exceptions, and user generated exceptions. Errors should not be caught in normal application as they mean problems with the platform rather than the program and are therefore not handled at all in the current formalization. More interesting is the way to deal with runtime exceptions. (1) We can just model them as they occur. That is, every instruction that can throw a runtime exception can have two possible outcomes — a normal case and an exceptional case. (2) We can add assertions in such a way that a program that possibly throws a runtime exception cannot be verified. In this case, the user is informed that a runtime exception could be thrown on a certain instruction. The second approach is chosen for this translation. However, it would be easy to support the first approach as well. User generated exceptions are fully supported.

In our translation, instructions that can throw exceptions lead to the creation of additional blocks which represent the normal and possibly several exceptional executions of the instruction. In such an exceptional block, the heap is transformed to contain an exception object of a given type and we assume that the variable `stack0r` holds the reference to it. If a handler for the exception exists in the current method, a jump to the block containing the handler is added. If the exception is caught in a parent frame, a jump to the block that asserts the method's exceptional postcondition for the given exception-type is added.

5.2.6 Translation of bytecode methods

In the following functions we use a `monospaced` font for literals of the translation. Lines beginning with the character "#" do not contain the resulting code, but they describe how the code is generated. The symbol "$\hookleftarrow$" is used to show line breaks in the resulting BoogiePL code.

The function Tr marks the starting point of the translation. The BoogiePL headers `procedure` and `implementation` are generated followed by the result of the functions TrVars that introduces the declarations of the BoogiePL variables, TrInit that does the initialization of the variables, and TrBody, TrBody that translates the method body.

We also introduce two translation functions TrType and TrTypeAbbrev to translate a type (primitive or reference) respectively to its BoogiePL representation and to an abbreviation for use in variable names (e.g. `r` for reference types).

```
$Tr[\![p : Program, m : Method]\!] =$
  procedure $TrSig[\![signature(p, m)]\!] \hookleftarrow$
  implementation $TrSig[\![signature(p, m)]\!] \hookleftarrow$
  $TrVars[\![p, m]\!] \hookleftarrow$
  $TrInit[\![p, m]\!] \hookleftarrow$
  $TrBody[\![p, m]\!] \hookleftarrow$
```

The TrSig function prints the signature of the bytecode method in BoogiePL style. The parameters are numbered from `param` 0 to `param` n.

```
$TrSig[\![ms : MethodSignature]\!] =$
  (
  #for $i := 0, i < |parameters(ms)|, i := i + 1$
      param $i$ : $TrType[\![parameters(ms)[i]]\!]$ ,
  #end for
  )
  #if $result(ms) \neq VoidType$
      returns ( result: $TrType[\![result(ms)]\!]$ )
  #end if
```

As described in the previous subsections, we use variables to model the stack and the heap. From the bytecode verifier we get the maximum size of the operand stack and the number of registers needed for a given method.

- $maxLocals : Program \rightarrow Method \rightarrow Int$
  Returns the number of locals (registers) used by the method. This includes the parameters of the method, as they are being stored in the first register-slots.
- $maxOperandStackSize : Program \rightarrow Method \rightarrow Int$
  Returns the maximum size of the operand stack.

With this information we can create the variables needed for primitive (`int`) and reference types. We also introduce two variables used for translating the `swap` instruction. `arg0` is used to store a copy of the first argument when calling a method. This is needed as this stack location is overwritten with the result of the callee.

```
$TrVars[\![p : Program, m : Method]\!] =$
  #for $i := 0, i < maxOperandStackSize(p, m), i := i + 1$
      var stack $i$ r:  ref, stack $i$ i:  int;↩
  #end for
  #for $i := 0, i < maxLocals(p, m), i := i + 1$
      var reg $i$ r:  ref, reg $i$ i:  int;↩
  #end for
      var swapr:  ref, swapi:  int;↩
      var old_heap:  Store, pre_heap:  Store;↩
      var arg0r:  ref, arg0i:  int;↩
```

TrInit creates a basic block init that assigns the method arguments to the corresponding registers. Note that in case of the instance method the first register always holds the reference to its target (this).

```
$TrInit[\![p : Program, m : Method]\!] =$
  init:↩
      old_heap := heap;↩
      #for $i := 0, i < |parameters(signature(m))|, i := i + 1$
          #if $parameters(signature(m))[i]$ is $RefType$
          assume typ(rval(param $i$ )) <:  $parameters(signature(m))[i]$;↩
          assume alive(rval(param $i$ ), heap);↩
          #end if
          reg $i$ $TrTypeAbbrev[\![parameters(signature(m))[i]]\!]$  := param $i$ ;↩
      #end for
      assume param0 != null;↩
      assume typ(rval(param0)) <:  $parameters(signature(m))[0]$;↩
      assume alive(rval(param0), heap);↩
      goto pre;↩
```

The method body is subsequently translated by the TrBody function. It starts with creation of a helper block pre that assumes the method's precondition. In the case that the first instruction is a jump target, we assert the local annotation (if there is any) and finish this block by jumping to the block starting at the first instruction. The translation of the JVM instructions then starts with the call to TrInstructions. At the end of the translation, the blocks for the method's normal postcondition and its optional exceptional postconditions are generated.

```
$TrBody[\![p : Program, m : Method]\!] =$
  # $p.MST(m) = (R, T, S)$
  # $params = param(0)\ ..\ param(|parameters(signature(m))|\ - 1)$
  pre:↩
      assume $R(old\_heap, params)$ ;↩
  #if $isEdgeTarget(p, m, first_m)$
      #if $isBackEdgeTarget(p, m, first_m)$
```

```
        assert $S(first_m)(old\_heap, params)(heap, \emptyset, regs)$↩
        #end if
        goto block_$first_m$;↩
    #end if
        ↩
    $TrInstructions[\![p, m, first_m]\!]$ ↩
    post: ↩
        assert $T(old\_heap, params)(heap, Normal\ result)$ ;↩
        return;↩
        ↩
    post_X: ↩
        assert $T(old\_heap, params)(heap, Exception\ stack0r)$ ;↩
        return;↩
    #end for
```

TrInstructions is a wrapper for the translation of a single JVM instruction. It uses the control flow graph information to determine if a new block needs to be started. This is the case for any instructions that are targets of jumps.

If we encounter a label in the bytecode that is a target of a back-edge, we add the necessary annotations for the loop, as back-edges are removed. To be able to verify a loop that is executed more than once, we have to ensure that any information about the local state which is available before the loop has been destroyed with havoc. The loop invariant is then assumed to hold.

If the successive bytecode instruction can be reached from other locations, the block has to be ended. At this point we cut the back-edge if there is any.

```
$TrInstructions[\![p : Program, m : Method, pc : PC]\!] =$
    # $p.MST(m) = (R, T, S)$
    #if $isEdgeTarget(p, m, pc)$
        block_ $pc$ : ↩
        #if $isBackEdgeTarget(p, m, pc)$
            havoc $all\ vars \backslash \{old\_heap, this, parami\}$↩
            assume $S(pc)$↩
        #end if
    #end
    $TrInstruction[\![p, m, pc]\!]$
    #if $isEdge(p, m, pc, next_m(pc))$
        #if $isBackEdgeTarget(p, m, jump(pc, o))$
            assert $S(jump(pc, o))(old\_heap, params)(heap, stacks, regs)$↩
        #end if
        #if $isBackEdge(p, m, pc, jump(pc, o))$
            return;↩
        #else
            goto block_ $next_m(pc)$ ;↩↩
        #end if
    #end if
    $TrInstructions[\![p, m, next_m(pc)]\!]$
```

TrInstruction translates a single JVM instruction. Certain instructions may result in creation of successive blocks (e.g. for assuming conditions after a conditional jump), note that a goto statement is added only to the block representing the branch decision, the goto to the block starting at the sequentially next program counter location is added by TrInstructions.

```
$TrInstruction[\![p : Program, m : Method, pc : PC]\!] =$
    #$p.MST(m) = (R, T, S)$
    #$h := getStackHeight(p, m, pc) - 1$
    #switch $instructionAt(p, m, pc)$
```

Integer binary arithmetic

Integer binary arithmetic operations are done by performing simple arithmetic on the corresponding integer stack variables.

#case ***ibinop*** *op* : *AddInt*
```
        stack(h − 1)i := stack(h − 1)i + stackhi;↩
```

Integer division and remainder operations cause an arithmetic runtime exception when dividing by zero. We take this fact into consideration by asserting that the divisor is not zero.

#case ***ibinop*** *op* : *DivInt*
```
        assert stackhi != 0;↩
        stack(h − 1)i := stack(h − 1)i / stackhi;↩
```

Since BoogiePL does not offer bit operations on integers, we use BoogiePL functions and axioms that describe the effect of the bit operation. For example, the effect of the bit left shift operation can be described with a function bit_shl as shown below.

```
function bit_shl(int, int) returns (int);

axiom (∀ i: int :: bit_shl(i, 0) = i);
axiom (∀ i: int, j: int :: 0 ≤ j ⟹ bit_shl(i, j + 1) = bit_shl(i, j) * 2);
```

Pushing constants

Pushing an integer constant is done by assigning a numerical constant to the appropriate stack variable of depth one greater than the current depth.

#case ***iconst*** *n* : *int*
```
        stack(h + 1)i := n;↩
```

Register manipulation

The loading or storing of an integer or reference value to a register is done by assigning the value of the top stack variable to the register variable and vice versa.

#case ***iload*** *n* : *RegNum*
```
        stack (h + 1)i := regni;↩
```

#case ***astore*** *n* : *RegNum*
```
        regnr := stackhr;↩
```

Field access

Field read and write operations are performed by applying the heap functions get and update. Depending on the type of the field, helper functions that convert to or from a BoogiePL type to objects stored in the heap are called. The location on the heap is formed from the object reference on the stack and the field signature (a `name` constant). Assertions prevent accessing a field through a `null` pointer.

```
#case $\boldsymbol{getfield}$ $f : FieldSig$
        assert stack$h$r != null;↩
        #switch $type(f)$
        #case $\$int$
            stack$h$i := toint(get(heap, instvar(stack$h$r, $f$)));↩
        #case $t : RefType$
            stack$h$r := toref(get(heap, instvar(stack$h$r, $f$)));↩
        #end switch

#case $\boldsymbol{putfield}$ $f : FieldSig$
        assert stack$(h-1)$r != null;↩
        #switch $type(f)$
        #case $\$int$
            heap := update(heap, instvar(stack$(h-1)$r, $f$), ival(stack$h$i));↩
        #case $t : RefType$
            heap := update(heap, instvar(stack$(h-1)$r, $f$), rval(stack$h$r));↩
        #end switch
```

Object allocation

Allocating an object with the instruction `new` is translated to applications of the heap functions new and add. The top stack item is assumed to hold a reference to an object of the given reference type after the instruction completes.

```
#case $\boldsymbol{new}$ $t : RefType$
        havoc stack$(h+1)$r;↩
        assume new(heap, $t$) == rval(stack$(h+1)$r);↩
        heap := add(heap, $t$);↩
```

Conditional and unconditional branches

Conditional branch instructions for integers and references are translated to non-deterministic jumps following assumptions depending on the branch condition. In this translation, *TrCond* yields the BoogiePL comparator for a given (Bicolano-)condition.

In order to simplify reading, we introduce a function blockEnd that emits the translation for a jump to a given PC. The generated BoogiePL code removes back-edges in the CFG.

```
$blockEnd(pc' : PC)$ =
    #if $isBackEdgeTarget(p, m, pc')$
        assert $S(pc')(old\_heap, params)(heap, stacks, regs)$↩
    #end if
    #if $isBackEdge(p, m, pc, pc')$
        return;↩
    #else
        goto block_ $pc'$ ;↩↩
    #end if
```

```
#case $\boldsymbol{if\_icmp}$ $cond : IntCond,\ o : Offset$
        goto block_$pc$_True, goto block_$pc$_False;↩
        ↩
        block_$pc$_True:↩
            assume stack$(h-1)$i $TrCond[\![cond]\!]$ stack$h$i;↩
            $blockEnd(jump(pc, o))$
        ↩
```

```
    block_$pc$_False:$\hookleftarrow$
        assume !(stack$(h-1)$i $TrCond[\![cond]\!]$ stack$h$i);$\hookleftarrow$
```

Return statements assign to a designated `result` variable and jump to the block which asserts the current method's normal postcondition. For methods that do not return anything, only a `goto` is required.

```
  #case ireturn
        result := stack$h$i;$\hookleftarrow$
        goto post;$\hookleftarrow$
```

The `goto` instruction, if it is not a back-edge, leads to a branch to the block that contains the translation of the instructions starting at the resulting offset. Here too, we assert the local annotation of the target if necessary.

```
  #case goto $o : Offset$
        $blockEnd(jump(pc, o))$
```

Method invocation

The most interesting point in the translation of method calls is the way exceptions are dealt with. Two blocks are generated. In `block_`pc`_Exception:`, we assume that the callee has thrown an exception. Exceptions that are caught in the caller method lead to a jump to the block of their handler. Exceptions with no such handler lead to the creation of a jump to a BoogiePL block which corresponds to the exceptional postcondition of the caller method. In this block, we state that `stack0r` contains an object that is a subtype of java.lang.Throwable. This could be more specific by using the information in the throws-clause of the method signature of the callee. But as the underlying VC generator does not use this information, we can safely also do so without creating a source of unsoundness.

If the method has a non-`void` return type, the `havoc` instruction is performed over the top stack item to indicate that it has changed to something arbitrary.

The heap state is lost which is modelled by the use of the `havoc` statement on the heap variable. The variables `pre_heap` and `arg0i/r` are used to save the state before calling the method.

```
  #case invokestatic $target : Method$
    #$p.MST(target) = (target\_R, target\_T, \_)$
        arg0 $TrTypeAbbrev[\![parameters(target)[i-1]]\!]$  :=
            stack $(h - |parameters(target)| + 1)$ $TrTypeAbbrev[\![parameters(target)[0]]\!]$ ;$\hookleftarrow$
        #$args = (arg0i/r, stack(h - |parameters(target)| + 2)..stack(h))$
        assert $target\_R\ (pre\_heap, args)$$\hookleftarrow$
        pre_heap := heap;$\hookleftarrow$
        havoc heap;$\hookleftarrow$
        assume (forall v:  Value ::  alive(v, pre_heap) ==> alive(v, heap));$\hookleftarrow$
        goto  block_$pc$_Normal; block_$pc$_Exception;
        $\hookleftarrow$
    block_$pc$_Exception:$\hookleftarrow$
        havoc stack0r;$\hookleftarrow$
        assume alive(rval(stack0r), heap);$\hookleftarrow$
        assume typ(rval(stack0r)) <:  javaLangThrowable;$\hookleftarrow$
        assume $target\_T\ (pre\_heap, args)(heap, Exception\ stack0r)$$\hookleftarrow$
        # $pc' = lookupHandler(m, pc, typ(rval(stack0r)))$
        $blockEnd(pc')$
        $\hookleftarrow$

    block_$pc$_Normal:$\hookleftarrow$
        #if $returnType(target) \neq Void$
```

```
#$first\_param$ =stack($h - |parameters(target)|$)$TrTypeAbbrev[\![returnType(target)]\!]$
havoc  $first\_param$ ;$\hookleftarrow$
assume $target\_T\ (pre\_heap, args)(heap, Normal\ first\_param)$$\hookleftarrow$
#else
assume $target\_T\ (pre\_heap, args)(heap, Normal\ None)$$\hookleftarrow$
#end if
```

Throwing exceptions

The `athrow` instruction throws an exception object referenced by the top item on the operand stack. The smallest common super-type of the top stack is known by the bytecode verifier.

```
#case athrow
        assert stack$h$r != null;$\hookleftarrow$
        stack0r := stack$h$r;$\hookleftarrow$
        # $pc' = lookupHandler(m, pc, typ(rval(stack0r)))$
        $blockEnd(pc')$
        $\hookleftarrow$
```

5.2.7 Soundness of the translation

Preliminaries

The starting point of the VC generator for MOBIUS base logic is an annotated program (p, $subclass$, MST). We assume that the translation to BoogiePL starts from the same input. That is, we have exactly the method specifications and local specifications that are contained in MST.

We make here a number of simplifying assumptions, mainly to focus on the most interesting aspects:

A0: The CFG of each bytecode method is reducible (or made reducible by code duplication before verification).

A1: Loop invariants and local preconditions:

1. For each pc that is the sink of a backward edge in the bytecode CFG, the specification table contains a local precondition for the loop invariant.
2. For all pc that are not sinks of backward edges, the specification table does not contain a local precondition.

A2: We use an untyped version of BoogiePL. That is, we omit BoogiePL types and casts.

A3: We assume that the VC generator and BoogiePL use exactly the same store model, even though the VC generator model cannot directly be encoded in BoogiePL.

A4: We assume that the following variables can be used in preconditions and postconditions:

1. Preconditions may mention the formal parameters and the heap.
2. Postconditions may mention the formal parameters and the heap of the prestate as well as the heap of the poststate and "result" (provided that the method is not void) or an exception object reference.
3. Local preconditions may mention the formal parameters and the heap of the prestate as well as the heap, the locals, and the operand stack of the current state.

A5: The bytecode program passed the Java bytecode verifier. We can assume everything that the bytecode verifier guarantees.

A6: The precondition of each method includes the following three requirements:

1. The variable "param0" is alive and not null in case of an instance method.

2. All parameters of a method are alive.
3. The actual type of a parameter is a subtype of the declared type.

Mapping between state

Assertions in the VC generator take one or two states as input:

$$\begin{array}{ll} \mathit{InitState}: & \mathit{Heap} \times \mathit{Locals} \\ \mathit{TermState}: & \mathit{Heap} \times \mathit{Result} \\ \mathit{LocalState}: & \mathit{Heap} \times \mathit{OperandStack} \times \mathit{Locals} \end{array}$$

Assertions in BoogiePL use local variables. When applying a VC generator state to a BoogiePL formula, we implicitly perform the following substitutions:

For $\mathit{InitState}(h, l)$
 `old_heap` $\leftarrow h$
 `parami` $\leftarrow l(i)$ for all arguments p of the method (where this = `param0` in instance methods)

For $\mathit{TermState}(h, r)$
 `heap` $\leftarrow h$
 `result` $\leftarrow r$

For $\mathit{LocalState}(h, os, l)$
 `heap` $\leftarrow h$
 `regi` $\leftarrow l(i)$ for all arguments p and locals of the method
 `stacki` $\leftarrow os(i)$ for all stack elements

The soundness theorem

The definition $\mathsf{CertMethod}_{\mathsf{BPL}}$ below expresses that a method can be verified in the BoogiePL calculus. Since we do not use the `requires` and `ensures` clauses of BoogiePL, the weakest precondition of the method is equivalent to true. Moreover, to satisfy the VC generator obligation for local annotations, we require that the weakest precondition holds for each block that starts with the translation of a local annotation, although we believe that this is not necessary to be sound with respect to the base logic.

The translation of a local annotation is "`havoc` s; `assume` $S(pc)$" where s stands for all variables used to map the *LocalState* to BoogiePL. As `havoc` s does not occur on other occasions, we can use this to identify a translated local annotation.

Definition 5.2.1 (Certified methods)

$$\begin{array}{ll} \forall(\mathit{bpl} : \mathit{BoogiePLProg}, m : \mathit{Method}). & \\ \mathsf{CertMethod}_{\mathsf{BPL}}(\mathit{bpl}, m) \iff & (\forall(s0, s).\ \mathsf{wp}_{\mathsf{b}}(\mathit{bpl}, m, \mathsf{init}_{m_{bpl}})(s0, s)) \wedge \\ & (\forall(s0, s, \mathit{pos}).\ \mathit{stmt}(\mathit{bpl}, m, \mathit{pos}) = \mathtt{havoc}\ s \Longrightarrow \mathsf{wp}_{\mathsf{b}}(\mathit{bpl}, m, p)(s0, s)) \end{array}$$

The soundness theorem we prove is the following.

Theorem 5.2.2 (Soundness of BoogiePL translation) For every method m of an annotated program P, if the translation of m can be verified in BoogiePL, then m can be verified in the VC generator:

$$\begin{array}{l} \forall(P : \mathit{AnnotProg}). \\ \quad (\forall(m : \mathit{Method}).\mathsf{CertMethod}_{\mathsf{BPL}}(\mathit{Tr}(P), m) \Longrightarrow \mathsf{CertMethod}_{\mathsf{VC}}(P, m)) \end{array}$$

In order to prove the soundness theorem, we have to show that the following lemma holds.

Lemma 5.2.1 (Weakest preconditions of instruction sequences) The translation of a bytecode instruction (sequence) requires a stronger weakest precondition than the original instruction (sequence).

$$\begin{array}{l} \forall(P, m, pc, s0, s). \\ \quad (\mathsf{wp}_{\mathsf{b}}(\mathit{Tr}(P), m, \mathit{instrpos}(\mathit{Tr}(P), m, pc))(s0, s) \Longrightarrow \mathsf{wp}_{\mathsf{i}}(P, m, pc)(s0, s)) \end{array}$$

where

$$instrpos(Tr(P), m, pc) = \begin{cases} position(Tr(P), m, pc) + 2 & : \text{ if } isBackEdgeTarget(P, m, pc), \\ position(Tr(P), m, pc) & : \text{ otherwise.} \end{cases}$$

In other words: $instrpos(Tr(P), m, pc)$ is the position of the first BoogiePL statement that comes from the translation of the bytecode instruction at pc after the possibly inserted translation of the loop invariant.

A proof sketch of the soundness theorem and the presented lemma can be found in appendix E.

Chapter 6

Related and future work

We have presented here the formal foundations for verification and specification of Java bytecode programs which are the result of Task 3.1. These foundations are based on Bicolano — a model of the Java Virtual Machine and the bytecode language. This model has been further extended with the MOBIUS base logic which is a program logic for bytecode programs. On the basis of the logic, we have built the semantics of the specification language BML (Bytecode Modeling Language) and the procedures to generate verification conditions which will be used in the tool set developed within the MOBIUS project.

We conclude with a short sketch of the main lines of the further development within the MOBIUS project and a description of scientific efforts outside of the MOBIUS project which are related to the development of Task 3.1.

6.1 Tool development

It is foreseen that the MOBIUS tool set developed in Tasks 3.6 and Task 4.4 will provide the following functionalities that will use the formalisms developed in the Task 3.1:

- a translator from bytecode programs augmented with specifications into the Bicolano semantics;
- a tool to inspect BML specifications;
- a verification condition generator for the MOBIUS base logic;
- an implementation of the translation into BoogiePL as defined in section 5.2;
- a compiler that compiles source code and JML specifications into the corresponding bytecode and BML specifications and preserves the related proofs;
- proof assistant tactics to ease verification in the Coq proof assistant.

The formalisation of the VC generator for the MOBIUS base logic is done so that it can produce verification conditions for actual programs. Thus, we can treat this as a prototype of the final VC generator. A prototype version of the translator from bytecode to Bicolano has already been developed. Moreover, a prototype compiler of JML to BML, `jml2bml`, has been implemented for a subset of BML (see [17]). We plan to redesign and extend the compiler and the other prototype tools in the future course of the MOBIUS project.

6.2 Extensions

The operational semantics developed in this task models only the core behaviour of the bytecode executed by the Java Virtual Machine. The most notable omissions are the bounded size of the memory and the multi-threading. We plan to extend the Bicolano semantics to cover these aspects of the bytecode execution in Tasks 3.2 and 3.3 respectively.

Task 3.2 is going to enrich the Bicolano semantics to tackle the resource usage for the Java Virtual Machine. This extension can take into account instruction counts, stack depth, heap size etc. Task 3.3 will provide formalisms to deal with the multi-threading semantics based on the Java Memory Model [31] and prove that the semantics is equivalent to the traditional interleaving semantics in case of properly synchronised programs. In particular this formalisation will concern the synchronisation primitives as well as the semantics of the access to volatile and final fields.

The MOBIUS base logic as presented in chapter 3 covers a large subset of JVM language. Only a few instructions that are modelled in Bicolano have not yet been covered, namely the switch instruction and some further method invocation instructions. These will be added in the near future. In addition, we plan to extend the logic by including a component for reasoning explicitly about resources and execution traces, and by including ghost variables and fields. These extensions are part of Task 3.2 and will be addressed jointly as the differences between the two tasks appear small (programmable vs non-programmable get/set operations).

A final issue concerns the completeness of the program logic. As the current formalisation in Coq uses a shallow embedding of the assertions, the appropriate notion for formalising completeness would be Cook's relative completeness w.r.t. the internal logic of Coq [19]. Given our own previous experience, and the work of Kleymannn and Nipkow, a proof of relative completeness appears certainly feasible. As completeness is not required for the trustworthiness of the MOBIUS architecture, however, we currently do not plan to carry out a formalisation of this property.

6.3 Related work

Specification languages for low level languages The specification language BML, the formal semantics of which was developed in the work of the task, was inspired by the JML specification language [34]. JML is a realisation of the Design by Contract principle introduced first by Meyer in Eiffel [39]. Another realisation of the Design by Contract principles is th language Spec# and Boogie tool set [6] which bring the ideas into the C# programming language setting.

One of the features of BML — the possibility to assert and check during runtime the constraints that must hold each time the control of a program reaches a particular place, has a long history as it was inspired by Dijkstra's guarded command language [23]. This `assert` construct was a long time ago adopted in the C programming language and lately in Java 5 (section 14.10 of [27]). A recent preliminary proposal of Extended Virtual Platform (EVP) advocates a strong dedicated support for assertions and parametric types at the level of the virtual machine [2].

Operational semantics for the bytecode language The meaning of BML is described in terms of the operational semantics of JVM formulated in Coq. There were several other formalisations of the machine. For instance, Stärk et al. [50] provide a high-level description, together with a mathematical and an experimental analysis, of Java and of the Java Virtual Machine. The formalisation is based on the Abstract State Machines and is done on paper. Wildmoser et al. [53] describe verification of annotated bytecode programs, in particular they propose an incremental verification of verification conditions. The authors formalise in their work a restricted fragment of JVM and show how results from a trusted type analyser may be combined with untrusted interval analysis to automatically verify that bytecode programs do not overflow. All trusted components are formalised and verified in Isabelle/HOL. A formalisation of JVM was also done within the Bali project [45]. This formalisation included a proof of its type-safety. Furthermore, a bytecode verifier and lightweight bytecode verification have been verified in the project [41]. Another formalisation of a representative subset of JVM and its bytecode verifier was done in Isabelle/HOL by Klein [32]. A formalisation of JVM in ACL2 was proposed by Moore et al. [40]. The group argues that a practical way to apply formal methods to the Java programming language is to apply formal methods to a formalisation of the JVM directly. The work for Java Card platform, which uses a significantly reduced version of the Java Virtual Machine, also resulted in formalisation of Java related tools and languages. In particular, the SecSafe project gave an operational semantics for the Java Card Virtual Machine Language [49]. Also, the work by Dufay includes a formalisation of the Java Card platform in Coq [24]. This formalisation includes a bytecode verifier.

Sometimes, a bytecode verifier is a part of a formalisation of a version of the Java Virtual Machine. A comprehensive description of the bytecode verification algorithms was presented by Leroy [37]. In particular, within the work the author surveys the use of proof assistants to specify bytecode verification and prove its correctness.

MOBIUS base logic The necessity of complementing partial-correctness assertions by guarantees that apply to intermediate states and non-terminating computations has also been observed by Hähnle and Mostowski [28]. Starting from an extension of first-order dynamic logic with trace modalities [10], they discuss the verification of transaction properties in the context of Java Card applications. Similar requirements arise from object invariants [36] and idioms like the validity of objects in ESC/Java [25]. Similarly to these logics, the logics developed in connection with the LOOP tool (e.g. [29]) operate at the source code level. More precisely, they work with a representation of the source code and JML specifications in a theorem prover. Various termination modes are considered, but some rules, such as the rule for while, can only be applied in special circumstances. The logic is formulated as a set of derived proof rules, so proof search may always fall back on the underlying operational semantics. In contrast, our formulation as a syntactic proof system admits a study of (relative) completeness, following the approach taken by Kleymann, Nipkow, and ourselves [33, 42, 4].

Benton's logic [12] includes (basic) type information in judgements, extending bytecode verification conditions. Consequently, methods can be given more modular specifications that, for example, constrain the heap to the areas relevant for the verification of the method body, similarly to Separation Logic [48]. In our approach, such local reasoning principles would be formulated in the interpretation of type judgements, i.e. in derived proof rules [14]. As a further difference, Benton's logic is interpreted extensionally, by reference to program contexts. This enables Benton to prove that certain program transformations are semantics-preserving (see also [11]), while we aim to certify code-inherent properties, including intensional properties such as the consumption of resources [?].

The need to use the deep embedding for a program logic is advocated in the paper [54] by Wildmoser and Nipkow. The use of de Bruijn notation for logical variables in the development of the deep embedding of the assertion language was inspired by Stefan Berghofer's solution to the POPLMark challenge, which may be viewed as deep embedding of the polymorphic λ-calculus [13].

Formalisms for verification tools Two notable approaches were developed that use an imperative intermediate language to verify a program automatically. Neither approach proves soundness of the translated output with respect to the operational semantics of the original program.

On the one hand, we have the Spec# Environment [8] introducing an annotated C# language called "Spec#". In this approach, Spec# is translated to the .NET CIL together with annotations. The CIL is then verified by translating it into BoogiePL [21] and then into verification conditions that are sent to the automatic prover Simplify [22]. The counterexample that Simplify might find is then transformed back into an error message for Spec# source code. The translation from annotated CIL to BoogiePL is similar to what we present in section 5.2, but it has two main differences: it uses a different heap model and it does not include exception handling. The latter point is one of the main contributions of the work done in this task.

On the other hand, we have ESC/Java and ESC/Java2 [18, 25]. In this approach, JML-annotated Java source code is the origin language for the verification process. It is transformed to a guarded commands language that is suitable only for structured programs. From there, different backends can be used to generate verification conditions for different provers. Compared to the original guarded commands language from Dijkstra [23], ESC/Java2 defines a weakest precondition calculus that also defines the behaviour for exceptional program states. Therefore, this approach includes exception handling. In contrast to ESC/Java2, we want to use annotated Java bytecode as the starting point and not source code. Latest developments in the backend

Flanagan and Saxe [26] showed how to generate verification conditions that are linear in size. Barnett and Leino [7] adapt this approach in their verification condition generation for BoogiePL programs.

Bibliography

[1] Java^{tm} 2 platform standard edition 5.0 API specification. Sun Microsystems, Inc., 2004. http://java.sun.com/j2se/1.5.0/docs/api/index.html.

[2] S. Alagi and M. Royer. Next generation of virtual platforms. Article in `odbms.org`, October 2005. `http://odbms.org/about_contributors_alagic.html`.

[3] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. In F. Spoto, editor, Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005), volume 141 of Electronic Notes in Theoretical Computer Science, pages 255–273. Elsevier, December 2005.

[4] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In Theorem Proving in Higher-Order Logics, volume 3223 of Lecture Notes in Computer Science, pages 34–49, Berlin, September 2004. Springer-Verlag.

[5] F. Y. Bannwart and P. Müller. A program logic for bytecode. Electronic Notes in Theoretical Computer Science, 141:255–273, 2005.

[6] 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, volume 4111 of Lecture Notes in Computer Science. Springer-Verlag, 2005.

[7] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In Program Analysis For Software Tools and Engineering, pages 82–87, New York, NY, USA, 2005. ACM Press.

[8] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [9], pages 151–171.

[9] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop, volume 3362 of Lecture Notes in Computer Science. Springer-Verlag, 2005.

[10] B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, International Joint Conference on Automated Reasoning, volume 2083 of Lecture Notes in Computer Science, pages 626–641. Springer-Verlag, 2001.

[11] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, Principles of Programming Languages, pages 14–25. ACM Press, 2004.

[12] N. Benton. A typed, compositional logic for a stack-based abstract machine. In K. Yi, editor, Programming Languages and Systems, Third Asian Symposium, APLAS'05, Tsukuba, Japan, November 2-5, 2005, Proceedings, volume 3780 of Lecture Notes in Computer Science, pages 364–380. Springer-Verlag, 2005.

[13] S. Berghofer. A solution to the POPLMark challenge in Isabelle/HOL. Slides of the talk on TYPES'06 Workshop `http://www4.in.tum.de/~berghofe/`, 2006.

[14] L. Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In Logic for Programming Artificial Intelligence and Reasoning, volume 3452, pages 347–362. Springer-Verlag, 2005.

[15] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. Science of Computer Programming, 55:53–80, 2005.

[16] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Workshop on Formal Methods for Industrial Critical Systems, volume 80 of Electronic Notes in Theoretical Computer Science, pages 73–89. Elsevier, 2003.

[17] L. Burdy and M. Pavlova. Java bytecode specification and verification. In Symposium on Applied Computing, pages 1835–1839. ACM Press, 2006.

[18] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In Barthe et al. [9], pages 108–128.

[19] S. A. Cook. Soundness and completeness of an axiom system for program verification. SIAM J. Comput., 7(1):70–90, February 1978. A corregendium appears in [?].

[20] S. A. Cook. Corrigendum: Soundness and completeness of an axiom system for program verification. SIAM J. Comput., 10(3):612, 1981.

[21] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[22] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. Journal of the Association of Computing Machinery, 52(3):365–473, 2005.

[23] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, 18(8):453–457, 1975.

[24] G. Dufay. Vérification formelle de la plate-forme Java Card. PhD thesis, Université de Nice Sophia-Antipolis, December 2003.

[25] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In Programming Languages Design and Implementation, volume 37, pages 234–245, June 2002.

[26] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In Principles of Programming Languages, pages 193–205, New York, NY, USA, 2001. ACM Press.

[27] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, Third Edition, chapter 17. Sun Microsystems, 2005. http://java.sun.com/docs/books/jls/.

[28] R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, pages 151–171, 2004.

[29] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, Fundamental Approaches to Software Engineering, volume 2029 of Lecture Notes in Computer Science, pages 284–299. Springer-Verlag, 2001.

[30] C. B. Jones. Systematic software development using VDM. Prentice Hall, 1990.

[31] JSR 133: Java memory model and thread specification, 2004.

[32] G. Klein. Verified Java Bytecode Verification. PhD thesis, Institut für Informatik, Technische Universität Mnüchen, 2003.

[33] T. Kleymann. Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs. PhD thesis, LFCS, University of Edinburgh, 1998.

[34] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML Reference Manual, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[35] H. F. Ledgard. A human engineered variant of BNF. ACM SIGPLAN Notices, 15:57–62, 1980.

[36] K. R. M. Leino and R. Stata. Checking object invariants. Technical Report #1997-007, Digital Equipment Corporation Systems Research Center, Palo Alto, USA, 1997.

[37] X. Leroy. Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning, 30(3-4):235–269, December 2003.

[38] T. Lindholm and F. Yellin. The Java^{TM} Virtual Machine Specification. Second Edition. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[39] B. Meyer. Object-Oriented Software Construction. Prentice Hall, 2nd revised edition, 1997.

[40] J. S. Moore, R. Krug, H. Liu, and G. Porter. Formal models of Java at the JVM level. a survey from the ACL2 perspective. In Workshop on Formal Techniques for Java Programs, June 2001.

[41] T. Nipkow. Verified bytecode verifiers. In M. Miculan and F. Honsell, editors, Foundations of Software Science and Computation Structures, volume 2030 of Lecture Notes in Computer Science, pages 347–363. Springer-Verlag, 2001.

[42] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, Computer Science Logic, volume 2471 of Lecture Notes in Computer Science, pages 103–119. Springer-Verlag, 2002.

[43] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL. A Proof Assistant for Higher-Order Logic. Springer-Verlag, 2002.

[44] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[45] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Technical University Munich, 1998.

[46] C. L. Quigley. A Programming Logic for Java Bytecode Programs. PhD thesis, University of Glasgow, 2004.

[47] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005.

[48] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In Logic in Computer Science, pages 55–74. IEEE Computer Society, 2002.

[49] I. Siveroni and C. Hankin. A proposal for JCVMLe operational semantics. Technical Report SECSAFE-ICSTM-001, Imperial College London, 2001.

[50] R.F. Stärk, J. Schmid, and E. Börger. Java and the Java Virtual Machine—Definition, Verification, Validation. Springer-Verlag, 2001.

[51] Sun Microsystems Inc., 4150 Network Circle, Santa Clara, California 95054. Mobile Information Device Profile for Java^{TM} 2 Micro Edition, November 2002. http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html.

[52] Sun Microsystems Inc., 4150 Network Circle, Santa Clara, California 95054. Connected Limited Device Configuration.Specification Version 1.1. Java^{TM} 2 Platform, Micro Edition (J2ME^{TM}), March 2003.

[53] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, Bytecode Semantics, Verification, Analysis and Transformation, volume 141 of Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[54] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, Theorem Proving in Higher-Order Logics, volume 3223 of Lecture Notes in Computer Science, pages 305–320. Springer-Verlag, 2004.

Appendix A

Detailed description of the Bicolano specification

The purpose of this appendix is to give a detailed description of how the JVM specification is interpreted in Bicolano. It also gives more precise information about which parts of the JVM are (not) covered by Bicolano. It can serve as a reference for those who would like to use the semantics for particular purpose and are interested if the semantics covers the intended scope. The structure of this chapter reflects the structure of chapter 3 in the JVM specification [38]. We assume the reader is familiar with the terminology and structure of the chapter and encourage reading of this appendix in parallel with the chapter.

A.1 The `class` file format

Bicolano does not read nor write class files. The binary format of class files is not meaningful to Bicolano. Instead, the semantics operates on programs after linking. The logical structure of programs and classes is specified in the Bicolano types `Program` and `Class` (file `Program.v`).

A.2 Data types

Bicolano follows the JVM specification in introducing a distinction between reference types and primitive types. However, it does not explicitly introduce the null type. Instead it allows the reference variables to assume the value null.

Relevant type names: `PROGRAM.type`, `DOMAIN.value`

Type definitions

```
  Inductive type : Set :=
      | ReferenceType (rt : refType)
      | PrimitiveType (pt: primitiveType)
  with refType :Set :=
      | ArrayType (typ:type)
      | ClassType  (ct:ClassName)
      | InterfaceType (it:InterfaceName)
  with
    primitiveType : Set :=
      | BOOLEAN  | BYTE | SHORT | INT.

 Inductive value : Set :=
   | Num : num → value
   | Ref: Location → value
   | Null : value.
```

Relevant operations

- `init_value : type →value`
  Returns the initial value of the given type (0 for numeric types, null for reference type).

Differences with the JVM specification The types `char`, `float`, `double`, `long` are not specified in Bicolano.

A.3 Primitive types and values

A.3.1 Integral types and values

Relevant type names: The module type `NUMERIC` is a common interface for all integral data types. The modules `Byte`, `Short`, `Int` in `Domain.SEMANTIC_DOMAIN` are instances of `NUMERIC` with the appropriate number of bits. The types of integral values are `Byte.t`, `Short.t`, `Int.t` in `Domain.SEMANTIC_DOMAIN`.

Relevant operations

- `toZ : NUMERIC.t →Z`
  Converts the given value to a Coq integer type (`Z`).
- `range : Z →Prop`
  Checks whether the given value in the range of the integral type.
- `smod : Z →Z`
  Returns the argument `mod` the base.
- Several basic arithmetic operations inside the type (e.g. `add`, `shr`, `xor`, etc.)
- `b2i : Byte.t →Int.t`
  Converts a byte value directly to an integer value.
- `i2b : Int.t →Byte.t`
  Converts an integer value to a byte value.
- Similarly for other integral values (`s2i`, `i2s`).

Properties Every value of a type is in its range.

```
Parameter range_prop : ∀ x:t, range (toZ x).
```

There are also axioms for arithmetic operations, i.e.:

```
Parameter mul_prop : ∀ i1 i2,
  toZ (mul i1 i2)= smod (toZ i1 * toZ i2).
```

A.3.2 Floating-point types, value sets and values

Differences with the JVM specification Floating-point types are not implemented in Bicolano.

A.3.3 The `returnAddress` type and values

Following the CLDC specification, the instructions `jsr`, `ret`, `jsr_w` are not specified in Bicolano, because Bicolano does not support subroutines. Therefore, also the return address value is not modelled in Bicolano.

A.3.4 The `boolean` type

The `BOOLEAN` data type is one of the allowed values of Bicolano's `primitiveType` inductive type. It specifies the boolean type of fields and method parameters.

However, there is no corresponding type that could hold boolean values directly. The only conversion related to the boolean type is:

```
Parameter i2bool : Int.t → Byte.t.
```

used when a boolean value is written to a boolean array.

The JVM specification defines that the `integer` type should be used in all places, where operands expect boolean values, except for boolean arrays, which are handled in the same way as byte arrays (`baload` and `bastore` operations should be used).

Following the JVM specification, Bicolano uses `0` to represent `false` and `1` to represent `true`.

A.4 Reference types and values

Relevant type names: `DOMAIN.Location`.

Type definitions

```
Parameter Location : Set.
```

Relevant operations The `Location` type is closely related to Heap. See A.5.3 for description of heap operations.

A.5 Runtime data areas

This section explains how the different JVM runtime data areas are represented in Bicolano. The following subsections describe the particular structures, as they are defined in the JVM specification.

First, we describe the overall JVM state.

Relevant type names: `DOMAIN.STATE.t`

Type definitions

```
Inductive t : Set :=
  normal : Heap.t → Frame.t → CallStack.t → t
| exception : Heap.t → ExceptionFrame.t → CallStack.t → t.
```

Differences with the JVM specification Bicolano implements only a single-threaded JVM, therefore there is only copy in the state of the per-thread structures of the JVM (program counter, JVM stack).

A.5.1 The `pc` register

Relevant type names: `PROGRAM.PROGRAM.PC`

Type definitions

```
Parameter PC : Set.
```

Relevant operations

- `jump : PC →PROGRAM.OFFSET.t →PC`
  Returns the position in the code after jumping from the given position by the given offset.
- `firstAddress : BytecodeMethod →PC`
  Returns the pointer to the first instruction of the given method.
- `nextAddress : BytecodeMethod →PC →option PC`
  Returns `Some pc`, where `pc` is the position of the next instruction in the given method. If the given position is out of range or has no following instruction then `None` is returned.
- `instructionAt : BytecodeMethod →PC →option Instruction`
  Returns `Some i`, where `i` is the instruction that has the given position in the given method. If there is no such instruction then `None` is returned.
- `next : Method →PC →option PC`
  Returns `Some pc`, where `pc` is the position of the next instruction in the given method. If the method has no body or the given position is out of range or there is no following instruction then `None` is returned.

Location in the formalisation The current program counter is a part of the current frame. Every frame in the frame stack contains one program counter.

type	constructor	parameter
`DOMAIN.FRAME.t`	`make`	2nd

A.5.2 Java Virtual Machine stacks

Relevant type names: `DOMAIN.CALLSTACK.t`

Type definitions

```
Definition t : Set := list Frame.t.
```

Location in the formalisation The current JVM stack is the third field of a (non-exceptional) state.

type	constructor	parameter
`DOMAIN.STATE.t`	`make`	3rd

A.5.3 Heap

Relevant type names: `DOMAIN.HEAP.t`, `DOMAIN.HEAP.AdressingMode`, `DOMAIN.HEAP.LocationType`

Type definitions

```
Parameter t : Set.

Inductive AdressingMode : Set :=
  | StaticField : FieldSignature → AdressingMode
  | DynamicField : Location → FieldSignature → AdressingMode
  | ArrayElement : Location → Int.t → AdressingMode.

Inductive LocationType : Set :=
  | LocationObject : ClassName → LocationType
  | LocationArray : Int.t → type → LocationType.
```

Relevant operations

- `get : HEAP.t →AdressingMode →option value`
  Returns `Some v`, where `v` is the value at the given address in the given heap, or `None` if there is no value at that address.
- `update : HEAP.t →AdressingMode →value →HEAP.t`
  Returns the given heap with the given value assigned to the given address.
- `typeof : t →Location →option LocationType`
  Returns the (dynamic) type of the value stored at the given location, or `None` if there is no value stored.
- `new : t →Program →LocationType →option (Location * t)`
  Returns a fresh location of the given type, or `None` if no fresh location has been allocated.

Location in the formalisation The heap is a part of the state (notice that also for the multi-threaded JVM, there is only one heap).

type	constructor	parameter
`DOMAIN.STATE.t`	`make`	1st

A.5.4 Method area

The access to methods and instructions is available through the functions described below.

Relevant operations

- `PROGRAM.METHOD.body : Method →option BytecodeMethod`
  Returns the body of a method. If the method has no body (it is abstract), `None` is returned.
- Further manipulation of methods and instructions is done via the functions for the program counter, as described above.

A.5.5 Runtime constant pool

Differences with the JVM specification The runtime constant pool is not explicitly handled in Bicolano as the semantics is described for a linked program. The JVM uses the runtime constant pool, among other things, for dynamic linking (delayed classes loading). As this is not allowed on the CLDC platform, this is outside the scope of interest for the MOBIUS project; and therefore it has not been modelled in Bicolano.

A.5.6 Native method stacks

Differences with the JVM specification Bicolano currently does not support native methods. This is motivated by the fact that on the CLDC platform, one cannot extend the set of native methods. However, to extend Bicolano to multi-threading, it will have to model several native methods from the standard API, to model thread creations, thread start etc.

A.6 Frames

This section describes the representation of frames in Bicolano. Below, we the describe the different elements of a frame, as defined in the JVM specification. First, we describe frames themselves.

Relevant type names: `DOMAIN.FRAME.t`

Type definitions

```
Inductive t : Set :=
  make : Method → PC → OperandStack.t → LocalVar.t → t.
```

Location in the formalisation The JVM state contains one current frame and zero or more frames in the call stack.

type	constructor	parameter
`DOMAIN.STATE.t`	`make`	2nd
`DOMAIN.CALLSTACK.t`	this is a list of frames	

A.6.1 Local variables

Relevant type names: `DOMAIN.LOCALVAR.t`

Type definitions

```
Parameter t : Set.
```

Relevant operations

- `get : LOCALVAR.t →Var →option value`
  Returns `Some v`, where `v` is the value of the given local variable, or `None` if there is no such variable.
- `update : LOCALVAR.t →Var →value →LOCALVAR.t`
  Returns the given local array with the given value set at the given index.

Location in the formalisation   Every frame contains one array of local variables.

type	constructor	parameter
`DOMAIN.FRAME.t`	`make`	4th

Differences with the JVM specification   Bicolano arrays are indexed with the abstract type `Var`, while the JVM specification uses ordinary non-negative integers.

A.6.2 Operand stacks

Relevant type names: `DOMAIN.OPERANDSTACK.t`

Type definitions

```
Definition t : Set := list value.
```

Location in the formalisation   Operand stack is part of the current frame and of each frame on the JVM stack.

type	constructor	parameter
`DOMAIN.FRAME.t`	`make`	3rd

A.6.3 Dynamic linking

Dynamic linking is outside the scope of interest in Bicolano, the semantics is defined for the post-linking state.

A.6.4 Normal method invocation completion

Bicolano specifies normal method invocation completion according to the JVM specification.

1. Invocation of a method causes putting the current frame on the frame stack and creating a new current frame. Consider for instance the semantics of the `invokestatic` instruction:

```
| invokestatic_step_ok : ∀ h m pc s l sf mid M args bM fnew,

  instructionAt m pc = Some (Invokestatic mid) →
  findMethod p mid = Some M →
  length args = length (METHODSIGNATURE.parameters mid) →
  METHOD.body M = Some bM →
  fnew = (Fr M
             (BYTECODEMETHOD.firstAddress bM)
              OperandStack.empty
             (stack2localvar (args++s)  (length args))) →
  step p (St h (Fr m pc (args++s) l) sf) (St h fnew ((Fr m pc s l)::sf))
```

2. Returning from a method causes replacing the current frame by the frame on the top of the frame stack and updating the program counter. If the called method returns a value, it is put on top of the operand stack of the current frame. Consider for instance the semantics of the `ireturn` instruction:

```
| ireturn_step_ok : ∀ h m pc s l sf i tm m' pc' pc'' l' s',

  instructionAt m pc = Some Ireturn →
  next m' pc' = Some pc'' →
  METHODSIGNATURE.result (METHOD.signature m) = Some (PrimitiveType tm) →

  step p (St h (Fr m pc (Num (I i)::s) l) ((Fr m' pc' s' l')::sf))
                         (St h (Fr m' pc'' (Num (I i)::s') l') sf)
```

A.6.5 Abrupt method invocation completion

Bicolano specifies abrupt method invocation completion according to the JVM specification.

When a thrown exception is not caught within the current method, the current frame is forgotten and the next frame is popped from the call stack, which means that the current method is exited.

This is described by the following semantics step:

```
| exception_uncaught : ∀ h m pc loc l m' pc' s' l' sf bm,
  METHOD.body m = Some bm →
  (∀ pc'', ¬ lookup_handlers p
    (BYTECODEMETHOD.exceptionHandlers bm) h pc loc pc'') →
  step p (StE h (FrE m pc loc l) ((Fr m' pc' s' l')::sf))
         (StE h (FrE m' pc' loc l') sf)
```

A.6.6 Additional information

The current version of Bicolano does not keep any additional information in frames.

A.7 Representation of objects

In Bicolano, there is no explicit type representing JVM objects. All operations on objects are done on `HEAP.t` and `Location` types, as described in A.5.3.

A.8 Floating-point arithmetic

Differences with the JVM specification Bicolano does not support floating-point arithmetic.

A.9 Specially named initialisation methods

Bicolano does not provide any way to get the actual name of a method as a sequence of characters. It is possible, however, to check if a name describes an instance initialisation method.

Relevant operations

- `isConstructorName : ShortMethodName →Prop`
  Holds iff the given name is a constructor name.

Differences with the JVM specification Bicolano does not support class initialisation methods (named `<clinit>` in class files).

A.10 Exceptions

Exception handling in Bicolano is consistent with the JVM specification.

A program is always in one of the two following states:

- normal state,
- exceptional state.

When an exception is thrown, the program goes into an exceptional state. There are only two possible semantic steps starting from an exceptional state:

- The state `exception_uncaught` — when the exception is not caught in the current method and the current method is not the main program method. The top frame is popped from the frame stack, and the state at the end of this step is the same, as if the exception was thrown by the method invocation instruction in the invoking method.
- The state `exception_caught` — when the exception is caught in the current method. The program goes into a normal state again, but the operand stack contains only one element: a reference to the exception object, and the program counter points to the first instruction of the exception handler.

Type definitions

```
Parameter ExceptionHandler : Set.
```

Relevant operations

- `exceptionHandlers : BytecodeMethod →list ExceptionHandler`
  Returns the list of exception handlers defined for the given method.
- `catchType : ExceptionHandler →option ClassName`
  Returns the name of the class which is the type of the given exception handler (i.e. the type used in the `catch` clause).
- `isPCinRange : ExceptionHandler →PC →bool`
  Checks whether the given program address is in the range of the given exception handler (i.e. whether an exception thrown in that address may be caught by the exception handler).
- `handler : ExceptionHandler →PC`
  Returns the address of the first instruction of an exception handler.

Differences with the JVM specification The Bicolano semantics does not specify or handle any "asynchronous" exceptions. An "asynchronous" exception is an exception that is thrown unexpectedly during execution of an instruction due to state inconsistency, JVM errors or operating system conditions (see also section 2.1).

Thus, in Bicolano, a program only throws an exception under the conditions explicitly described in the JVM specification, chapter 6.

A.11 Instruction set summary

Types and the Java Virtual Machine

Bicolano uses the Coq inductive type `Instruction` to represent JVM instructions. In many cases one instruction in Bicolano represents several JVM instructions. In such a case the Bicolano instruction has additional parameters specifying a type or a value. For instance in the JVM there are separate load instructions for different argument types. Additionally, there are some extra instructions for frequently used array indexes:

```
iload, iload_0, iload_1, iload_2, iload_3,
lload, lload_0, lload_1, lload_2, lload_3,
fload, fload_0, fload_1, fload_2, fload_3,
dload, dload_0, dload_1, dload_2, dload_3,
aload, aload_0, aload_1, aload_2, aload_3
```

In Bicolano this is modelled by the following type- and value-parametrised instruction:

```
Vload (k:ValKind) (x:Var)
```

However, it should be noted that, as mentioned above, the types long, float and double are not supported in Bicolano.

Load and store instructions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
iload, iload_¡n¿, aload, aload_¡n¿	`Vload`
istore, istore_¡n¿, astore, astore_¡n¿	`Vstore`
bipush, sipush, iconst_¡i¿, aconst_null, ldc, ldc_w	`Const`

The following JVM instructions are not supported in Bicolano:

JVM opcodes	Reason
lload, lload_¡n¿, lstore, lstore_¡n¿, lconst_¡l¿	long not supported
fload, fload_¡n¿, fstore, fstore_¡n¿, fconst_¡f¿	float not supported
dload, dload_¡n¿, dstore, dstore_¡n¿, dconst_¡d¿	double not supported
ldc2_w	double and long not supported
wide	wide not supported

Arithmetic instructions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
iadd, isub, imul, idiv, irem, ineg, ishl, ishr, iushr, ior, iand, ixor	`Ibinop`
iinc	`Iinc`

The following JVM instructions are not supported in Bicolano:

JVM opcodes	Reason
ladd, lsub, lmul, ldiv, lrem, lneg, lshl, lshr, lushr, lor, land, lxor, lcmp	long not supported
fadd, fsub, fmul, fdiv, frem, fneg, fcmpg, fcmpl	float not supported
dadd, dsub, dmul, ddiv, drem, dneg, dcmpg, dcmpl	double not supported

Type conversion instructions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
i2b	`I2b`
i2s	`I2s`

The following JVM instructions are not supported in Bicolano:

JVM opcodes	Reason
i2c	char not supported
i2l, l2i	long not supported
i2f, f2i	float not supported
i2d, d2i	double not supported
l2f, l2d, f2d, f2l, d2l, d2f	long, float and double not supported

Object creation and manipulation

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
new	`New`
getfield	`Getfield`
putfield	`Putfield`
getstatic	`Getstatic`
putstatic	`Putstatic`
baload, caload, saload, iaload, aaload	`Vaload`
bastore, castore, sastore, iastore, aastore	`Vastore`
arraylength	`Arraylength`
instanceof	`Instanceof`
checkcast	`Checkcast`

The following JVM instructions are not supported in Bicolano:

JVM opcodes	Reason
multianewarray	
laload, lastore	long not supported
faload, fastore	float not supported
daload, dastore	double not supported
l2f, l2d, f2d, f2l, d2l, d2f	long, float and double not supported

Operand stack management instructions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
pop	`Pop`
pop2	`Pop2`
dup	`Dup`
dup2	`Dup2`
dup_x1	`Dup_x1`
dup2_x1	`Dup2_x1`
dup_x2	`Dup_x2`
dup2_x2	`Dup2_x2`
swap	`Swap`

Control transfer instructions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
ifeq, iflt, ifle, ifne, ifgt, ifge	`If0`
ifnull, ifnonnull	`Ifnull`
if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge	`If_icmp`
if_acmpeq, if_acmpne	`If_acmp`
tableswitch	`Tableswitch`
lookupswitch	`Lookupswitch`
goto, goto_w	`Goto`
lookupswitch	`Lookupswitch`

The following JVM instructions are not supported in Bicolano:

JVM opcodes	Reason
jsr, jsr_w, ret	subroutines not supported

Method invocation and return instructions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
invokevirtual	`Invokevirtual`
invokeinterface	`Invokeinterface`
invokespecial	`Invokespecial`
invokestatic	`Invokestatic`

Throwing exceptions

The following JVM instructions are supported in Bicolano:

JVM opcodes	Instruction in Bicolano
athrow	`Athrow`

Implementing finally

Subroutines, required to implement `finally`, are not supported in Bicolano.

Synchronization

Multi-threading is currently not supported in Bicolano.

A.12 Class libraries

Currently, the Bicolano semantics does not include descriptions for classes that require special support from the JVM. However, the semantics has a provision for such cases: method specifications have an attribute `isNative` which allows to mark a method as one which should be described externally (i.e. not as a sequence of instructions).

A.13 Public design, private implementation

In the development of Bicolano in Coq, we tried to follow Sun's specification as close as possible. In some cases the specification explicitly allowed several different, but equivalent, ways of seeing certain design issues; in these cases we took the freedom to choose the approach which seemed most comfortable for us.

Appendix B

Bicolano translation of the running example

This appendix presents the full representation of the running example program (Figure 1.4) in Coq.

B.1 The main module

First, we present the module which specifies the complete program (notice that Bicolano defines semantics for complete programs only).

```
Require Import ImplemProgramWithList.

Import P.

Module TheProgram.

  Load "java_lang_Object.v".

  Load "java_lang_Throwable.v".

  Load "java_lang_Exception.v".

  Load "Bill.v".

  Definition AllClasses : list Class := java_lang_Object.class ::
               java_lang_Throwable.class :: java_lang_Exception.class ::
               Bill.class :: nil.

  Definition AllInterfaces : list Interface := nil.

  Definition program : Program := PROG.Build_t
    AllClasses
    AllInterfaces
End TheProgram.
```

This file loads the representation of three standard library classes that are required to properly define the `Bill` class (i.e. `java.lang.Object`, `java.lang.Throwable` and `java.lang.Exception`) and then the representation of the class itself.

B.2 Representation of standard library classes

```
(* ============================================================ *)
  Module java_lang_Object.

    Definition className : ClassName := (javaLang, object).

    Definition _init_Signature : MethodSignature := METHODSIGNATURE.Build_t
      (className, 10%positive)
      nil
      None
    .

    Definition _init_Instructions : list (PC*Instruction) :=
      (0%N, Return)::
      nil
    .

    Definition _init_Body : BytecodeMethod := BYTECODEMETHOD.Build_t
      _init_Instructions
      nil
      1
      0
    .

    Definition _init_Method : Method := METHOD.Build_t
      _init_Signature
      (Some _init_Body)
      false
      false
      false
      Public
    .

    Definition class : Class := CLASS.Build_t
      className
      None (* (Some java_lang_Object.className) *)
      nil
      nil
      (_init_Method::nil)
      false
      true
      false
    .

  End java_lang_Object.

(* ======================================================== *)

  Module java_lang_Throwable.

    Definition className : ClassName := (javaLang, throwable).

    Definition _init_Signature : MethodSignature := METHODSIGNATURE.Build_t
      (className, 10%positive)
      nil
      None
    .

    Definition _init_Instructions : list (PC*Instruction) :=
      (0%N, Vload Aval 0%N)::
      (1%N, Invokespecial java_lang_Object._init_Signature)::
      (14%N, Return)::
      nil
    .

    Definition _init_Body : BytecodeMethod := BYTECODEMETHOD.Build_t
      _init_Instructions
      nil
      1
      2
    .

    Definition _init_Method : Method := METHOD.Build_t
```

```
      _init_Signature
      (Some _init_Body)
      false
      false
      false
      Public
    .

    Definition class : Class := CLASS.Build_t
      className
      (Some java_lang_Object.className)
      nil
      nil
      (_init_Method::nil)
      false
      true
      false
    .

  End java_lang_Throwable.

(* ==================================================================== *)

  Module java_lang_Exception.

    Definition className : ClassName := (javaLang, 9%positive).

    Definition _init_Signature : MethodSignature := METHODSIGNATURE.Build_t
      (className, 10%positive)
      nil
      None
    .

    Definition _init_Instructions : list (PC*Instruction) :=
      (0%N, Vload Aval 0%N)::
      (1%N, Invokespecial java_lang_Throwable._init_Signature)::
      (4%N, Return)::
      nil
    .

        Definition _init_Body : BytecodeMethod := BYTECODEMETHOD.Build_t
      _init_Instructions
      nil
      1
      1
    .

    Definition _init_Method : Method := METHOD.Build_t
      _init_Signature
      (Some _init_Body)
      false
      false
      false
      Public
    .


    Definition class : Class := CLASS.Build_t
      className
      (Some java_lang_Throwable.className)
      nil
      nil
      (_init_Method::nil)
      false
      true
      false
    .

  End java_lang_Exception.
```

B.3 Representation of class !Bill!

```
Module Bill.

  Definition className : ClassName := (EmptyPackageName, 11%positive).

  Definition sumFieldSignature : FieldSignature := FIELDSIGNATURE.Build_t
    (className, 100%positive)
    (PrimitiveType INT)
  .

  Definition sumField : Field := FIELD.Build_t
    sumFieldSignature
    false
    false
    Package
    FIELD.UNDEF
  .

  Definition _init_Signature : MethodSignature := METHODSIGNATURE.Build_t
    (className, 10%positive)
    nil
    None
  .

  Definition roundCostSignature : MethodSignature := METHODSIGNATURE.Build_t
    (className, 11%positive)
    ((PrimitiveType INT)::nil)
    (Some (PrimitiveType INT))
  .

  Definition produceBillSignature : MethodSignature := METHODSIGNATURE.Build_t
    (className, 12%positive)
    ((PrimitiveType INT)::nil)
    (Some (PrimitiveType BOOLEAN))
  .

  Definition _init_Instructions : list (PC*Instruction) :=
    (0%N, Vload Aval 0%N)::
    (1%N, Invokespecial java_lang_Object._init_Signature)::
    (4%N, Return)::
    nil
  .

  Definition _init_Body : BytecodeMethod := BYTECODEMETHOD.Build_t
    _init_Instructions
    nil
    1
    1
  .

  Definition _init_Method : Method := METHOD.Build_t
    _init_Signature
    (Some _init_Body)
    false
    false
    false
    Package
  .

  Definition roundCostMethod : Method := METHOD.Build_t
    roundCostSignature
    None
    false
    false
    false
    Package
  .

  Definition produceBillInstructions : list (PC*Instruction) :=
    (0%N, Const INT 1%Z)::
    (1%N, Vstore Ival 2%N)::
    (2%N, Goto 20%Z)::
    (5%N, Vload Aval 0%N)::
    (6%N, Vload Aval 0%N)::
    (7%N, Getfield Bill.sumFieldSignature)::
    (10%N, Vload Aval 0%N)::
    (11%N, Vload Ival 2%N)::
    (12%N, Invokevirtual Bill.roundCostSignature)::
    (15%N, Ibinop AddInt)::
    (16%N, Putfield Bill.sumFieldSignature)::
    (19%N, Iinc 2%N 1%Z)::
    (22%N, Vload Ival 2%N)::
    (23%N, Vload Ival 1%N)::
    (24%N, If_icmp LeInt (-19)%Z)::
    (27%N, Const INT 1%Z)::
    (28%N, Vreturn Ival)::
    (29%N, Vstore Aval 3%N)::
    (30%N, Const INT 0%Z)::
    (31%N, Vreturn Ival)::
    nil
  .

  Definition produceBillHandlers : list ExceptionHandler :=
    (EXCEPTIONHANDLER.Build_t
           (Some java_lang_Exception.className) 0%N 29%N 29%N)::
    nil
  .

  Definition produceBillBody : BytecodeMethod := BYTECODEMETHOD.Build_t
    produceBillInstructions
    produceBillHandlers
    4
    4
  .

  Definition produceBillMethod : Method := METHOD.Build_t
    produceBillSignature
    (Some produceBillBody)
    false
    false
    false
    Public
  .

  Definition class : Class := CLASS.Build_t
    className
    (Some java_lang_Object.className)
    nil
```

```
      (sumField::nil)
      (_init_Method::roundCostMethod::produceBillMethod::nil)
      false
      false
      true
    .

  End Bill.
```

Appendix C

Grammar for BML predicates and specification expressions

As in the JML Reference Manual [34], we use an extended Backus-Nauer Form (BNF) grammar to describe the syntax of JML. The extensions are as follows [35].

- Nonterminal symbols are written as follows: nonterminal.
- Terminal symbols are written as follows: `terminal`.
- The terminal symbol `#` is marked as `\#`.
- Square brackets ([and]) surround optional text. Notice that '`[`' and '`]`' are terminal symbols.
- The notation ... means that the preceding nonterminal or group of optional text can be repeated zero (0) or more times.

```
predicate ::= spec-expression
spec-expression-list ::= spec-expression
                         [ , spec-expression ] ...
spec-expression ::= expression
expression-list ::= expression [ , expression ] ...
expression ::= conditional-expr
conditional-expr ::= equivalence-expr
                   [ ? conditional-expr : conditional-expr ]
equivalence-expr ::= implies-expr
                    [ equivalence-op implies-expr ] ...
equivalence-op ::= <==> — <=!=>
implies-expr ::= logical-or-expr
             [ ==> implies-non-backward-expr ]
        — logical-or-expr <== logical-or-expr
             [ <== logical-or-expr ] ...
implies-non-backward-expr ::= logical-or-expr
             [ ==> implies-non-backward-expr ]
logical-or-expr ::= logical-and-expr [ '||' logical-and-expr ] ...
logical-and-expr ::= inclusive-or-expr [ && inclusive-or-expr ] ...
inclusive-or-expr ::= exclusive-or-expr [ '|' exclusive-or-expr ] ...
exclusive-or-expr ::= and-expr [ ^ and-expr ] ...
and-expr ::= equality-expr [ & equality-expr ] ...
equality-expr ::= relational-expr [ == relational-expr] ...
        — relational-expr [ != relational-expr] ...
relational-expr ::= shift-expr < shift-expr
        — shift-expr > shift-expr
        — shift-expr <= shift-expr
        — shift-expr >= shift-expr
```

— shift-expr `<:` shift-expr
— shift-expr [`instanceof` type-spec]
shift-expr ::= additive-expr [shift-op additive-expr] . . .
shift-op ::= `<<` — `>>` — `>>>`
additive-expr ::= mult-expr [additive-op mult-expr] . . .
additive-op ::= `+` — `-`
mult-expr ::= unary-expr [mult-op unary-expr] . . .
mult-op ::= `*` — `/` — `%`
unary-expr ::= `(` type-spec `)` unary-expr
— `+` unary-expr
— `-` unary-expr
— unary-expr-not-plus-minus
unary-expr-not-plus-minus ::= `~` unary-expr
— `!` unary-expr
— `(` built-in-type `)` unary-expr
— `(` reference-type `)` unary-expr-not-plus-minus
— primary-expr [primary-suffix] . . .
primary-suffix ::= `(` [expression-list] `)`
— ‘`[`’ expression ‘`]`’
primary-expr ::= \#natural — `lv[` natural `]`
— constant — `super` — `true`
— `false` — `this` — `null`
— `(` expression `)`
— bml-primary
— jml-primary
built-in-type ::= `void` — `boolean` — `byte`
— `char` — `short` — `int`
— `long` — `float` — `double`
constant ::= java-literal
bml-primary ::= array-length-expression —
— opstack-counter-expression
— stack-expresion
array-length-expression ::= `length(` expression `)`
opstack-counter-expression ::= `cntr`
stack-expression ::= `st(` additive-expr `)`
jml-primary ::= result-expression
— old-expression
— fresh-expression
— nonnullelements-expression
— typeof-expression
— elemtype-expression
— type-expression
— spec-quantified-expr
result-expression ::= `\result`
old-expression ::= `\old` `(` spec-expression `)`
— `\pre` `(` spec-expression `)`
fresh-expression ::= `\fresh` `(` spec-expression-list `)`
nonnullelements-expression ::= `\nonnullelements` `(` spec-expression `)`
typeof-expression ::= `\typeof` `(` spec-expression `)`
elemtype-expression ::= `\elemtype` `(` spec-expression `)`
type-expression ::= `\type` `(` type `)`
spec-quantified-expr ::= `(` quantifier quantified-var-decls `;`
[[predicate] `;`]
spec-expression `)`
quantifier ::= `\forall` — `\exists`
quantified-var-decls ::= [bound-var-modifiers] type-spec quantified-var-declarator

[`,` quantified-var-declarator] . . .
quantified-var-declarator ::= ident [dims]
bound-var-modifiers ::= `non_null` — `nullable`
store-ref-list ::= store-ref [`,` store-ref] . . .
store-ref ::= store-ref-expression
— store-ref-keyword
store-ref-expression ::= store-ref-name [store-ref-name-suffix] . . .
store-ref-name ::= `\#` natural — `super` — `this`
store-ref-name-suffix ::= `(` store-ref-expression `)`
— '`[`' spec-array-ref-expr '`]`'
spec-array-ref-expr ::= spec-expression
— spec-expression `..` spec-expression
— `*`
store-ref-keyword ::= `\nothing` — `\everything` — `\not_specified`

Appendix D

The correctness proof for the `produceBill` method

This appendix presents the full proof script of the lemma which is required to prove the correctness of the `produceBill` method from the running example. The formulation of the lemma uses the `certifiedMethod` definition from the VC generator. In the course of the proof one can see how the subsequent verification conditions are generated by the base VC generator described in section 5.1.

```
Lemma produce_bill_correct :
    certifiedMethod anno_prog Bill.produce_billMethod
                              produce_bill_spec.
Proof with solve.
 myred ...
 destruct s0 as (l0,h0);destruct s as ((h,s),l).
 destruct H as (loc, (n, (sum0, (sum, (i, (Hpre, (Hmod, (Heq, HH))))))))...
 myred.
 assert (Rpre : round_cost_pre
            (stack2localvar (Num (I i) ::
                              Ref loc ::
                              Num (I sum) ::
                              Ref loc ::
                               s)
               (length
                  (METHODSIGNATURE.parameters
                        (snd Bill.round_costSignature)) + 1), h)).
    myred. trivial.
 case rv...
 assert (Rpost := H17 Rpre). clear Rpre H17...
 simpl in Rpost ...
 assert (RP2':= H0 _ (refl_equal _));clear H0;
     destruct RP2' as (n1,(Heq1, Hbound));inversion_mine Heq1...
 clear H13 H12.
 injection H11;clear H11;intros ...
 unfold mk_produce_bill_pre in Hpre...
 assert (Int.range 1).
   unfold Int.range;compute;split;trivial;intros Heq;inversion Heq.
 assert (Int.toZ (Int.const 1) = 1).
    rewrite Int.const_prop;trivial.
 assert (Heqi : Int.toZ (Int.add i0 (Int.const 1)) = i0 + 1).
  rewrite Int_add_spec. rewrite H12. omega.
  unfold Int.range;rewrite H12. omega.
 assert (Haux := arith_aux _ H3).
 assert (Haux2 := arith_aux2 _ _ H7).
 assert (Heq2 : Int.toZ (SemBinopInt AddInt sum n1) = sum + n1).
  simpl;apply Int_add_spec. unfold Int.range;split. omega.
  apply Zle_lt_trans with (sum0 + (i0 - 1) * i0 / 2 + i0);try omega.

 exists loc;
     exists n;
     exists sum0;
     exists (SemBinopInt AddInt sum n1);
     exists (Int.add i0 (Int.const 1));
     try rewrite Heqi;
     try rewrite Heq2;
     repeat split;
     auto;
     try rewrite Heqi;
     try omega.
 intros;rewrite Heap.get_update_old;auto.
 rewrite Heap.get_update_same;trivial.
 econstructor;eauto.
 replace (i0 + 1 - 1) with (Int.toZ i0);omega.
 assert (Rpost := H18 Rpre). clear Rpre H18...
 simpl in Rpost...
 myred ...
 intro XX;hnf;intros;hnf;clear XX;intros.
 simpl in Hpre, H8;simpl_hyps;subst;simpl_eq.
 injection H9;clear H9;intros;subst ...
 exists sum;repeat split;auto.
 assert (XX:= arith_aux _ H3); assert (XX2 := arith_aux2 _ _ H7);omega.
 intro XX;clear XX;simpl in H7;hnf;intros;hnf;intros.
 simpl in Hpre, H10;simpl_hyps;subst;simpl_eq.
 injection H11;clear H11;intros;subst ...
 exists sum;repeat split;auto.
 assert (n0 + 1 =  i). omega.
 rewrite <- H in H4.
 pattern n0 at -1;replace (Int.toZ n0) with  (n0 + 1 - 1);[trivial|ring].

 (* precondiction implies wp of first address *)

 intro XX;clear XX.
 simpl in H.
 destruct H as (loc, (n, (sum, (Heq1,XX)))).
 subst l.
 hnf.
 intros ...
 assert (1 = Int.const 1).
   rewrite Int.const_prop;trivial.
   compute;split;auto. intro H;discriminate H.
 exists loc;exists n; exists sum; exists sum; exists (Int.const 1);
 simpl;rewrite <- H;repeat split;auto;simpl;try omega.
 change (0/2) with 0;omega.
Time Qed.
```

Appendix E

Soundness proof of BoogiePL translation

This appendix shows the most interesting parts of the soundness proof of the translation from bytecode to BoogiePL presented in section 5.2. We first introduce three lemmas that are needed in the soundness proof.

Lemma E.1 (Weakest precondition of the post block) In Tr(P), the BoogiePL position `post` indicates the start of the postcondition. Therefore, its weakest precondition is exactly the postcondition:

$$
\begin{array}{l}
(\forall (P, m, T, s_0, s, heap, Normal\ result))\ . \\
\quad s = (heap, result :: \_, \_) \wedge p.MST(m) = (\_, T, \_) \Longrightarrow \\
\quad \mathsf{wp_b}(Tr(P), m, \texttt{post})(s_0, s) = T(s_0, (heap, Normal\ result)) \\
)
\end{array}
$$

Proof of Lemma E.1

The BoogiePL code at position post is:
`assert` $T(s_0, (heap, Normal\ result))$;
`return`;

By the definition of $\mathsf{wp_b}$, we get
$\mathsf{wp_b}(Tr(P), m, \texttt{post})(s_0, s)$
$=$ ⟦Definition of $\mathsf{wp_b}$; Assumption A4⟧
$T(s_0, (heap, Normal\ result)) \wedge true$

Lemma E.2 (Weakest precondition of the post_X block) In Tr(P), the BoogiePL position `post_X` indicates the start of the exceptional postcondition. Therefore, its weakest precondition is exactly the postcondition:

$$
\begin{array}{l}
(\forall (P, m, T, s_0, s, heap, Exception\ loc))\ . \\
\quad s = (heap, Exception\ loc :: \_, \_) \wedge p.MST(m) = (\_, T, \_) \Longrightarrow \\
\quad \mathsf{wp_b}(Tr(P), m, \texttt{post\_X})(s_0, s) = T(s_0, (heap, Exception\ loc)) \\
)
\end{array}
$$

Proof of Lemma E.2

The BoogiePL code at position post is:
`assert` $T(s_0, (heap, Exception\ loc))$;
`return`;

By the definition of $\mathsf{wp_b}$, we get
$\mathsf{wp_b}(Tr(P), m, \texttt{post\_X})(s_0, s)$
$=$ ⟦Definition of $\mathsf{wp_b}$; Assumption A4⟧
$T(s_0, (heap, Exception\ loc)) \wedge true$

Lemma E.3 (Operand stack height) Stack elements that are located above the height given by getStack-Height(P,m,pc) do not influence the value of $\mathsf{wp_l}$.

$$\begin{array}{l}\forall(P, m, pc, h, os, os', l).\\ \quad getStackHeight(P, m, pc) - 1 = h \wedge suffix(os, h) = suffix(os', h) \Longrightarrow\\ \quad \mathsf{wp}_\mathrm{l}(P, m, pc)(h, os, l) = \mathsf{wp}_\mathrm{l}(P, m, pc)(h, os', l)\end{array}$$

Proof for the Theorem 5.2.2

In order to prove the soundness theorem we show for any bytecode program P and method m that they meet our assumptions:

$$\begin{array}{l}\quad \mathsf{CertMethod_{BPL}}(Tr(P), m)\\ \Longrightarrow\\ \quad (\forall(s0)\ .\ \mathsf{wp}_\mathrm{b}(Tr(P), m, \mathsf{init}_{m_{bpl}})(s0, s0)) \wedge\\ \quad (\forall(p : Position)\ .\ stmt(Tr(P), m, p) = \text{"}\mathtt{havoc}\text{ s"} \Longrightarrow\\ \quad\quad (\forall(s0, s)\ .\ \mathsf{wp}_\mathrm{b}(Tr(P), m, p)(s0, s))\\ \quad)\\ \Longrightarrow [\![\text{Derivations (*) and (**)}]\!]\\ \quad (\forall(s0)\ .\ R(s0) \Longrightarrow \mathsf{wp}_\mathrm{l}(P, m, \mathsf{init}_m)(s0, s0)) \wedge\\ \quad (\forall(pc)\ .\ isBackEdgeTarget(P, m, pc) \Longrightarrow\\ \quad\quad (\forall(s0, s)\ .\ S(pc)(s0, s) \Longrightarrow \mathsf{wp}_\mathrm{i}(P, m, pc)(s0, s))\\ \quad)\\ \Longrightarrow\\ \quad \mathsf{CertMethod_{VC}}(P, m)\end{array}$$

$isBackEdgeTarget(P, m, pc)$ expresses that pc is the sink of a backward edge in m's CFG.

Derivation (*)

In this derivation, we shows that
$\mathsf{wp}_\mathrm{b}(Tr(P), m, \mathsf{init}_{m_{bpl}})(s_0, s)$ implies $R(s_0) \Longrightarrow \mathsf{wp}_\mathrm{l}(P, m, \mathsf{init}_m)(s_0, state(s_0))$.

Case 1: NOT isBackEdgeTarget(P,m,init)

We prove for any s_0,s:

$$\begin{array}{l}\quad \mathsf{wp}_\mathrm{b}(Tr(P), m, \mathsf{init}_{m_{bpl}})(s_0, s)\\ = [\![\text{Translation is}\\ \quad\quad \mathtt{old\_heap\ :=\ heap;}\\ \quad\quad \mathtt{regi\ :=\ parami;}\ \text{(for all parameters of the method}\\ \quad\quad \mathtt{assume}\ R\mathtt{;}\\ \quad\quad TrInstructions[\![p, m, \mathsf{init}_m]\!];\\ \quad\quad \text{State mapping}]\!]\\ \quad R(s_0) \Longrightarrow \mathsf{wp}_\mathrm{b}(Tr(P), m, instrpos(\mathsf{init}_m))(s_0, state(s_0))\\ \Longrightarrow [\![\text{Lemma 5.2.1; assumption of Case 1 implies } \mathsf{wp}_\mathrm{l}(\mathsf{init}_m) = \mathsf{wp}_\mathrm{i}(\mathsf{init}_m)]\!]\\ \quad R(s_0) \Longrightarrow \mathsf{wp}_\mathrm{l}(P, m, \mathsf{init}_m)(s_0, state(s_0))\end{array}$$

Case 2: isBackEdgeTarget(P,m,init_m)

We prove for any s_0,s:

$\mathsf{wp}_{\mathrm{b}}(Tr(P), m, \mathsf{init}_{m_{bpl}})(s_0, s)$
$=$ $[\![$Translation is

```
old_heap := heap;
this := param0;
regi := parami;
assume $R$;
assert $S(\mathsf{init}_m)$;
goto $\mathsf{init}_m$; ;
```

State mapping$]\!]$
$R(s_0) \Longrightarrow S(\mathsf{init}_m)(s_0, state(s_0)) \wedge$
$\mathsf{wp}_{\mathrm{b}}(Tr(P), m, position(\mathsf{init}_m))(s0, state(s_0))$
$\Longrightarrow [\![$predicate logic$]\!]$
$R(s_0) \Longrightarrow S(\mathsf{init}_m)(s_0, state(s_0))$
$\Longrightarrow [\![$Def. of $\mathsf{wp}_{\mathrm{l}}]\!]$
$R(s_0) \Longrightarrow \mathsf{wp}_{\mathrm{l}}(P, m, \mathsf{init}_m)(s_0, state(s_0))$

Derivation (**)

We prove for any pc, s_0, s:

$isBackEdgeTarget(P, m, pc) \wedge$
$(\forall (p : Position)\ .\ stmt(P, m, p) =$ "`havoc` s" $\Longrightarrow$
$(\forall (s_0, s)\ .\ \mathsf{wp}_{\mathrm{b}}(Tr(P), m, p)(s_0, s))$
$)$
$\Longrightarrow [\![position(P, m, pc)$ for p$]\!]$
$isBackEdgeTarget(P, m, pc) \wedge$
$(stmt(P, m, position(P, m, pc)) =$ "`havoc` s" $\Longrightarrow$
$(\forall (s_0, s)\ .\ \mathsf{wp}_{\mathrm{b}}(Tr(P), m, position(P, m, pc))(s_0, s))$
$)$
$\Longrightarrow [\![$Translation of pc starts with `havoc` s;$]\!]$
$(\forall (s_0, s)\ .\ \mathsf{wp}_{\mathrm{b}}(Tr(P), m, position(P, m, pc))(s_0, s))$
$\Longrightarrow [\![s_0, s$ for $s_0, s]\!]$
$\mathsf{wp}_{\mathrm{b}}(Tr(P), m, position(P, m, pc))(s_0, s)$
$\Longrightarrow [\![$Translation of the first two instructions$]\!]$
$(\forall (z)\ .\ S(pc)(s_0, z) \Longrightarrow \mathsf{wp}_{\mathrm{b}}(Tr(P), m, position(P, m, pc) + 2)(s_0, z))$
$\Longrightarrow [\![s$ for z; $instrpos(P, m, pc) = position(P, m, pc) + 2]\!]$
$S(pc)(s_0, s) \Longrightarrow \mathsf{wp}_{\mathrm{b}}(Tr(P), m, instrpos(P, m, pc))(s_0, s)$
$\Longrightarrow [\![$Lemma 5.2.1$]\!]$
$S(pc)(s_0, s) \Longrightarrow \mathsf{wp}_{\mathrm{i}}(P, m, pc)(s_0, s)$

Proof of Lemma 5.2.1

The proof runs by induction on the CFG of the bytecode method m.

Induction base

Return statements are the only instructions that form the induction base. We illustrate it by showing the case return.

Case instructionAt(P,m,pc) = return

$Translation:$
$\quad Tr(P, m, pc)$
$= [\![$Definition of TrInstructions; pc cannot be the sink of a backward edge; $p.MST(m) = (\_, T, \_)]\!]$
$\quad$ `goto post;`

$Proof:$
$\quad \mathsf{wp_b}(Tr(P), m, position(P, m, pc))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_b}]\!]$
$\quad \mathsf{wp_b}(Tr(P), m, \texttt{post})(s_0, s)$
$= [\![$Lemma E.1$]\!]$
$\quad T(s_0, (heap, Normal\ None))$
$= [\![$Definition of $\mathsf{wp_i}]\!]$
$\quad \mathsf{wp_i}(P, m, pc)(s_0, s)$

Induction step

For the induction step, we have to distinguish

(a) whether pc is the sink of a backward edge and

(b) whether pc is the source of a backward edge

because Tr is defined differently in these cases. Issue (a) is taken care of by using $instrpos(P, m, pc)$ rather than $position(P, m, pc)$. Issue (b) is addressed by the derivation (***), which we use for non-jumps.

We only show the proof for four interesting cases: one basic instruction, a control flow instruction, an exception throwing instruction and an invocation instruction.

Case instructionAt(P,m,pc) = iload n

$Translation:$
$\quad Tr(P, m, pc)$
$= [\![$Definition of TrInstructions, Assumption A2, $getStackHeight(P, m, pc) - 1 = cntr]\!]$

```
#if $isBackEdgeTarget(P, m, pc)$
  havoc $s$; assume $S(pc)$;
stack$(cntr + 1)$ := reg$(n)$
#if $isBackEdge(P, m, pc, next_m(pc))$
  assert $S(next_m(pc))$; return;
#if $isBackEdgeTarget(P, m, next_m(pc)) \wedge !isBackEdge(P, m, pc, next_m(pc))$
  assert $S(next_m(pc))$; goto $position(P, m, next_m(pc))$;
#if $!isBackEdgeTarget(P, m, next_m(pc)) \wedge isEdge(P, m, pc, next_m(pc))$
  goto $position(P, m, next_m(pc))$;
```

$Proof:$
$\quad \mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_b}$, $s = (h, os, l)$, $lenght(os) - 1 = cntr]\!]$
$\quad \mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc) + 1)[reg(n)/stack(cntr + 1)](s_0, s)$
$\Longrightarrow [\![s = (h, v :: os, l)$; state mapping $v = reg(n)]\!]$
$\quad \mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc) + 1)(s_0, (h, v :: os, l))$
$\Longrightarrow [\![$Derivation (***)$]\!]$
$\quad \mathsf{wp_l}(P, m, next_m(pc))(s_0, (h, v :: os, l))$
$\Longrightarrow [\![$Assumption A5$]\!]$
$\quad compat\_ValKind\_value(Ival, v) \Longrightarrow$
$\quad \mathsf{wp_l}(P, m, next_m(pc))(s_0, (h, v :: os, l))$
$= [\![$Definition of $\mathsf{wp_i}$, $s = (h, os, l)]\!]$
$\quad \mathsf{wp_i}(P, m, pc)(s_0, s)$

Case instructionAt(P,m,pc) = goto o

$Translation:$
$Tr(P, m, pc)$
$= [\![$Definition of TrInstructions, Assumption A2, $getStackHeight(P, m, pc) - 1 = cntr]\!]$

```
  #if $isBackEdgeTarget(P, m, pc)$
    havoc $s$; assume $S(pc)$;
  #if $isBackEdge(P, m, pc, jump(pc, o))$
    assert $S(jump(pc, o))$; return;
  #if $isBackEdgeTarget(P, m, jump(pc, o)) \wedge !isBackEdge(P, m, pc, jump(pc, o))$
    assert $S(jump(pc, o))$; goto $position(P, m, jump(pc, o))$;
  #if $!isBackEdgeTarget(P, m, jump(pc, o)$
    goto $position(P, m, jump(pc, o))$;
```

We have to distinguish between forward and backward jumps because the translation to BoogiePL is different. Case a and b would be identical if we would not insert a goto statement in case b.

$Proof:$

Case a: Backward jump:
$\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_b}]\!]$
$S(jump(pc, o))(s_0, s) \wedge true$
$\Longrightarrow [\![$Definition of $\mathsf{wp_l}$, Assumption A1$]\!]$
$\mathsf{wp_l}(P, m, jump(pc, o))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_i}]\!]$
$\mathsf{wp_i}(P, m, pc)(s_0, s)$

Case b: Forward jump, there are back-edges to target:
$\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_b}]\!]$
$S(jump(pc, o))(s_0, s) \wedge$
$\mathsf{wp_b}(Tr(P), m, position(P, m, jump(pc, o)))(s_0, s)$
$\Longrightarrow [\![$Predicate logic$]\!]$
$S(jump(pc, o))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_l}$, Assumption A1$]\!]$
$\mathsf{wp_l}(P, m, jump(pc, o))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_i}]\!]$
$\mathsf{wp_i}(P, m, pc)(s_0, s)$

Case c: Forward jump, there are no back-edges to target:
$\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_b}]\!]$
$\mathsf{wp_b}(Tr(P), m, position(jump(pc, o)))(s_0, s)$
$= [\![position(P, m, jump(pc, o)) = instrpos(P, m, jump(pc, o))]\!]$
$\mathsf{wp_b}(Tr(P), m, instrpos(P, m, jump(pc, o)))(s_0, s)$
$\Longrightarrow [\![$Induction hypothesis$]\!]$
$\mathsf{wp_i}(P, m, jump(pc, o))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_l}$, Assumption A1$]\!]$
$\mathsf{wp_l}(P, m, jump(pc, o))(s_0, s)$
$= [\![$Definition of $\mathsf{wp_i}]\!]$
$\mathsf{wp_i}(P, m, pc)(s_0, s)$

Case instructionAt(P,m,pc) = athrow

```
$Translation:$
  $Tr(P, m, pc)$
$=$ [[Definition of TrInstructions; Assumption A2; $getStackHeight(P, m, pc) - 1 = cntr$;
      $p.MST(m) = (\_, T, \_)$; $pc' = lookupHandler(m, pc, typ(stack(cntr)))$]]
  #if $isBackEdgeTarget(P, m, pc)$
    havoc $s$; assume $S(pc)$;
  assert stack($cntr$) != null;
  stack0 := stack($cntr$);
  #if $isBackEdge(P, m, pc, pc')$
    assert $S(pc')$; return;
  #if $isBackEdgeTarget(P, m, pc') \wedge !isBackEdge(P, m, pc, pc')$
    assert $S(pc')$; goto $position(P, m, pc')$;
  #if $!isBackEdgeTarget(P, m, pc')$
    goto $position(P, m, pc')$;
```

```
$Proof:$
  $\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc))(s_0, s)$
$=$ [[Definition of $\mathsf{wp_b}$]]
  $(stack(cntr) \neq null \wedge$
  $\mathsf{wp_b}(Tr(P), m, instrpos(p, m, pc) + 2))[stack(cntr)/stack0](s_0, s)$
$=$ [[Predicate Logic; State Mapping: stack(cntr) = loc; Lemma E.3]]
  $loc \neq null \wedge$
  $\mathsf{wp_b}(Tr(P), m, instrpos(p, m, pc) + 2)(s_0, s)$
$=$ [[$\mathsf{wp_b}(Tr(P), m, instrpos(p, m, pc) + 2)$ is exactly translation of goto instruction.]]
  $loc \neq null \wedge$
  $\mathsf{wp_l}(P, m, pc')(s_0, s)$
$\Longrightarrow$ [[Assumption A5]]
  $loc \neq null \wedge$
  $typ(loc) <: javaLangThrowable \Longrightarrow$
  $\mathsf{wp_l}(P, m, pc')(s_0, s)$
$=$ [[Definition of $\mathsf{wp_i}$]]
  $\mathsf{wp_i}(P, m, pc)(s_0, s)$
```

Case instructionAt(P,m,pc) = invokestatic target; target is void;

```
$Translation$ :
  $Tr(P, m, pc)$
= $[\![$Definition of TrInstructions, Assumption A2, $getStackHeight(P, m, pc) - 1 = cntr]\!]$
  #if $isBackEdgeTarget(P, m, pc)$
    havoc $s$; assume $S(pc)$;
  pre_heap := heap;
  arg0 := stack($cntr - |parameters(target)|$);
  assert $target\_R(pre\_heap, arg*)$;// P.MST(target) = (target_R,target_T,_)
  havoc heap;
  goto block_$pc$_Exception, block_$pc$_Normal :
  block_$pc$_Exception :
  havoc stack0r;
  assume alive(rval(stack0r), heap)
  assume typ(stack0) <: javaLangThrowable
  assume $target\_T((pre\_heap, arg*), (heap, Exception\ loc))$;
  $\#pc' = lookupHandler(m, pc, typ(loc))$
  #if $isBackEdge(P, m, pc, pc')$
    assert $S(pc')$; return;
  #if $isBackEdgeTarget(P, m, pc') \wedge !isBackEdge(P, m, pc, pc')$
    assert $S(pc')$; goto $position(P, m, pc')$;
  #if $!isBackEdgeTarget(P, m, pc')$
    goto $position(P, m, pc')$;
  block_$pc$_Normal :
  assume $target\_T((pre\_heap, arg*), (heap, Normal\ None))$;
  #if $isBackEdge(P, m, pc, next_m(pc))$
    assert $S(next_m(pc))$; return;
  #if $isBackEdgeTarget(P, m, next_m(pc)) \wedge !isBackEdge(P, m, pc, next_m(pc))$
    assert $S(next_m(pc))$; goto $position(P, m, next_m(pc))$;
  #if $!isBackEdgeTarget(P, m, next_m(pc)) \wedge isEdge(P, m, pc, next_m(pc))$
    goto $position(P, m, next_m(pc))$;

$Proof$ :
  $\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc))$
= $[\![$Definition of $\mathsf{wp_b}$, $s = (h, os, l)$, $getStackHeight(P, m, pc) - 1 = cntr$;$]\!]$
  $target\_R(h, arg*) \wedge$
  $(\forall (h')$ .
    $\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc) + 4)[stack(cntr - |parameters(target)|)/arg0, heap/pre\_heap]$
      $(s_0, (h', os, l)))$
  )
$\Longrightarrow$ $[\![$pre_heap and arg0 are only used within the translation of an invoke statement,
     As we always assign values to them, they don't show up in the vc's and we
     don't have to substitute them; State mapping;$]\!]$
  $target\_R(h, arg*) \wedge$
  $(\forall (h')$ .
    $\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc) + 4)(s_0, (h', os, l)))$
  )
$\Longrightarrow$
  $target\_R(h, arg*) \wedge$
  $(\forall (h')$ .
    $\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc) + 5)(s_0, (h', os, l))) \wedge$
      $\mathsf{wp_b}(Tr(P), m, instrpos(P, m, pc) + 12)(s_0, (h', os, l)))$
  )
$\Longrightarrow$ $[\![$Derivations (invoke*), (invoke**)$]\!]$
  $\mathsf{wp_i}(P, m, pc)(s_0, s)$
```

The two new blocks that are being generated in the translation of the invoke instruction are only reachable from the goto statement also generated for the same invoke instruction. So we can be sure that the two edges introduced here are no back-edges in the control flow graph, and therefore, the induction still works.

We split the proof in the two derivations below. We can show that the translation is sound for both choices of the nondeterministic goto. And as it is the case for both choices separately, it is also true for the conjunction.

Derivation (invoke*)

If we only look at the normal behavior, we can derive the following

$$
\begin{array}{l}
\quad \mathit{target\_R}(h, \mathit{arg}*) \wedge \\
\quad (\forall (h')\ . \\
\quad\quad \mathsf{wp_b}(Tr(P), m, \mathit{instrpos}(P, m, pc) + 12)(s_0, (h', os, l))) \\
\quad) \\
\Longrightarrow \\
\quad \mathit{target\_R}(h, \mathit{arg}*) \wedge \\
\quad (\forall (h')\ . \\
\quad\quad (\mathit{target\_T}((h, \mathit{arg}*), (h', \mathit{Normal\ None})) \Longrightarrow \\
\quad\quad \mathsf{wp_b}(Tr(P), m, \mathit{instrpos}(P, m, pc) + 13)(s_0, (h', os, l))) \\
\quad) \\
\Longrightarrow [\![\text{Derivation (***)}]\!] \\
\quad \mathit{target\_R}(h, \mathit{arg}*) \wedge \\
\quad (\forall (h')\ . \\
\quad\quad (\mathit{target\_T}((h, \mathit{arg}*), (h', \mathit{Normal\ None})) \Longrightarrow \\
\quad\quad \mathsf{wp_l}(P, m, \mathit{next}_m(pc))(s_0, (h', os, l))) \\
\quad) \\
\Longrightarrow [\![\mathit{getStackHeight}(P, m, pc) - 1 = \mathit{cntr} \Longrightarrow \mathit{getStackHeight}(P, m, \mathit{next}_m(pc)) - 1 = \mathit{cntr} - n; \\
\quad\quad \text{Lemma E.3}]\!] \\
\quad \mathit{target\_R}(h, \mathit{arg}*) \wedge \\
\quad (\forall (h')\ . \\
\quad\quad (\mathit{target\_T}((h, \mathit{arg}*), (h', \mathit{Normal\ None})) \Longrightarrow \\
\quad\quad \mathsf{wp_l}(P, m, \mathit{next}_m(pc))(s_0, (h', \mathit{pop}(os, n), l))) \\
\quad) \\
\Longrightarrow [\![\text{Predicate logic}]\!] \\
\quad (\forall (h')\ . \\
\quad\quad \mathit{target\_R}(h, \mathit{arg}*) \Longrightarrow \\
\quad\quad \mathit{target\_T}((h, \mathit{arg}*), (h', \mathit{Normal\ None})) \Longrightarrow \\
\quad\quad \mathsf{wp_l}(P, m, \mathit{next}_m(pc))(s_0, (h', \mathit{pop}(os, n), l)) \\
\quad) \\
= [\![\text{Definition of } \mathsf{wp_i},\ s = (h, os, l)]\!] \\
\quad \mathsf{wp_i}(P, m, pc)(s_0, s)
\end{array}
$$

Derivation (invoke**)

If we only look at the exceptional behavior, we can derive the following

$$
\begin{array}{l}
\quad target\_R(h, arg*) \wedge \\
\quad (\forall (h') \;. \\
\qquad \mathsf{wp}_{\mathrm{b}}(Tr(P), m, instrpos(P, m, pc) + 5)(s_0, (h', os, l))) \\
\quad) \\
\Longrightarrow [\![\text{os' = os without the first element}]\!] \\
\quad (target\_R(h, arg*) \wedge \\
\quad (\forall (stack0r, h') \;. \\
\qquad alive(rval(stack0r), h) \Longrightarrow \\
\qquad typ(stack0r) <: javaLangThrowable \Longrightarrow \\
\qquad (target\_T((h, arg*), (h', Exception\ stack0r)) \Longrightarrow \\
\qquad \mathsf{wp}_{\mathrm{b}}(Tr(P), m, instrpos(P, m, pc) + 10)(s_0, (h', os' :: stack0r :: \emptyset, l))) \\
\quad)) \\
\Longrightarrow [\![getStackHeight(P, m, pc') = 1; \\
\qquad \text{Lemma E.3}]\!] \\
\quad (target\_R(h, arg*) \wedge \\
\quad (\forall (stack0r, h') \;. \\
\qquad alive(rval(stack0r), h) \Longrightarrow \\
\qquad typ(stack0r) <: javaLangThrowable \Longrightarrow \\
\qquad (target\_T((h, arg*), (h', Exception\ stack0r)) \Longrightarrow \\
\qquad \mathsf{wp}_{\mathrm{b}}(Tr(P), m, instrpos(P, m, pc) + 10)(s_0, (h', stack0r :: \emptyset, l))) \\
\quad)) \\
= [\![\mathsf{wp}_{\mathrm{b}}(Tr(P), m, instrpos(p, m, pc) + 2) \text{ is exactly translation of goto instruction.}]\!] \\
\quad (target\_R(h, arg*) \wedge \\
\quad (\forall (stack0r, h') \;. \\
\qquad alive(rval(stack0r), h) \Longrightarrow \\
\qquad typ(stack0r) <: javaLangThrowable \Longrightarrow \\
\qquad (target\_T((h, arg*), (h', Exception\ stack0r)) \Longrightarrow \\
\qquad \mathsf{wp}_{\mathrm{l}}(P, m, pc')(s_0, (h', stack0r :: \emptyset, l))) \\
\quad)) \\
= [\![\text{Definition of } \mathsf{wp}_{\mathrm{i}}]\!] \\
\quad \mathsf{wp}_{\mathrm{i}}(P, m, pc)(s_0, s)
\end{array}
$$

Derivation (***) inside proof of Lemma 5.2.1

$\forall(P, m, pc, pos, s_0, s)$.
  $\mathsf{wp_b}(Tr(P), m, pos)(s_0, s) \wedge$
  pos marks a (possible empty) suffix of the translation of instructionAt(P,m,pc)$\wedge$
  $stmt(P, m, pos) =$
    #if $isBackEdge(P, m, pc, next_m(pc))$
      `assert` $S(next_m(pc))$; `return`;
    #if $isBackEdgeTarget(P, m, next_m(pc))\wedge !isBackEdge(P, m, pc, next_m(pc))$
      `assert` $S(next_m(pc))$; `goto` $position(P, m, next_m(pc))$;
    #if $!isBackEdgeTarget(P, m, next_m(pc)) \wedge isEdge(P, m, pc, next_m(pc))$
      `goto` $position(P, m, next_m(pc))$;
$\Longrightarrow$
  $\mathsf{wp_l}(P, m, next_m(pc))(s_0, s)$

Proof :
Case a: Backward jump, $isBackEdgeTarget(P, m, next_m(pc))$:
  $\mathsf{wp_b}(Tr(P), m, pos)(s_0, s)$
$=$
  $S(next_m(pc))(s_0, s)$
$\Longrightarrow$ $[\![isBackEdgeTarget(P, m, next_m(pc))$, Assumption A1$]\!]$
  $\mathsf{wp_l}(P, m, next_m(pc))(s_0, s)$

Case b: Forward jump, $isBackEdgeTarget(P, m, next_m(pc))$:
  $\mathsf{wp_b}(Tr(P), m, pos)(s_0, s)$
$=$
  $S(next_m(pc))(s_0, s) \wedge \mathsf{wp_b}(Tr(P), m, position(P, m, next_m(pc)))$
$\Longrightarrow$ $[\![isBackEdgeTarget(P, m, next_m(pc))$, Assumption A1$]\!]$
  $\mathsf{wp_l}(P, m, next_m(pc))(s_0, s)$

Case c: Forward jump, $!isBackEdgeTarget(P, m, next_m(pc))$:
  $\mathsf{wp_b}(Tr(P), m, pos)(s_0, s)$
$=$ $[\![$Assumption of Case c ($stmt(P, m, pos) =$ `goto` $position(P, m, next_m(pc))$;)$]\!]$
  $\mathsf{wp_b}(Tr(P), m, position(P, m, next_m(pc)))(s_0, s)$
$\Longrightarrow$ $[\![$Induction hypothesis$]\!]$
  $\mathsf{wp_i}(P, m, next_m(pc))(s_0, s)$
$=$ $[\![$Assumption of Case c (No local annotation)$]\!]$
  $\mathsf{wp_l}(P, m, next_m(pc))(s_0, s)$

Case d: No jump:
  $\mathsf{wp_b}(Tr(P), m, pos)(s_0, s)$
$=$ $[\![pos = position(P, m, next_m(pc))]\!]$
  $\mathsf{wp_b}(Tr(P), m, position(P, m, next_m(pc)))(s_0, s)$
$\Longrightarrow$ $[\![$Induction hypothesis$]\!]$
  $\mathsf{wp_i}(P, m, next_m(pc))(s_0, s)$
$=$ $[\![$Assumption of Case d$]\!]$
  $\mathsf{wp_l}(P, m, next_m(pc))(s_0, s)$