Automated Program Verifier for Java Bytecode

Samuel Willimann

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

3/5/2007 - 9/4/2007

Supervised by:
Hermann Lehner
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

This master thesis is based on B2BPL, a translator from BML annotated Java Bytecode to BoogiePL conceived and implemented in previous work at ETH Zurich. We are primarily interested in extending the existing translator with some more elaborate mechanisms in terms of translating method invocations and invariant checks.

Although B2BPL is doing fine in translating annotated Bytecode programs already, there is still room for improvement. We are also keen to see B2BPL integrated in a broader verification environment, where Java Bytecode can be directly translated to BoogiePL and verified by Boogie in an integrated development environment like Eclipse.

In order to achieve these goals, we roll up the existing implementation of the method call translation algorithm and introduce a mechanism to perform simple but more flexible invariant checks (with a particular focus on method calls).

The translation of method calls is improved by using the Boogie-specific *call* command while at the same time, exception handling is extended with an additional level of flexibility. Furthermore, invariant checks are refined such that only the invariants of modified objects need to be checked, and that inadmissible invariants are ruled out automatically. In addition to these modification in the translator itself, an Eclipse plugin is developed which is intended to integrate the translator seemlessly into the Eclipse IDE as well as the anticipated Mobius workspace, which is aimed at developers who want to have even more capabilities to verify their Java/Bytecode projects in Eclipse.

Contents

Chapter 1

Introduction

Software verification is a methodology to prove that a given piece of software fulfills its specified requirements, and can be divided into two separate disciplines; dynamic verification and static verification. While dynamic verification is performed during runtime and checks the behavior of a program dynamically, static verification is used to check software requirements by inspecting the actual source code. These checks are performed over a certain specification or property of the program using formal methods.

During the process of software verification, the program source code and the specification (or a compiled version of both) is usually translated into an intermediate representation such as guarded commands [5], for which so-called *verification conditions* (VCs) are computed. These verification conditions are then passed to a theorem prover, which tries to prove them. Successful proofs confirm the correctness of the program with respect to its specification, whereas failed proofs lead to useful error messages (including counterexamples) which programmers can use to correct the program. Figure 1.1 gives you a general idea of the software verification process.

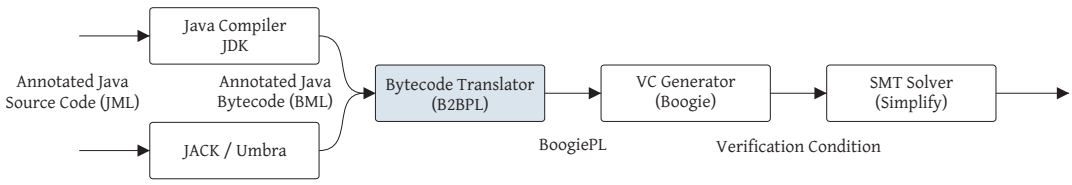

Figure 1.1: Software verification process

In this thesis, we are particularly concentrating on the translation from Bytecode to BoogiePL [3] via B2BPL (highlighted box). B2BPL takes a compiled Java program (or alternatively a set of compiled Java classes) and creates an abstract syntax tree (AST) and a control flow graph (CFG) for every method of every class type it encounters. The AST is generated via the ASM framework [7], a fast and light-weight Bytecode manipulation framework developed at INRIA (Institut National de Recherche en Informatique) in France. Both AST and CFG are then used to translate the Bytecode into BoogiePL procedures. Visitors [17] are used to perform the translation from Bytecode instructions to BoogiePL blocks and to write the resulting BoogiePL code to a file. The translation also includes BML specifications (see Section 1.1.2) such as assertions, assumptions, and loop specifications.

1.1 Preliminaries

1.1.1 The Mobius Project

Mobius (for "Mobility, Ubiquity and Security") is an integrated project launched under the *FET Global Computing Proactive Initiative* [8]. The project started on September 1, 2005 and is supported by the *European Commission* [2]. Its consortium is composed of 16 partners, one of which is the *Software Component Technology Group* at ETH Zurich.

Mobius aims at developing the technology for establishing trust and security for the next generation of global computers, using the Proof Carrying Code (PCC) paradigm. The essential features of the mobius security architecture will be:

- Innovative trust management, dispensing with centralized trust entities, and allowing individual components to gain trust by providing verifiable certificates of their innocuousness.
- Static enforcement mechanisms, sufficiently flexible to cover the wide range of security concerns arising in global computing, and sufficiently resource-aware and configurable to be applicable to the wide range of devices in global computers.
- Support for system component downloading, for compatibility with the view of a global computer as an evolving network of autonomous, heterogeneous and extensible devices.

As part of this project, a Bytecode specification language was developed [11] to define program specifications directly at Bytecode level (see Section 1.1.2). This has some advantages over using specifications only in Java source code. On the one hand, it is common in the industry (mainly for developers of mobile applications for cell phones) to write software directly in Java Bytecode in order to produce clean, optimized and, most importantly, small code. On the other hand, annotations in Bytecode can be deployed more easily, which allows clients to perform their own verifications as they see fit.

1.1.2 JML and BML

The *Java Modelling Language* (JML) [10] is a behavioral interface specification language (BISL) which allows developers to annotate their programs with specifications such as pre- and postconditions, frame conditions or invariants directly in the Java source code. Because JML is based on *JavaDoc*, adding JML annotations to a Java program is straightforward, and static verification can be performed instantly with an appropriate assertion-checking compiler (such as ESC/Java2 [6] for instance).

For the reasons explained in the previous section, a counterpart to JML has been created to provide the same specification capabilities for Java Bytecode. The *Bytecode Modeling Language* (BML) was designed to be directly written in Java Bytecode, and the language currently supports the most frequently used JML constructs. BML not only lets you write your annotated Java application directly in Bytecode (which happens to be the preferred proceeding of software developers of mobile applications), but the correctness of the program can also be verified at any time (i.e. even *after* deployment, potential customers themselves can check whether the program really fulfills its specification).

1.1.3 Spec# and Boogie

In order to transform BML into verification conditions, it is first translated into an intermediate representation, which improves the interoperability of different tools and facilitates the computation of small and (more importantly) efficient verification conditions. For our purpose, we chose BoogiePL as intermediate language, because Boogie is one of the most sophisticated VC generators available at the moment. It performs loop-invariant inference using abstract interpretation and generates verification conditions to be passed to an automatic theorem prover [1], e.g. *Simplify* [21].

Although Boogie accepts input directly written in BoogiePL (which decouples it from any concrete programming language), it was originally designed for Spec# (a superset of C# with specification features), and the existing translator only processes Spec# files. To allow compiled and annotated Java class files as input for Boogie, a translation tool has been built (see Section 1.1.4), which we will improve in the course of this thesis.

1.1.4 Previous Work on B2BPL

A translator for converting BML annotated Java Bytecode to BoogiePL was formalized in [12, 24], and a first version was implemented in [16]. Throughout this report, we will refer to the translator as *B2BPL* (short for *Bytecode to BoogiePL*). While some aspects have not yet been considered (as for example floating-point arithmetic), a notable amount of programs can already be translated to BoogiePL. The translator takes a compiled Java class together with its source file, produces an abstract syntax tree and outputs a BoogiePL file. This file can then be passed to Boogie, which will generate the resulting verification conditions.

We assume in this thesis that the reader is familiar with the formalization of B2BPL [24], especially the heap model, which is rather different from the one used by Spec#.

A lot of research in the area of software verification and static program checking has been done by Robert DeLine and K. Rustan M. Leino at Microsoft Research [22]. Although some of their ideas will also be discussed briefly in this report, the task of translating BML annotated Bytecode to BoogiePL is different from the translation of Spec# to BoogiePL in such a way that we are going to address somewhat different issues here for the most part.

1.2 Notation

Although this thesis addresses the translation from BML (annotated Java Bytecode) to BoogiePL, we will give most of our examples in JML (annotated Java source code), simply for the reason of legibility. However, keep in mind that we are always translating directly from Java Bytecode.

Nonetheless, we are dealing with a couple of different languages and formalizations and therefore use a selection of distinct notations to make those code passages and formulas easier to distinguish.

1.2.1 Source Code

Source code fragments written in Java, Java Bytecode, and BoogiePL respectively are shown in typewriter font in order to separate code examples more distinctly from the remaining formal extracts, and keywords are highlighted according to the respective language.

```
1 public int twice(int x) {
2   return x + x;
3 }
```

1.2.2 Formal Transformations

Formalized transformation algorithms are presented similarly to those in previous work. Abstract functions and variable names are written in italics, whereas branch conditions and loops in the formalization itself are colored in a light gray tone. However, BoogiePL code does not appear directly in the formalizations. Instead, we use abstract functions, and we will give a couple of comprehensive examples after each formalization to illustrate how the individual transformations work.

$$value := heap.get(object, field);$$

Chapter 2

Method Calls

The current version of B2BPL (as formalized in [24] and implemented in [16]) handles method invocations by directly performing pre- and postcondition checks through assumptions and assertions. However, BoogiePL offers a built-in statement for method calls. Instead of handling the contract of every method manually, the *call* statement takes care of all necessary steps automatically. One of the advantages of using this particular syntactic sugar is that the contract of a method has to be declared only once in the appropriate method declaration. It is not necessary to perform these checks explicitely every time a method is called.

Consider the following method declaration in BoogiePL:

```
procedure C.m ( args ) returns ( results ) {
  requires P;
  modifies mm;
  ensures Q;
}
```

where P is the precondition of method $C.m$, Q its postcondition, and $args$, $results$ and mm are lists of variables (Note that BoogiePL has no notion of lists. It is only a shorthand used in this thesis to avoid unnecessarily cluttered code fragments). If we invoke this method with the predefined *call* statement

```
call results := C.m ( args );
```

it will be desugared and interpreted by Boogie as

```
var temp_args, temp_results; // fresh variables in the local scope
temp_args := args;           // save old version of method arguments
assert P;                    // assert precondition of C.m
havoc mm;                    // destroy all knowledge about global variables in mm
assume Q;                    // assume postcondition of C.m
results := temp_results;     // assign return values
```

Note again that all variables mentioned in the example above are actually lists of variables. For simplicity, we write $x := y$ when we actually assign each value of x an according value of y (i.e. $x_1 := y_1$, $x_2 := y_2$ etc.).

Unfortunately, the *call* statement has no means of exception handling whatsoever. In order to translate method calls which may throw exceptions correctly, we need to provide a mechanism for recovering from exceptions thrown by the method and delegating the control flow to an appropriate exception handler manually.

In this chapter, we develop a formal specification for translating method calls from BML to BoogiePL. To illustrate the improvements of the translation compared to the existing solution, we

will fall back on the example used in [24]. The example is based on a simple *Account* class which contains several methods and will be used throughout this thesis.

Listing 2.1: Sample class used in this thesis (implementation omitted)

```
1  public class Account {
2
3    int balance;
4
5    public C(int initial) {
6      this.balance = initial;
7    }
8
9    public int clear() { /* ... */ }
10
11   public void deposit(int amount) { /* ... */ }
12
13   public void withdraw(int amount)
14     throws InsufficientFundsException { /* ... */ }
15
16   public static void transfer(Account src, Account dest, int amount)
17     throws TransferFailedException { /* ... */ }
18 }
```

2.1 Preliminaries

2.1.1 Method Specifications

Before we go into detail, let us take a look at method specifications in general. Method specifications are used to describe a method's behavior in terms of requirements (preconditions) and effects (postconditions), expressed as predicates over abstract states. The methodology of defining precise and checkable interface specifications for software components based upon the theory of abstract data types is also known as *Design by Contract* [18]. Essentially, every method can specify requirements which have to be met before it can be called. If the caller satisfies these requirements, the method guarantees certain consequences in return (hence the term *contract*).

Preconditions

Preconditions are predicates over fields of the current class and all its superclasses[1] on the one hand, and over method arguments on the other hand. Preconditions must hold before the corresponding method is called. If the caller of a method cannot satisfy the method's preconditions, a precondition violation is triggered, and the method call is invalid. This also implies that the called method is not bound to fulfill its part of the contract.

We usually denote preconditions as P. In BoogiePL, a precondition is encased in a *requires* statement right after a procedure declaration:

```
requires P;
```

Postconditions

While preconditions express premises which the caller must satisfy prior to a method call, postconditions express conditions which the callee must guarantee at the end of its invocation. Unlike

[1] Here and throughout, the term superclass comprises all parent classes as well as the base class itself.

preconditions, postconditions are not only predicates over fields of the current class, all its superclasses and method arguments, but also over their corresponding state prior to the method call (usually refered to as the *old* state). Hence, postconditions are also called *two-state predicates*. If a method does not fulfill its postcondition, a postcondition violation will be triggered. This implies that the method implementation is faulty, and that the method body needs to be revised.

Postconditions are commonly denoted as Q. However, the relation between pre- and postcondition has to be explicitely specified. If we want to guarantee that Q holds after a method execution only if the precondition P was valid at the beginning, we have to express our postcondition as $P \Rightarrow Q$ in BoogiePL:

```
ensures P ==> Q;
```

Note: This conception of postconditions is somewhat ambiguous. Actually, Q should be sufficient as postcondition, because if P does not hold, we would not be allowed to call the method in the first place. Since P can be assumed to hold when a method is called ($P = true$), $P \Rightarrow Q$ can be reduced to Q. In general, we will write postconditions in their expanded form, i.e. $P \Rightarrow Q$ (especially in formalizations) to point out that P is an implicit part of it. However, we will omit P in some of our examples where the actual pre- and postcondition is irrelevant.

Exceptional Postconditions

If a method terminates by throwing an exception, the normal postcondition might not hold. To avoid a totally unpredictable program state in this particular case (that is, to prevent a postcondition violation), special postconditions can be declared along with any exception type that is thrown by the method, which enables programmers to reason about exceptional method termination. These conditions can be formalized as $(exception = MyException) \Rightarrow (Q_{MyException})$ and state that if the exception that was thrown is of type $MyException$, the postcondition $Q_{MyException}$ will hold. In order to make postcondition declarations more legible, every exceptional postcondition is defined in a separate `ensures` clause:

```
ensures P && (exception == MyException) ==> Q_MyException;
```

Method Frame

If a method is not side-effect-free (that is, if it modifies local class fields), it is imperative that all the fields that are modified by this method are specified as part of the contract. A method $C.m$ is not allowed to change fields which are not explicitely defined as being *modified* in $C.m$. Variables declared within the scope of the method can always be modified without special declaration. Both JML and BML provide the $modifies$ keyword to define modified variables:

```
modifies C.a;
```

2.1.2 Checked and Unchecked Exceptions

Checked exceptions force the programmer to define the occurrence of exceptions explicitely. An exception is called *checked* if a method has to explicitely define that it potentially throws an exception of that particular type. On the other hand, exceptions which can be thrown without an explicit declaration are called *unchecked*. Unchecked exceptions have not to be specified mainly because they can occur at virtually any place in the program (like the $OutOfMemoryException$ in Java) or because they signal unrecoverable errors (like $DivisionByZeroException$ or $NullPointerException$ in Java). However, Java and C#[2] handle checked and unchecked exceptions quite differently.

[2] We include C# in our considerations because Boogie was originally intended to verify source code written in C#/Spec#. Other translators whose target language is BoogiePL presumably have to deal with C#/Spec# in one way or another.

In Java, all exceptions inherit from the class *Throwable*, which is a checked exception. That is, if a given exception is a subtype of *Throwable*, it will usually be a checked exception. Only a few subtypes of *Throwable* (such as *Error* and *RuntimeExceptions* and their subtypes) are unchecked exceptions.

The methodology behind exception handling in C# (or Spec# for that matter) on the other hand works the other way around. The base class for any exception is *Exception*, which is an unchecked exception. This means that all derived exception types are usually unchecked. There are no checked exceptions in Spec# by default. However, if checked exceptions are needed for specific purposes, they can implement the *ICheckedException* interface.

2.1.3 Subtyping and Subclassing

Subclassing is one of the most important and most powerful mechanisms in object-oriented programming. Subclasses inherit properties from their superclasses, but can specify additional properties which makes them more specific than their base classes. Subtyping on the other hand expresses the fact that whenever an object of type T is required, the applied object can be either of type T or a subtype thereof.

Of course, a subclass inherits its specifications from the base class as well, which includes class invariants, method pre- and postconditions, and frame conditions. However, a subclass can refine these conditions to adapt them to its particular needs. In order not to violate the rules of subtyping, there are certain restrictions for those refinements:

- A class invariant for example can only be made weaker, i.e. existing invariants can be redefined to be less restrictive. If a class C defines the invariant $x > 3$ for instance, a subclass C' of C can only state that $x > 0$, but *not* $x > 5$. This rule can be expressed as:

$$inv(C) \Rightarrow inv(C')$$

- The precondition of an inherited method $C'.m$ must be weaker than the precondition of the corresponding base class method $C.m$:

$$pre(C.m) \Rightarrow pre(C'.m)$$

- The postcondition of an inherited method $C'.m$ must be stronger than the postcondition of the corresponding base class method $C.m$. This leads to the following relation:

$$post(C'.m) \Rightarrow post(C.m)$$

Let C be a class and C' a direct subclass of C. Further, let P and P' be the explicitely defined preconditions (or *true* if none are specified) and Q and Q' the explicitely defined postconditions (or *true* if none are specified) of C and C' respectively.

During the translation, we need to take into account that P is part of C''s precondition and Q part of C''s postcondition as well. According to the rules specified above, we use the following contract conditions for the methods $C.m$ and $C'.m$:

$$\begin{aligned} inv(C) &:= I \\ pre(C.m) &:= P \\ post(C.m) &:= P \Rightarrow Q \\ inv(C') &:= I \vee I' \\ pre(C'.m) &:= P \vee P' \\ post(C'.m) &:= (P \vee P') \Rightarrow (Q \wedge Q') \end{aligned}$$

2.2 Procedure Declarations and Method Calls in BoogiePL

In BoogiePL, method declarations consist of two separate constructs; its signature declaration (which also includes the contract specifications for that particular method) and an optional implementation [3]. Methods can have an arbitrary number of input and output parameters. However, the signature of a concrete implementation must match the signature of its corresponding specification declaration. *C.m* can be replaced by an arbitrary method name.

```
1  procedure C.m ( in_1: t_in_1, ..., in_n: t_in_n )
2    returns ( out_1: t_out_1, ..., out_n: t_out_n );
3    requires P;
4    modifies M;
5    ensures Q;
6
7  procedure C.m ( in_1: t_in_1, ..., in_n: t_in_n )
8    returns ( out_1: t_out_1, ..., out_n: t_out_n ) {
9    var x: t_x;
10 init:
11   // do something
12   return;
13 }
```

The syntax of method calls is straightforward and resembles method invocations of common programming languages. Again, the number and type of the method arguments and the return values must match the exact signature of the called method.

```
call result_1, ..., result_n := C.m ( arg_1, ..., arg_n );
```

2.3 Formal Translation

This section addresses the formal translation of method calls and the corresponding method specifications involved. A detailed description of the actual implementation in BoogiePL can be found in Section 2.4.

Basically, our method call consists of $n(+1)$ method parameters (n for the actual arguments and 1 for referencing the *this* object, if the method is non-static) as well as three return values:

$$C.m\ (\ this, arg_0, arg_1, \ldots, arg_n\ )\ returns\ (\ state, result, exception\ )$$

This is, every method invocation yields a triple of special values. *state* indicates whether the method was terminated properly (i.e. without exception) or whether an exception was thrown. In the former case, *result* will contain the corresponding return value, and in the latter the thrown exception will be returned in the variable *exception*. *result* may be omitted if the method has no return value (*void*).

Since a method call can either result in a return value or an exception, but not both at the same time, it is tempting to use a single return value which can potentially contain both normal and exceptional "return values". However, we decided to separate both types of method terminations. One reason was that a method's regular return value can be of an arbitrary type (however, only *ref* and *int* are currently available in BoogiePL), whereas the thrown exception is always of type *ref* (and in fact a subtype of *Exception*). Furthermore, normal and exceptional method termination are not the same thing from a conceptional point of view, and we also need more flexibility in terms of exception handling later on.

2.3.1 The callee's commitments

Let $C.m$ be an arbitrary method with precondition P, postcondition Q, and exceptional postconditions $Q_{Ex_0}, \ldots, Q_{Ex_n}$ (all of which are explicitely specified as BML annotations in the Bytecode). In its most general form, $C.m$ has to guarantee that, if P holds at the beginning of its execution, either Q holds after a normal method termination, or Q_{Ex_i} holds after an exceptional method termination where an exception of type Ex_i was thrown.

This sounds fairly simple, but unfortunately, the predicates mentioned above are not the only components of the contract. Behind the scenes, P and Q have to be extended to perform additional type checks, and the heap needs to be managed accordingly.

Precondition

In particular, P is required to check the types of the method arguments and whether they are alive on the heap. In case of a non-static method, the *this* object must not be *null*. In addition, the object invariant I is required to hold as well. This yields the new precondition for method $C.m$ (see Listing 2.2).

Listing 2.2: Formal translation of preconditions

$TrPre(C.m) :=$
    $require(P)$;
    if C.m is not a constructor {
        $require(I_{pre}(C.m))$;
    }
    if C.m is not static {
        $require(this \neq null)$;
    }
    for all input parameters p_i do {
        $require(heap.alive(p_i))$;
        $require(type(p_i) = T_{p_i})$;
    }

Note that we will omit all details of I and postpone in-depth explanations about invariant checks to Chapter 3.

Postcondition

Extending the postcondition is a bit trickier, but follows the same rules. First of all, we define the postcondition for the normal method termination ($state = t_n$). Regardless of any other property, we always guarantee Q and the frame condition FC (see paragraph Frame Condition below) to hold. If method $C.m$ has a return value, we also need to check the type of the return value. Constructors on the other hand use the return value to return their initialized owner object.

In case of an exceptional method termination ($state = t_{ex}$), we guarantee Q_{Ex_i} and make sure that *exception* contains a valid exception object.

Taking all these considerations into account, we propose the postcondition for method $C.m$ (see Listing 2.3).

Listing 2.3: Formal translation of postconditions

$TrPost(C.m) :=$
  if C.m has a return value {
      $ensure(P \Rightarrow state = t_n \Rightarrow (Q \wedge FC(C.m) \wedge heap.alive(result) \wedge type(result) = T_{result}))$;
  } else if C.m is a constructor {
      $ensure(P \Rightarrow state = t_n \Rightarrow (Q \wedge FC(C.m) \wedge heap.alive(result) \wedge type(result) = T_{owner}))$;

```
} else {
    ensure(P ⇒ state = t_n ⇒ (Q ∧ FC(C.m)));
}
for each potential exception Ex_i do {
    ensure(P ⇒ state = t_ex ⇒ type(exception) = T_Ex_i ⇒ (Q_Ex_i ∧ heap.alive(exception)));
}
ensure(I_post(C.m));
```

where T_x denotes the static type of object x.

Frame Condition

FC denotes the method's *frame condition*, which ensures that only variables mentioned in the $modifies$ clause are modified, and all other variables remain unchanged on the heap. As we will explain in detail later, the heap is invalidated in the course of a method call, and all knowledge about it has to be reestablished at the end of a method call. The frame condition makes sure that after a method body is executed, the new heap is equal to the heap prior to the method execution, except that the fields that are listed in the method's $modifies$ clause may possibly have been altered.

Because our new algorithm only relies on the invariants (and not on the actual values), we first considered removing the postcondition which ensured that values on the heap remained intact over method calls by explicitely guaranteeing their equality. However, it turned out that this was not quite possible, because explicitely defined postconditions ensuring exactly this property in case of an exception relied on this transitive property. Imagine an arbitrary method $C.m$ which, in case of an exceptional termination, ensures that some variables keep their original value. This property has to be somehow propagated to the caller so the caller can rely on it in its own postcondition. Hence, every method call that takes place within $C.m$ must also ensure the equality property for all unmodified fields:

$$\begin{aligned} FC(C.m) \quad := \quad & \forall\, loc, v : v = loc.value \Rightarrow old\_heap.alive(v) \Rightarrow heap.alive(v) \wedge \\ & (\neg loc.modified \Rightarrow heap.get(loc) = old\_heap.get(loc)); \end{aligned}$$

where v denotes a value and loc its location on the heap. FC states that, if a given location was properly allocated prior to the method call, it will still be allocated after the method has terminated. Also, the value stored at the given location on the heap is equal to the corresponding value on the old heap (that is, as long as the location is not modified by $C.m$).

Note that in this version of the fame condition, invariants are not yet considered. In order to handle object invariants, we use the predicate I which is already present in the formalizations in Listings 2.2 and 2.3. We will discuss the exact formalization and implementation of object invariants in Sections 3.6 and 3.7 respectively.

Boogie's Desugaring of the Callee

If a given BoogiePL procedure with manually defined precondition P and postcondition Q is processed by Boogie, these specifications will be transformed into a pair of assertions and assumptions and replace the actual method implementation as follows:

Listing 2.4: Boogie's transformation of the invoked method (inside the callee)

```
1 init:
2   // precondition check
3   assume P;
4
5   // method implementation
6   /* ... */
```

```
7
8   // postcondition check
9   assert Q;
10  return;
```

2.3.2 The caller's commitments

By incorporating BoogiePL's *call* statement into the translation process, the actual code needed to perform a method call becomes conveniently compact. The formalization is essentially the same as our method declaration we presented at the beginning of Section 2.3:

$$TrCall(C.m) := (state, result, exception) = C.m(this, p_1, ..., p_n);$$

Boogie's Desugaring of the Caller

If a call statement to a given BoogiePL procedure with manually declared precondition P and postcondition Q is processed by Boogie, the specifications will be interpreted as the following sequence of statements:

Listing 2.5: Boogie's transformation of the method invocation (inside the caller)

```
1  // Precondition is required to hold here
2  assert P;
3
4  // call result := C.m(params); // obsolete
5
6  // Postcondition holds after method termination
7  assume Q;
```

This is, the actual method call is omitted and replaced by its contract conditions.

2.3.3 Additional Modifications

Due to the fact that method calls are an integral part of object-oriented software, introducing the *call* command meant a lot of minor changes in the existing background logic as well. It was especially challenging to adapt the behavior of the existing heap model. A method call implicitely destroys all knowledge about the current state of the heap (because all objects that were allocated during method execution must be removed from the stack), and the heap properties have to be reestablished by the method's frame condition. These modifications will be described in detail in the course of Section 2.4.

2.4 Implementation in BoogiePL

In this section, we will describe the actual implementation of our formalization from the previous section in BoogiePL by first presenting the individual components and exemplifying the translation in detail at the end of this chapter.

2.4.1 Background Theory for handling the Return State

As explained above, method calls return three distinct values—one of which is the return state (which can either be *normal* or *exceptional*). We define two constants of a new special type *ReturnState* to model these two values:

```
type ReturnState;

const $normal: ReturnState;
const $exceptional: ReturnState;
```

Depending on the value of a method's *return state*, either the default postcondition or one of the exceptional postconditions must hold. Listing 2.6 shows all required modifications of the background theory to support this notion of *normal* and *exceptional* method termination.

Listing 2.6: Background theory for handling the return state

```
function isNormalReturnState(ReturnState) returns (bool);
function isExceptionalReturnState(ReturnState) returns (bool);

axiom (forall s: ReturnState :: s != $normal <==> !isNormalReturnState(s));
axiom (forall s: ReturnState :: s != $exceptional <==> !isExceptionalReturnState(s));
```

Essentially, we define two helper functions to easily check whether a given return state is *normal* or *exceptional*, and we make sure that a variable of type *ReturnState* must be either `$normal` or `$exceptional` in order to satisfy either of the two helper functions.

Now let us take a look at the concrete implementation of a method invocation. Let *state* be a variable of type *ReturnState*. As explained in Section 2.3, *state* is the first return parameter of our translated method call, followed by a variable of type *ref* (or *int* if the method returns an integer value) and another variable of type *ref* containing a reference to the exception thrown (if applicable).

After a method invocation, the caller has to branch undeterministically to all possible return states (i.e. the *normal* method termination state on the one hand and all of the possible *exceptional* method termination states on the other hand). In case of a normal method termination, the control flow continues at the next regular line in the code (i.e. after the exception handlers, if any are present). Otherwise, the program jumps to the appropriate exception handler.

Listing 2.7: Method invocation with exception handling (conceptional example)

```
method_call:
  call state, result, exception := C.m( /* ... */ );
  goto normal_state, exception_1, exception_2; // ...

normal_state:
  assume isNormalReturnState(state);
  // result contains return value
  goto end_method_call;

exception_1:
  assume isExceptionalReturnState(state) && isInstanceOf(rval(exception), $Exception1);
  // exception contains Exception1 object
  goto exception_handler_A;

exception_2:
  assume isExceptionalReturnState(state) && isInstanceOf(rval(exception), $Exception2);
  // exception contains Exception2 object
  goto exception_handler_B;

exception_handler_A:
  // handle exception A
  goto end_method_call;

exception_handler_B:
```

```
25   // handle exception B
26   goto end_method_call;
27
28 end_method_call:
```

2.4.2 Frame Condition

The translation of a typical frame condition FC (without any *assignables*) into BoogiePL is straighforward. Keep in mind that the frame condition is part of the normal postcondition ($P \Rightarrow Q \wedge FC$):

Listing 2.8: Source code for frame condition

```
ensures (forall v: Value :: alive(v, old(heap)) ==> alive(v, heap));
ensures (forall l: Location :: alive(rval(obj(l)), old(heap)) ==>
        get(heap, l) == get(old(heap), l);
```

If we want to consider *assignables* as well, we must adapt the second clause accordingly (assuming that m is a modified location):

```
ensures (forall l: Location :: alive(rval(obj(l)), old(heap)) ==>
        l != m ==> get(heap, l) == get(old(heap), l);
```

2.4.3 Stack and Register Values

So far, the translator used special variables of the form `stack#_r` and `stack#_i` to cope with stack operations as well as `reg#_r` and `reg#_i` to temporarily store method arguments at the beginning of a method. For method calls, we introduce additional variables `rs#`, `rv#` and `ex#` to cope with the special return values. Every method call uses a fresh set of those variables. They are basically used to evaluate the return state of a method and to jump to the correct instruction afterwards. After that, they are not used any further.

If the return value of a method is used after a method call, it will first be assigned to a stack variable `stack#_r` though:

Listing 2.9: Calling a normal method or a constructor (simplified example)

```
call rs0, rv0_r, ex0 := C.m(arg0, arg1, arg2 /* ... */);
stack1_r := rv0_r;
```

However, this behavior applies only to normal methods and constructor calls. There is an exceptional case when a super-constructor is called within a constructor. In this particular case, the return value (which we then define to be the instantiated object) is always assigned to the corresponding register variable `reg#_r` which already contains the (unitialized) object. This is done because every time a class field is either read or updated, the reference to the current *this* object is loaded from register 0, where the initialized object is supposed to be located. By storing the return value of a super-constructor directly to the correct register variable, we make sure that subsequent accesses to fields are always performed on a properly initialized *this* object.

Listing 2.10: Calling a super-constructor (simplified example)

```
stack0_r := reg0_r;
call rs0, rv0_r, ex0 := $java.lang.Object..init(stack0_r);
reg0_r := rv0_r;
```

2.4.4 Constructors

The handling of constructors differs slightly from the handling of normal methods, mainly because constructors have to establish the class invariant, unlike normal methods which only have to preserve it. To do so, constructors are responsible for initializing the current class in which they are defined (which also includes instantiating and initializing all field variables).

First, let us assume that we want to create a new object of type C and therefore need to call the constructor of that particular class. Our method call is translated into BoogiePL as follows:

```
1  // A new object of type C is allocated on the heap and assigned to stack2_r:
2  havoc stack1_r;
3  assume new(heap, objectAlloc($C)) == rval(stack1_r);
4  heap := add(heap, objectAlloc($C));
5  stack2_r := stack1_r;
6
7  // The constructor of C is called, and the constructor's postcondition is assumed:
8  call rs0, rv0_r, ex0 := C..init(stack2_r);
9  assume true;
10
11 // Exception handling omitted here.
12
13 // The instantiated object of type C is stored in stack1_r for further use:
14 stack1_r := rv0_r;
```

After this block of code, the topmost variable on the stack ($stack1\_r$ in our example above) contains the initialized object, which is returned by the constructor of C on line 6.

2.4.5 Example

To clarify the general issues of the previous sections, we resume our example from the beginning of this chapter and translate a sample implementation into (readable) BML and BoogiePL consecutively.

Listing 2.11: Class *Account* in Java

```
1  public class Account {
2
3    protected int balance;
4
5    rotected int interest; // [%]
6
7    //@ invariant this.balance >= 0 && this.interest >= 1;
8
9    public Account() {
10     this.balance = 0;
11     this.interest = 1;
12   }
13
14   //@ requires initial >= 0;
15   //@ ensures this.balance == initial;
16   public Account(int initial) {
17     this.balance = initial;
18     this.interest = 1;
19   }
20
21   //@ requires amount >= 0;
22   //@ modifies this.balance;
23   //@ ensures this.balance == \old(this.balance) + amount;
```

```
24   public void deposit(int amount) {
25     this.balance += amount;
26   }
27
28   //@ requires this.balance >= amount;
29   //@ modifies this.balance;
30   //@ ensures this.balance == \old(this.balance) - amount;
31   //@ signals (InsufficientFundsException e) this.balance == \old(this.balance);
32   public void withdraw(int amount) throws InsufficientFundsException {
33     if (this.balance < amount) {
34       throw new InsufficientFundsException();
35     } else {
36       this.balance -= amount;
37     }
38   }
39
40   //@ requires source != null && target != null && amount <= source.balance;
41   //@ ensures \old(source.balance) - source.balance == amount &&
42   //           target.balance - \old(target.balance) == amount;
43   //@ signals (TransferFailedException e) source.balance == \old(source.balance) &&
44   //           target.balance == \old(target.balance);
45   public static void transfer(Account source, Account target, int amount)
46     throws TransferFailedException {
47     try {
48       source.withdraw(amount);
49     } catch (InsufficientFundsException e) {
50       throw new TransferFailedException();
51     }
52     target.deposit(amount);
53   }
54
55   //@ ensures \\result == this.balance;
56   public int getBalance() {
57     return this.balance;
58   }
59 }
```

Listing 2.12: Class *Account* in Java Bytecode

```
1  class Account {
2  public void <init>()
3  0:   aload_0
4  1:   invokespecial   java.lang.Object.<init> ()V (11)
5  4:   aload_0
6  5:   iconst_0
7  6:   putfield        Account.balance I (13)
8  9:   aload_0
9  10:  iconst_1
10 11:  putfield        Account.interest I (15)
11 14:  return
12
13 public void <init>(int initial)
14 {|   requires initial >= 0
15       ensures this.balance == initial |}
16 0:   aload_0
17 1:   invokespecial   java.lang.Object.<init> ()V (11)
18 4:   aload_0
19 5:   iload_1
```

```
6:   putfield        Account.balance I (13)
9:   aload_0
10:  iconst_1
11:  putfield        Account.interest I (15)
14:  return

public void deposit(int amount)
{|   requires amount >= 0
     modifies this.balance
     ensures this.balance == \old(this.balance) + amount |}
0:   aload_0
1:   dup
2:   getfield        Account.balance I (13)
5:   iload_1
6:   iadd
7:   putfield        Account.balance I (13)
10:  return

public void withdraw(int amount)
              throws InsufficientFundsException
{|   requires this.balance >= amount
     modifies this.balance
     ensures this.balance == \old(this.balance) - amount
     signals (InsufficientFundsException e) this.balance == \old(this.balance) |}
0:   aload_0
1:   getfield        Account.balance I (13)
4:   iload_1
5:   if_icmpge       #16
8:   new             <InsufficientFundsException> (27)
11:  dup
12:  invokespecial   InsufficientFundsException.<init> ()V (29)
15:  athrow
16:  aload_0
17:  dup
18:  getfield        Account.balance I (13)
21:  iload_1
22:  isub
23:  putfield        Account.balance I (13)
26:  return

public static void transfer(Account src, Account dest, int amount)
              throws TransferFailedException
{|   requires source != null && target != null && amount <= source.balance
     ensures \old(source.balance) - source.balance == amount &&
             target.balance - \old(target.balance) == amount
     signals (TransferFailedException e) source.balance == \old(source.balance) &&
             target.balance == \old(target.balance) |}
0:   aload_0
1:   iload_2
2:   invokevirtual   Account.withdraw (I)V (34)
5:   goto            #17
8:   astore_3
9:   new             <TransferFailedException> (32)
12:  dup
13:  invokespecial   TransferFailedException.<init> ()V (36)
16:  athrow
17:  aload_1
18:  iload_2
```

```
19:   invokevirtual   Account.deposit (I)V (37)
22:   return

public int getBalance()
{|    ensures \result == this.balance |}
0:    aload_0
1:    getfield        Account.balance I (13)
4:    ireturn

}
```

The resulting BoogiePL code of class *Account* above is relatively large and encompasses several hundreds of lines of code. Instead of giving you the full translation here, we will focus only on the translation of method *transfer*, since this is the only method where we actually call other methods.

Listing 2.13: Method *transfer* in BoogiePL

```
procedure Account.transfer.Account.Account.int
  (param0: ref, param1: ref, param2: int)
  returns (retstate: ReturnState, exception: ref);
  requires alive(rval(param0), heap) && isOfType(rval(param0), $Account);
  requires alive(rval(param1), heap) && isOfType(rval(param1), $Account);
  requires (forall o: ref, t: name :: alive(rval(o), heap) && t <: typ(rval(o)) ==>
    inv(t, o, heap));
  requires param0 != null && param1 != null &&
    param2 <= toint(get(heap, fieldLoc(param0, Account.balance)));
  modifies heap;
  ensures isNormalReturnState(retstate) ==>
    param0 != null && param1 != null &&
    param2 <= toint(get(old(heap), fieldLoc(param0, Account.balance))) ==>
      toint(get(old(heap), fieldLoc(param0, Account.balance))) -
      toint(get(heap, fieldLoc(param0, Account.balance))) == param2 &&
      toint(get(heap, fieldLoc(param1, Account.balance))) -
      toint(get(old(heap), fieldLoc(param1, Account.balance))) == param2;
  ensures isExceptionalReturnState(retstate) &&
    isOfType(rval(exception), $TransferFailedException) ==>
    (
      param0 != null &&
      param1 != null &&
      param2 <= toint(get(old(heap), fieldLoc(param0, Account.balance))) &&
      ==>
        toint(get(heap, fieldLoc(param0, Account.balance))) ==
        toint(get(old(heap), fieldLoc(param0, Account.balance))) &&
        toint(get(heap, fieldLoc(param1, Account.balance))) ==
        toint(get(old(heap), fieldLoc(param1, Account.balance)))
    ) && alive(rval(exception), heap);
  ensures true && inv(typ(rval(param0)), param0, heap) &&
    inv(typ(rval(param1)), param1, heap);

implementation Account.transfer.Account.Account.int
  (param0: ref, param1: ref, param2: int)
  returns (retstate: ReturnState, exception: ref)
{
  var reg0_r: ref;
  var reg1_r: ref;
  var reg2_i: int;
  var reg3_r: ref;
  var stack0_r: ref;
  var stack1_i: int, stack1_r: ref;
```

```
43    var rs0: ReturnState, ex0: ref;
44    var rs1: ReturnState, rv1_r: ref, ex1: ref;
45    var rs2: ReturnState, ex2: ref;
46
47  init:
48    reg0_r := param0;
49    reg1_r := param1;
50    reg2_i := param2;
51    retstate := $normal;
52    goto block_2;
53
54  block_2:
55    stack0_r := reg0_r;
56    stack1_i := reg2_i;
57    assert stack0_r != null;
58    call rs0, ex0 := Account.withdraw.int(stack0_r, stack1_i);
59    assume (forall o: ref, t: name :: true && (o != param0 || t != typ(rval(param0))) &&
60          (o != param1 || t != typ(rval(param1))) ==> inv(t, o, heap));
61    goto block_2_Normal_0, block_2_X_#InsufficientFundsException_0;
62
63  block_2_X_#InsufficientFundsException_0:
64    assume isExceptionalReturnState(rs0) &&
65          isInstanceOf(rval(ex0), $InsufficientFundsException);
66    goto block_2_Handler_#InsufficientFundsException;
67
68  block_2_Handler_#InsufficientFundsException:
69    retstate := $exceptional;
70    assume isInstanceOf(rval(stack0_r), $InsufficientFundsException);
71    exception := stack0_r;
72    goto block_4;
73
74  block_2_Normal_0:
75    assume isNormalReturnState(rs0);
76    goto block_3;
77
78  block_3:
79    goto block_5;
80
81  block_4:
82    reg3_r := stack0_r;
83    havoc stack0_r;
84    assume new(heap, objectAlloc($TransferFailedException)) == rval(stack0_r);
85    heap := add(heap, objectAlloc($TransferFailedException));
86    stack1_r := stack0_r;
87    call rs1, rv1_r, ex1 := TransferFailedException..init(stack1_r);
88    assume (forall o: ref, t: name :: true && (o != param0 || t != typ(rval(param0))) &&
89          (o != param1 || t != typ(rval(param1))) ==> inv(t, o, heap));
90    goto block_4_Normal_1;
91
92  block_4_Normal_1:
93    assume alive(rval(rv1_r), heap);
94    assume isOfType(rval(rv1_r), $TransferFailedException);
95    assume isNormalReturnState(rs1);
96    stack0_r := rv1_r;
97    assert stack0_r != null;
98    goto postX_TransferFailedException;
99
100 block_5:
```

```
101   stack0_r := reg1_r;
102   stack1_i := reg2_i;
103   assert stack0_r != null;
104   call rs2, ex2 := Account.deposit.int(stack0_r, stack1_i);
105   assume (forall o: ref, t: name :: true && (o != param0 || t != typ(rval(param0))) &&
106         (o != param1 || t != typ(rval(param1))) ==> inv(t, o, heap));
107   goto block_5_Normal_2;
108
109 block_5_Normal_2:
110   assume isNormalReturnState(rs2);
111   goto exit;
112
113 postX_TransferFailedException:
114   assume isInstanceOf(rval(exception), $TransferFailedException);
115   assume alive(rval(exception), heap);
116   retstate := $exceptional;
117   goto exit;
118
119 exit:
120   return;
121 }
```

The method contract on lines 4 to 31 contains all necessary pre- and postconditions we discussed in this chapter. Let us explain them here in detail:

Line 4 and 5: The first two input parameters are checked to be of type *Account*.

Line 6: All invariants (i.e. the invariants of all allocated and instantiated objects) must hold.

Line 8: The first two input parameters must not be *null*, and the third parameter must be less or equal to the value of the class field *balance*.

Line 10: The *heap* is the only entity which is modified in the BoogiePL procedure. If fields of class *Account* would be modified in this method, they would not be listed here. Instead, additional pre- and postconditions would be introduced to cope with them.

Line 11: This is the postcondition for the normal method termination. The value of *balance* is modified according to the explicitely defined postconditions in the original Java code or Bytecode respectively.

Line 18: This is the postcondition for the exceptional method termination where an exception of type $TransferFailedException$ is thrown. Here, the explicitely defined exceptional postcondition is established (i.e. the values of *balance* of both *Account* classes remains unchanged).

Line 30: The object invariant of the first two input parameters is checked, for either of them could have been modified during the transaction.

Line 58: Method *withdraw* is called.

Line 59: The invariant of all non-modified variables can be safely assumed. Modified variables at this particular point of time are *stack*0_*r* and *stack*1_*r* (although only *stack*0_*r* is a reference type variable).

Line 63: Exception handling: If an exception of type $TransferFailedException$ was thrown, the program flow is transfered to the appropriate exception handler (line 68).

Line 74: Normal method termination: The program flow is transfered to the next ordinary program statement (line 100).

Line 81f: In case of a $TransferFailedException$, a new object of type $TransferFailedException$ is instantiated and initialized. Therefore, the appropriate constructor $TransferFailed-Exception..init$ is called. Afterwards, the control flow jumps to line 113, where the newly created exception object is thrown and the method $transfer$ is aborted.

Line 100: Method *deposit* is invoked. This method call proceeds similarly to the one before, except that there is no exception thrown in *deposit*.

Chapter 3

Invariant Checks

Object invariants are predicates over a set of object fields and an integral part of static software verification. An invariant is said to *hold* if the objects referred to in the invariant are in a consistent state, i.e. the properties stated in the invariant apply to every initialized object of the same type at any *visible state* during program execution. Visible states are:

- At the beginning of any method which can be invoked from outside the class in which it is defined. Private methods which can only be invoked by the same class do not have to satisfy the object invariant, because the transitional states between those kinds of method executions are not visible to other classes. If the object invariant is required to hold at the beginning of a method, the invariant becomes part of the method's precondition.
- At the end of any method which can be invoked from outside the class in which it is defined. The callee is responsible for establishing the object invariant before the control flow is handed back to the caller. In this case, the object invariant becomes part of the method's postcondition.
- Before and after a public method is invoked. It does not matter whether the invoked method is part of the same object or another one. Before a caller can invoke a method, the object invariant has to be established by all means.

Note that there is an important exception to this rule: Before an object is initialized (usually by calling its constructor), its object invariant can obviously not hold because none of the fields are initialized. The constructor is responsible for initializing all fields properly and to establish the object invariant. Therefore, the object invariant cannot be part of a constructor's precondition in the first place.

3.1 Verification Methodology

According to [19], a technique for specifying and reasoning about invariants must address the following fundamental issues about invariants themselves:

- Encapsulation
- Admissibility
- Semantics
- Modular proof techniques

We are going to address all of these issues briefly in the following four sections.

3.2 Encapsulation

Encapsulation determines the scope in which read or write access in granted for individual variables. A variable x is said to be encapsulated in a class C if access to x is limited to methods of C. In connection with object invariants, it is particularly important to define the encapsulation properties of variables that appear in object invariants.

For our considerations in this chapter however, we do not impose any specific encapsulation properties apart from the restrictions that are already entailed by the Java type system and the Java compiler [23].

3.3 Admissibility

Admissibility essentially defines which variables an invariant may depend on. We need to make sure that no location on which an invariant depends is subject to uncontrolled modification. We therefore limit variables used in invariants to class fields which are declared in the same class as the object invariant itself.

```
1  public class C {
2
3    int f;
4    A a;
5
6    //@ invariant this.f != 0; // admissible invariant
7
8    //@ invariant this.a.f != 0; // inadmissible invariant
9
10 }
```

3.4 Semantics

The semantics of an invariant defines the points in the program flow where an object invariant must hold. For our verification methodology, we use the visible state semantics which we have already explained at the beginning of this chapter.

3.5 Modular Proof Techniques

The goal of a modular proof technique is to show that objects satisfy their invariants without having to examine the entire program. We want to start with a sound proof technique, and try to gradually refine it by removing unnecessary invariant checks.

3.5.1 A Sound Approach

For now, we claim that all object invariants of our whole type universe must hold at all visible states, and that object invariants may only contain predicates about local fields that are defined in the same class as the invariant itself. With this limitations, we can establish an invariant predicate I which leads to a sound but extremely over-approximated check for object invariants:

$$I := \forall o, T : inv_T(o, heap);$$

where o denotes an object reference, T an object type, and $inv_T(o, heap)$ the object invariant of a concrete object of type T referenced by o on a given *heap*. The equation states that for all object references o to objects of type T, the object invariant holds. To be precise, it actually states

that *inv* holds for all possible references o and types T, but $inv_T(o, heap)$ only considers correctly instantiated objects, i.e.

$$inv_T(o, heap) := o.isInstanceOf(T) \Rightarrow explicit\_inv_T(o, heap)$$

where the explicitely defined invariant $explicit\_inv_T(o, heap)$ is only tested if the object referenced by o is indeed an instance of T.

As we have explained above, the invariant I has to be checked at all visible states (particularly in both pre- and postcondition of every visible method) to maintain soundness. Apparently, the formula is not very performant if there are hundreds of potentially modified objects involved. Even the most basic Java programs rely heavily on predefined types from the Java Runtime Library (such as *Object*, *String*, *Exception*, etc.), whose invariants would have to be checked every time we entered or left a method. Instead, we want to reduce the complexity of invariant checks by making I context-dependent.

3.5.2 Refining Invariant Checks

Because we have defined in Section 3.3 that object invariants can only contain fields which are defined in the same class (in contrast to fields defined in other classes and superclasses), an invariant can only be broken if a local field is modified (either by a method in the same class or by another class). Let us look at some examples:

Listing 3.1: Breaking the invariant

```
1  //@ invariant x >= 1;      // invariant 1
2  //@ invariant a != null;   // invariant 2
3
4  this.x = 0;
5  // invariant 1 is broken
6
7  this.a = getA();
8  // invariant 2 is broken (assuming that getA() returns null)
9
10 this.a.f = 0;
11 // invariant of object 'a' might be broken
```

Note that on line 4 and 7, the invariant of the owner object is broken, whereas on line 10, the invariant of object a might be broken. In all three cases, the appropriate invariant has to be checked at the end of the enclosing method.

In order to demonstrate the benefits of only checking the object invariants of those objects which might have broken their invariant, we consider the following example:

Listing 3.2: Checking only broken invariants is crutial

```
1  public clasc C {
2
3    A a, B b, /* ... */ Z z;
4
5    //@ modifies a;
6    public void m() {
7      // assume all invariants to hold
8
9      /* ... */
10
11     // only check invariant of 'a' (and 'this')
12   }
13
14 }
```

In order to keep our invariant checks at a bare minimum, we need to find out exactly which objects might be modified in a given method and check only the invariants of those particular objects. Because the object invariant of a given object can only be violated if fields of that object are modified, we will consider field assignments in particular.

3.6 Formalization Concept

In this section, we will give you the formal specification for checking object invariants. The specification varies depending on whether a normal method, a constructor or a super-constructor is called, or whether a precondition or postcondition is translated. For clarity, we will give a separate formalization for each of these cases, and discuss them separately.

3.6.1 Normal Method Calls

Normal methods assume that all invariants hold before they are called (see Precondition below). In return, they guarantee that all violated invariants are established before the method is left (see Postcondition).

Precondition

Basically, we assume all invariants to hold before a method is called. In the existing heap model, we ought to narrow down the necessary checks by looking only at objects which are actually allocated on the heap, and which are an instance of the appropriate type or subtype respectively.

$$I_{pre}(C.m) := \forall o, T : heap.alive(o) \wedge T \prec o.type \Rightarrow inv_T(o, heap);$$

where $\prec$ is a binary operator denoting subtype relationship ($T \prec o.type$ denotes that T is a subtype of the type of o).

Postcondition

The postcondition looks similar to the precondition, except that we are now more restrictive. At this point, we want to check invariants of modified objects only. If an object was not modified in $C.m$, its invariant will not be checked here. This is perfectly admissible and can be expressed as follows:

$$I_{post}(C.m) := \forall o, T : heap.alive(o) \wedge T \prec o.type \wedge \langle o, T \rangle \in M_{C.m} \Rightarrow inv_T(o, heap);$$

where $M_{C.m}$ is the set of all modified objects of $C.m$.

3.6.2 Constructor Calls

Constructors are handled slightly differently from other methods, because they have other requirements and a different behavior. Constructors do not presume any invariant to hold at all, but rather they are respondible for establishing the object invariant of the current object in the first place.

Precondition

The difference between the precondition of a normal method and the one of a constructor is that for the *this* object, the invariant does not hold initially (since the object has not yet been initialized). So the *this* object is excluded from the invariant check. Because a constructor is responsible for establishing the object invariant, it actually does not require any other object invariant to hold, which is why we say that the precondition of a constructor generally assumes *true*:

$$I_{pre}(C..init) := \top$$

Postcondition

The invariant check at the end of a constructor looks exactly like the one performed at the end of normal methods.

$$I_{post}(C..init) := \forall o, T : heap.alive(o) \wedge T \prec o.type \wedge \langle o, T \rangle \in M_{C..init} \Rightarrow inv_T(o, heap);$$

where $M_{C..init}$ has basically the same semantics as $M_{C.m}$, i.e. it contains the set of all modified variables. The only difference is that $M_{C..init}$ actually contains *all* reference type fields of class C, because they are all modified (i.e. initialized) in the constructor.

3.6.3 Caller

Since our postconditions only ensure that the invariants of modified objects hold, the caller of a method is allowed (or actually obliged) to assume all other invariants to hold according to our proof technique. In fact, it is imperative that all other invariants are assumed by the caller in order to constitute a sound state where all invariants hold again. Thus, the following assumption has to be made after every single method call:

$$I_{after}(C.m) := \forall o, T : heap.alive(o) \wedge T \prec o.type \wedge \langle o, T \rangle \notin M_{C.m} \Rightarrow inv_T(o, heap);$$

Note that this is no assertion, but an assumption, i.e. Boogie does not have to check whether these invariants hold. Rather, we tell Boogie that these invariants hold on any account.

3.7 Implementation

This sections explains the translation process from the implementational point of view. We will show the main issues that made implementing our formalization from the previous section more challenging than expected before working through a complete example of a real-life translation.

3.7.1 Finding Modified Objects

Finding out which objects might have been modified in a given method forms the basis of our new algorithm. At first glance, this problem sounds trivial—at least for a program written in JML. If we take the declaration of the following method for example, it is obvious that we need to check the invariants of the class fields x, y and z respectively in order to guarantee soundness (unless they are value types of course).

```
//@ modifies x, y, z;
public void C.m() {
  /* ... */
}
```

However, this looks a bit more complicated at the level of Bytecode or BoogiePL code. First of all, the *modifies* clause is not translated from BML to BoogiePL. Although Boogie does provide the `modifies` keyword (with the same semantics as its JML counterpart), it is not used in the translation process because the heap model did not allow for it when it was designed. In fact, BoogiePL's *modifies* keyword is only used in B2BPL to express that the *heap* is modified. Thus, every method which is generated in the course of the translation process will contain the following line:

```
modifies heap;
```

where *heap* refers to a publicly declared variable which serves as the global heap of the program. Every field access (both reading and writing) is performed on this heap variable. No further variables will occur in the $modifies$ clause.

But even if we knew what objects were going to be modified in a method (which we actually do because the $modifies$ clause from a JML program is directly translated into a corresponding $modifies$ clause in BML which we could read out directly from the given Bytecode file), it does not exactly help us solve the problem either, because variables are handled differently in BoogiePL.

Imagine a field update in the form $a.f = 0$ in Java. This assignment will be translated into something like $update(heap, fieldLoc(\langle o \rangle, \$f), 0)$ in BoogiePL. Depending on its exact occurrence in the program (i.e. the current size of the stack the be precise), o might be either $stack0\_r$, $stack1\_r$ or in fact any other stack variable. This is why we need to keep track of which variables refer to which objects in order to be able to tell exactly which object is modified if we encounter a field update.

Basically, modified objects can only be either method arguments or references to field locations on the heap, and they are only modified via assignment statements. Method arguments (as well as local register and stack variables for that matter) are usually directly assigned to other local variables. Field updates on the other hand are performed via the *update* function.

During the translation of a Bytecode program, the algorithm looks for these assignment statements and checks whether the target of the assignment is a method argument or the global heap variable. In the latter case, we know that a field was modified if the right hand side of the assignment is a *heap update*. At the end of the current method, we know that we have to check the invariants of all these method arguments and heap locations.

As we have already stated, invariants can only make predications about fields that are declared in the same class. Similarly, pre- and postconditions can only refer to method arguments as well as fields declared in the same class. With other words, locally declared objects within a method body can be used in neither pre- nor postcondition and are thus (almost) irrelevant for reasoning about the contract of the method they are declared in. The invariant of a locally declared object must only hold after its constructor is called, and before (and after) a method is invoked on this object. However, this lies in the responsibility of the called method and does not concern us at the moment. If a locally defined variable is modified by a field update and assigned to either a method argument, the method's return value or another field, the invariant of the target object of the assignment has to be included in the invariant check as well.

Example

To illustrate the problem, consider the following extract of a BoogiePL program:

```
1 implementation C.m (arg1: ref, arg2: ref) returns (result: ref) {
2 init:
3   stack0_r := arg1;
4   heap := update(heap, fieldLoc(stack0_r, $C.f), 5);
5 }
```

On line 4, we notice that field f of class C is modified. However, the only reference given is $stack0\_r$, which we cannot use in a postcondition, because it is only locally available within the method body of $C.m$. In order to check the object invariant of the correct object, we need to go back until we can resolve the reference $stack0\_r$ to the method argument $arg1$, which is allowed to appear in a postcondition.

Algorithm

In order to find those object references relevant for the invariant checks, we step through the instructions of the given Bytecode program (during the translation process). As soon as we detect a field update (indicated by the *putfield* Bytecode instruction), we add the reference name of the receiver object to an internal list of *modified variables*.

As noted above, heap updates are usually performed on local stack variables, i.e. a typical update looks as follows:

```
heap := update(heap, fieldLoc(stack0_r, $C.f), ival(stack1_i));
```

where the value of integer variable $stack1\_i$ is assigned to the field f of an object of type C referenced by the local stack variable $stack0\_r$. Because stack variables are only visible locally within a method, they cannot be part of any method contract. If we examine the program carefully, we note that stack variables contain references to either locally instantiated objects or method arguments. The invariants of the former must be checked if they are assigned to method arguments or if they are used as return values, whereas the invariants of the latter are always checked, as method arguments are exposed references (i.e. they can also be used as output parameters of methods).

The idea behind our algorithm is that we use variable assignments to find out what object a given stack variable refers to (either a method argument, the method's return value or an arbitrary field). For this purpuse, we build up a hash table storing exactly this information. Assume we had the following sequence of assignments in our BoogiePL program:

Listing 3.3: Possible assignments statements in translated code

```
1 reg0_r := param0;
2
3 // ... use reg0_r every time 'this' is used
4 stack0_r := reg0_r;
5
6 // ... modify stack0_r, e.g. change some fields
7 return := stack_0r;
```

On line 1, we assign the first method parameter (which is a reference to the current object, i.e. *this*, if the current method is a non-static method) to a local register variable. Every time we need a reference to the owner object, we use this register variable. This has to be done because in BoogiePL, method arguments cannot be modified. Thus, the input parameter $param0$ has to be assigned to a local variable first. When register variables are used during the program, they are loaded onto the stack (via the *aload* statement in Java Bytecode), which is translated into the assignment on line 3. Finally, on line 5, we have used another assignment which is used when an object is assigned to be the return value of the given method.

As you can see, there are several different situations in which assignments are involved. Every time we encounter an assignment, we add it to our hash table, defining that the variable name on the left hand side references the same object as the variable name on the right hand side. If the stack variable $stack0\_r$ is used in a heap update as reference to the target object (as on line 4 for example), we know that at this time, it is actually the same object as referenced by $param0$, which means that we need to include $param0$ in the postconditional invariant check. Every time we add another tuple $\langle x, y \rangle$ to our hash table (where x is the name of the target variable and y the name of the assigned variable), we need to check whether y has already been added before, and add a new tuple $\langle x, hashmap.get(y) \rangle$ accordingly. In the example above, we try to add the tuple $\langle stack0\_r, reg0\_r \rangle$ to the hash table on line 3, but since $reg0\_r$ is already included (through the assignment on line 1), the tuple $\langle stack0\_r, param0 \rangle$ is added instead.

When we reach the end of a method body, we use the union of all collected variables for the invariant check. This is primarily done to circumvent the difficulties of analyzing conditional assignments as in Listing 3.4.

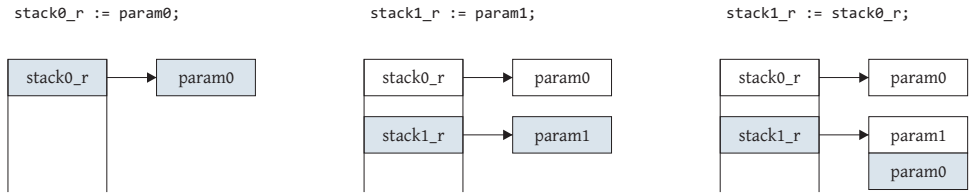

Figure 3.1: Example: In a first assignment (left), *stack*0_*r* is set to *param*0. Both variables names are added as a new pair to our hashtable. In a second assignment (middle), another method argument *param*1 is assigned to a different stack variable *stack*1_*r*. Again, a new tuple is added to the hashtable. In the third assignment, *stack*0_*r* is assigned to *stack*1_*r*. Because the latter has already been added to the table as referencing *param*0, we append *param*0 to the existing entry of *stack*1_*r*. This means that whenever *stack*1_*r* is updated, the object invariants of both *param*0 and *param*1 have to be checked.

Listing 3.4: Conditional heap update

```
1   reg0_r := param0; stack0_r := reg0_r;
2   reg1_r := param1; stack1_r := reg1_r;
3   goto if_condition_then, else;
4 if_condition_then:
5   stack2_r := stack0_r;
6   goto heap_update;
7 else:
8   stack2_r := stack1_r;
9   goto heap_update;
10 heap_update:
11  heap := update(heap, fieldLoc(stack2_r, $C.f), ival(stack1_i));
```

If conditional heap updates are involved, we cannot determine directly whether the heap update was performed on *param*0 or *param*1. In order to avoid unnecessarily complex analyses and resulting postconditions thereof, we check all possibly modified objects. In the example above, we would include both parameters in the (postconditional) invariant check, i.e.

```
ensures inv(typ(rval(param0)), param0, heap) && inv(typ(rval(param1)), param1, heap);
```

3.7.2 Background Theory

During the implementation phase of the new features, we noticed that we needed to add several new axioms to the background theory without which programs could not be verified. The most essential additions to the existing theory are discussed in this section.

Maintain object invariants over heap updates and object allocations

If the value of a reference type field is changed on the heap, we cannot automatically guarantee that the invariant of the owner object still holds. It will be the job of Boogie to verify that this particular invariant is not broken when the method is left. However, we can guarantee that all other invariants are not affected by the heap upate and thus remain intact.

```
// Heap updates do not affect invariants of other objects in the heap:
axiom (forall l: Location, o: ref, t: name, h: Store, v: Value ::
      o != obj(l) ==> inv(t, o, update(h, l, v)) == inv(t, o, h));
```

Heap updates are not only performed on field updates, but also during the allocation of a new object. The newly allocated, but not yet instantiated object is added to the heap, and we need to make sure that this process does not affect existing invariants. However, because the new object is not yet instantiated, its invariant does not hold at this particular point of time.

```
1  // Object allocation preserves existing invariants,
2  // but makes sure that the invariant of the newly allocated object
3  // does not hold.
4  axiom (forall o: ref, t: name, pre_h: Store, new_h: Store ::
5        new(pre_h, objectAlloc(t)) == rval(o) &&
6        new_h == add(pre_h, objectAlloc(t)) ==>
7        !inv(t, o, new_h) && (forall o2: ref, t2: name ::
8        t2 != t || o2 != o ==> inv(t2, o2, new_h) == inv(t2, o2, pre_h)));
```

Annotated JRL classes

Every Java application uses predefined classes from the Java Runtime Library. Most of them are simple, yet powerful classes like *String* or *List*, all of which expose individual interfaces with many different methods, including one or several constructors. Our approach relies on the fact that all of these methods implement invariant checking as well, i.e. they ensure that the invariants of all modified objects hold after they terminate.

The most prominent method call is the constructor of $\$java.class.Object$, which is called explicitely from any other constructor in our translated BoogiePL programs. Let us look at an arbitrary constructor $A..init$ of class A:

```
1  init:
2    reg0_r := param0;
3    goto block_2;
4
5  block_2:
6    stack0_r := reg0_r;
7    call rs0, rv0_r, ex0 := java.lang.Object..init(stack0_r);
8    assume (forall o: ref, t: name :: true ==> inv(t, o, heap));
9    reg0_r := rv0_r;
10   goto block_2_Normal_0;
```

where *java.lang.Object..init* is the constructor name of the *Object* class in Java. It is assumed that this constructor does not modify anything on the heap. Thus, its contract is implemented as follows:

```
1  procedure java.lang.Object..init(param0: ref)
2    returns (retstate: ReturnState, result: ref, exception: ref);
3
4    requires (forall o: ref, t: name ::
5        alive(rval(o), heap) && t <: typ(rval(o)) ==> inv(t, o, heap));
6    ensures result == param0 && result != null && alive(rval(result), heap) &&
7        isInstanceOf(rval(result), $java.lang.Object) &&
8        (forall o: ref :: alive(rval(o), old(heap)) ==> alive(rval(o), heap));
9    ensures (forall o: ref, t: name :: o != param0 ==> inv(t, o, heap));
```

The last ensures clause is responsible for establishing the necessary invariants. As we discussed previously, methods have to guarantee that invariants of modified objects must hold after their execution. The constructor of the *Object* class however has no side-effects whatsoever (all it does is initialize the current object) and therefore does not need to assert any invariants. Line 9 in the source code above results from our algorithm which looks for possibly modified objects and

uses the predicate $o = param0$ to indicate that the current object is correctly instantiated and its invariant holds.

The caller of this particular constructor can then safely assume the invariants of all objects (except for the current object) to hold after the call:

```
call rs0, rv0_r, ex0 := java.lang.Object..init(stack0_r);
assume (forall o: ref, t: name :: o != stack0_r ==> inv(t, o, heap));
```

Additional Axioms

During the implementation of our new features, we had to adapt the existing background theory to our needs. In particular, we needed to add the following axioms:

```
// If a value is updated on the heap, it must be allocated on the same heap:
axiom (forall l: Location, h: Store, v: Value :: alive(v, update(h, l, v)));
```

and for every field $C.f$ of type A:

```
// The type of a given field C.f is a subtype of class £A:
axiom (forall o: ref, h: Store ::
      typ(get(h, fieldLoc(o, C.f))) <: $A);
```

3.7.3 Establishing invariant in constructors

At the end of a constructor, the invariants of the current object as well as all its fields are required to hold. Due to the fact that all fields of an object are set in the constructor in Java Bytecode anyway, its postcondition responsible for the invariant check takes all class field into account as follows:

```
ensures (forall o: ref, t: name ::
          (o == o_1 && t == t_1) || ... || (o == o_n && t == t_n)
          ==> inv(t, o, heap));
```

where `o_i` will be either `param0`, `result`, or in the form `toref(get(heap, fieldLoc(param0, C.f)))` for all class fields $C.f$ of the current object.

3.7.4 Maintaining an invariant over several methods

A normal method expects all invariants to hold, hence it implements the following precondition:

```
requires (forall o: ref, t: name :: inv(t, o, heap));
```

The postcondition which is responsible for checking all relevant invariants is similar to the one described above, except that not all class fields have to be checked:

```
ensures (forall o: ref, t: name ::
          (o == o_1 && t == t_1) || ... || (o == o_n && t == t_n)
          ==> inv(t, o, heap));
```

which we will simplify by eliminating the *forall* quantor as follows:

```
ensures inv(t_1, o_1, heap) && ... && inv(t_n, o_n, heap);
```

where `o_i` will be either `param0` if it is a non-static method, `result` if there is a return value to be checked, or it will be in the form `toref(get(heap, fieldLoc(param0, C.f)))` for all class fields $C.f$ of the current object that were modified.

3.7.5 Checking invariant admissibility

As explained in Section 3.3, we allow invariants to only refer to fields which are declared in the same class. References to fields of other classes (or superclasses) are not admissible. During the static program analysis preceding the translation process, it is checked whether the invariants obey this restriction. In case of an invalid invariant, and error message is displayed stating exactly which invariant is not admissible.

3.8 Example

Let us again take a look at our example from Section 2.4.5. In order to demonstrate the coherences between method calls and invariant checks, we added another class *Bank* to our example which manipulates different object references. In the following example, the method *swapBalanceA* modifies two fields of type *Account* declared in the current class. Note how the method handles the invariants checks for both fields.

Listing 3.5: Class *Bank* (original Java code)

```
1  public class Bank {
2
3    Account accountA = new Account();
4    Account accountB = new Account();
5    Account accountC = new Account();
6
7    //@ requires accountA != null &&
8    //          accountB != null &&
9    //          accountC != null;
10   //@ modifies accountA, accountB, accountC;
11   //@ ensures accountA.balance == 0 &&
12   //          accountB.balance == 0 &&
13   //          accountC.balance == 0;
14   public void rob() {
15     accountA.balance = 0;
16     accountB.balance = 0;
17     accountC.balance = 0;
18   }
19
20   //@ requires accountA != null && accountB != null;
21   //@ modifies accountA, accountB;
22   //@ ensures accountA.balance == \old(accountB.balance) &&
23   //         accountB.balance == \old(accountA.balance);
24   public void swapBalanceAB() {
25     int balanceA = accountA.clear();
26     int balanceB = accountB.clear();
27     accountA.deposit(balanceB);
28     accountB.deposit(balanceA);
29   }
30
31 }
```

Listing 3.6: Invariant of class *Account*

```
1  axiom (forall o: ref, h: Store, t: name ::
2          t <: $accounting.Account ==>
3            (inv(t, o, h) <==> isInstanceOf(rval(o), t) ==>
4             toint(get(h, fieldLoc(o, accounting.Account.balance))) >= 0 &&
5             toint(get(h, fieldLoc(o, accounting.Account.interest))) >= 1));
```

Listing 3.7: Method *swapBalanceAB* of class *Bank* (in BoogiePL)

```
procedure Bank.swapBalanceAB(param0: ref)
  returns (retstate: ReturnState, exception: ref);
  requires alive(rval(param0), heap) && isInstanceOf(rval(param0), $Bank);
  requires (forall o: ref, t: name ::
           alive(rval(o), heap) && t <: typ(rval(o)) ==> inv(t, o, heap));
  requires toref(get(heap, fieldLoc(param0, Bank.accountA))) != null &&
          toref(get(heap, fieldLoc(param0, Bank.accountB))) != null && true;
  modifies heap;
  ensures toref(get(old(heap), fieldLoc(param0, Bank.accountA))) != null &&
         toref(get(old(heap), fieldLoc(param0, Bank.accountB))) != null && true ==>
         toint(get(heap, fieldLoc(toref(get(heap, fieldLoc(param0, Bank.accountA))),
         Account.balance))) == toint(get(old(heap), fieldLoc(toref(get(old(heap),
         fieldLoc(param0, Bank.accountB))), Account.balance))) &&
         toint(get(heap, fieldLoc(toref(get(heap, fieldLoc(param0, Bank.accountB))),
         Account.balance))) == toint(get(old(heap), fieldLoc(toref(get(old(heap),
         fieldLoc(param0, Bank.accountA))), Account.balance)));
  ensures true && inv(typ(rval(param0)), param0, heap) &&
         inv(typ(get(heap, fieldLoc(param0, Bank.accountA))),
         toref(get(heap, fieldLoc(param0, Bank.accountA))), heap) &&
         inv(typ(get(heap, fieldLoc(param0, Bank.accountB))),
         toref(get(heap, fieldLoc(param0, Bank.accountB))), heap);

implementation Bank.swapBalanceAB(param0: ref)
  returns (retstate: ReturnState, exception: ref)
{
  var reg0_r: ref;
  var reg1_i: int;
  var reg2_i: int;
  var stack0_r: ref, stack0_i: int;
  var stack1_i: int;
  var rs0: ReturnState, rv0_i: int, ex0: ref;
  var rs1: ReturnState, rv1_i: int, ex1: ref;

init:
  reg0_r := param0;
  retstate := $normal;
  goto block_2;

block_2:
  stack0_r := reg0_r;
  assert stack0_r != null;
  stack0_r := toref(get(heap, fieldLoc(stack0_r, Bank.accountA)));
  call rs0, rv0_i, ex0 := Account.clear(stack0_r);
  assume (forall o: ref, t: name ::
          true && (o != param0 || t != typ(rval(param0))) ==> inv(t, o, heap));
  assume isOfType(ival(rv0_i), $int);
  assume isNormalReturnState(rs0);
  stack0_i := rv0_i;
  reg1_i := stack0_i;
  stack0_r := reg0_r;
  assert stack0_r != null;
  stack0_r := toref(get(heap, fieldLoc(stack0_r, Bank.accountB)));
  call rs1, rv1_i, ex1 := Account.clear(stack0_r);
  assume (forall o: ref, t: name ::
          true && (o != param0 || t != typ(rval(param0))) ==> inv(t, o, heap));
  assume isOfType(ival(rv1_i), $int);
  assume isNormalReturnState(rs1);
```

```
58    stack0_i := rv1_i;
59    reg2_i := stack0_i;
60    stack0_r := reg0_r;
61    assert stack0_r != null;
62    stack0_r := toref(get(heap, fieldLoc(stack0_r, Bank.accountA)));
63    stack1_i := reg2_i;
64    assert stack0_r != null;
65    heap := update(heap, fieldLoc(stack0_r, Account.balance), ival(stack1_i));
66    stack0_r := reg0_r;
67    assert stack0_r != null;
68    stack0_r := toref(get(heap, fieldLoc(stack0_r, Bank.accountB)));
69    stack1_i := reg1_i;
70    assert stack0_r != null;
71    heap := update(heap, fieldLoc(stack0_r, Account.balance), ival(stack1_i));
72    goto exit;
73
74  exit:
75    return;
76  }
```

Especially note the invariant checks performed in the method specification at the beginning of Listing 3.7:

Line 4: The object invariant of all allocated objects is checked in the precondition.

Line 17: The object invariants of the current object (referenced by *param*0) and both local fields *accountA* and *accountB* are checked in the postcondition, because they have been potentially altered by *swapBalanceAB*.

The method specification of method *Account.clear* (invoked on lines 43 and 53 respectively) is shown in Listing 3.8.

Listing 3.8: Method specifications of *Account.clear*

```
1   procedure Account.clear(param0: ref)
2     returns (retstate: ReturnState, result: int, exception: ref);
3     requires alive(rval(param0), heap) && isInstanceOf(rval(param0), $Account);
4     requires (forall o: ref, t: name ::
5       alive(rval(o), heap) && t <: typ(rval(o)) ==> inv(t, o, heap));
6     modifies heap;
7     ensures (true ==>
8       toint(get(heap, fieldLoc(param0, Account.balance))) == 0 &&
9       result == toint(get(old(heap), fieldLoc(param0, Account.balance)))) &&
10      alive(ival(result), heap) && isOfType(ival(result), $int);
11    ensures true && inv(typ(rval(param0)), param0, heap);
```

Chapter 4

Triggers

Triggers (also known as *Matching Patterns*) are a special feature recently added to BoogiePL which allows to pass information to an underlying theorem prover as of how to instantiate universal quantifiers [1, 4, 13]. They represent a set of terms that together mention all the bound variables, and none of which is just a bound variable by itself. Triggers in BoogiePL can appear in quantification expressions. Some examples are:

Quantification Expression	Trigger
$\forall x : f(x) \geq 0$	$f(x)$
$\forall x : f(x) \geq f(g(x))$	$f(g(x))$
$\forall x, y : f(x) \geq g(x, y)$	$g(x, y)$
$\forall x, y : f(x) \geq g(y)$	$f(x), g(y)$
$\forall x, y : x = y \Rightarrow g(x, y)$	$g(x, y)$

Table 4.1: Quantification expressions and their triggers

A typical axiom might for instance appear in the following form:

```
axiom (forall i: int :: toint(ival(i)) == i);
```

where we added the trigger $toint(ival(i))$, for i alone is already a bound variable by itself. Unfortunately, the example above barely illustrates the issue that triggers actually try to solve. Let us briefly explain the problem. The SMT Solver tries to derive triggers from the axioms automatically. For this purpose, individual terms are added to the set of triggers, beginning from the left hand side. The algorithm stops when all bound variables are contained in at least one of the selected terms. Now consider the following axiom:

```
axiom (forall x :: x != null ==> f(x) == f(next(x)));
```

According to the procedure just described, $f(x)$ will be selected as trigger. As explained in [14], this trigger is not limiting enough though. If $f(X)$ occurs in the e-graph[1], then the quantifier in the axiom above will be instantiated with X, $next(X)$, $next(next(X))$, ... , causing a *matching loop*. A more limiting trigger for this quantifier would be $f(next(x))$, which does not cause a matching loop.

4.1 Modified Background Theory

In Appendix A, we will give you a list of all axioms in our background theory to which we have added one or possibly several triggers. Function and constant declarations are omitted. The

[1] *Equality Graph.* For more information on e-graphs, please refer to corresponding literature like [20].

reference in Appendix A gives you also an overview over the additional axioms added in the course of this thesis.

4.2 Performance Gain

We wanted to measure to which degree our newly introduced triggers improved the performance of the underlying theorem prover (*Simplify* in this particular case). For this purpose, we constructed several test files in JML, compiled them into BML annotated Bytecode (with JACK[2]) and run them through a benchmark program (see Appendix B).

The benchmark program created several processes consecutively, each of which started Boogie with the given BoogiePL file. The runtime of each process was then measured. In Figure 4.2, we have depicted the results of two different programs, both of which were verified fifty times without triggers and fifty times with triggers. The diagrams depict the average runtime computed over all individual test runs.

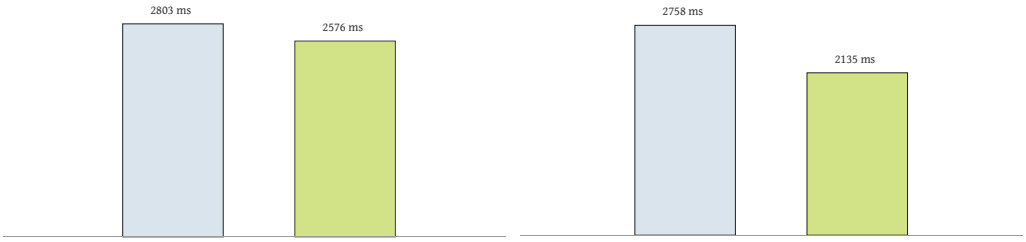

Figure 4.1: In order to analyze the impact of the newly added triggers, we verified sample programs with Boogie and Simplify and measured the time it took the process to complete. We have performed fifty test runs per program and then evaluated the average over all test runs. The first diagram depicts a small program with a couple of object instantiations and method calls, where a slight improvement of 8% could be determined (blue: average runtime without triggers, green: average runtime with triggers). The second diagram shows the results of a basic *QuickSort* algorithm, where field and array operations are involved. We could measure an impressive speed-up of 23%. The improvement might even increase if more axioms (in terms of number and complexity) are involved in the verification process. However, we did not perform any further in-depth tests to fortify these assumption.

[2] see Chapter 5

Chapter 5

IDE for Automatic Bytecode Verification

Right from the beginning of this thesis, it was quite clear that we wanted to integrate the improved B2BPL translator in the *Eclipse IDE* in order to facilitate the verification of Bytecode programs. The idea was to provide a workflow that supports programmers during the development and specifically the verification process of Java applications. Once a program has been written in Java Bytecode, it should be possible to automatically translate it into BoogiePL and have it verified by Boogie directly in Eclipse.

We decided to integrate the Umbra [25] plugin into this workflow (see Section 5.2), mainly because Umbra provides Bytecode editing capabilities based on the *Byte Code Engineering Library* [15]. Unfortunately, the plugin was still under development when this chapter was written, and it was not possible to edit Java Bytecode (and BML annotations) directly. Thus, we used an alternative tool for generating BML annotated Bytecode instead (see Section 5.1).

5.1 Java Applet Correctness Kit

The *Java Applet Correctness Kit* (JACK) [9] is a JML-based verification tool for Java developers designed at *INRIA Sophia Antipolis*. Existing JML annotations can be directly converted to BML and are then embedded in Java class files. Currently, the plugin seems only to work on Unix, but not on Windows. Hopefully, this issue is resolved in the future so that all components work seemlessly together on the same machine.

Similar to other related projects such as ESC/Java2 [6], the developers of JACK made three distinct design choices which had to be met:

Keep Java as the validation language: Programs should be validated directly from their Java code. In order to achieve this, JML is used as base language.

Do not change the working environment: As Eclipse is one of the most popular integrated development environments for Java programming, it was natural to base the verification tool on Eclipse. No additional tools (apart from specific Eclipse plugins) are needed.

Provide different validation levels: The verification tool should provide automatic provers as well as interactive provers.

Basically, developers write JML annotated Java code which is translated to formal lemmas. A collection of different provers (both automatic and interactive) can be used to generate a correctness proof based on these lemmas. An integrated lemma viewer hides the mathematical details from the developer.

Originally, JACK was developed in the software research labs of *Gemplus* (a French company which offers security solutions, especially in conjunction with smart cards). In the course of a major project transfer from Gemplus to INRIA in September 2003, more plugins were integrated, such as automatic annotation generation and annotation propagation. Validation at Bytecode level as well as a few alternative provers (such as Simplify, PVS and Coq) were added too.

JACK then became part of the EVEREST project, which aims at ensuring system security for mobile and embedded applications, where space and computational capacity is limited (i.e. on mobile phones or smart cards). The Eclipse plugin can be downloaded from the EVEREST project website [9]. Currently, the licence is limited to educational and experimental purposes only.

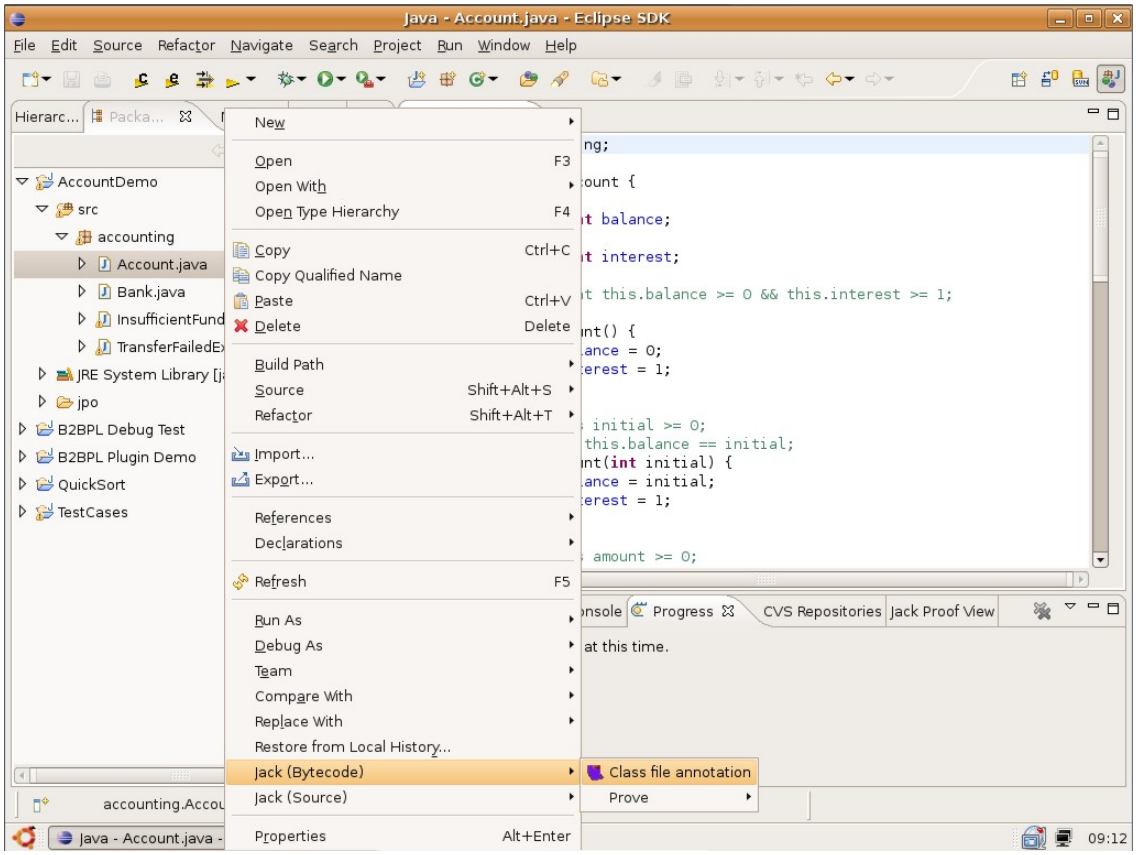

Figure 5.1: Class file annotations: JACK can generate BML annotated Bytecode from JML annotated Java source code.

During this thesis, JACK was primarily used to convert existing JML annotated Java code into BML annotated Bytecode, which proved to be very straightforward. As it is illustrated in Figure 5.1, right-clicking the corresponding *.java* file and selecting *Jack (Bytecode) > Class file annotation* from the context menu will automatically create a new class file containing BML annotations. The generated class file (which is stored in a separate subdirectory) can immediately be copied to the project's output folder to replace the existing, unannotated Java class files.

5.2 Umbra

Umbra is a Bytecode editor plugin for Eclipse. It allows you to view and modify the Bytecode of already compiled Java classes while working on other classe's Java source code. The plugin provides an additional Eclipse text editor component for Java Bytecode, which even supports

syntax highlighting (see Figure 5.2).

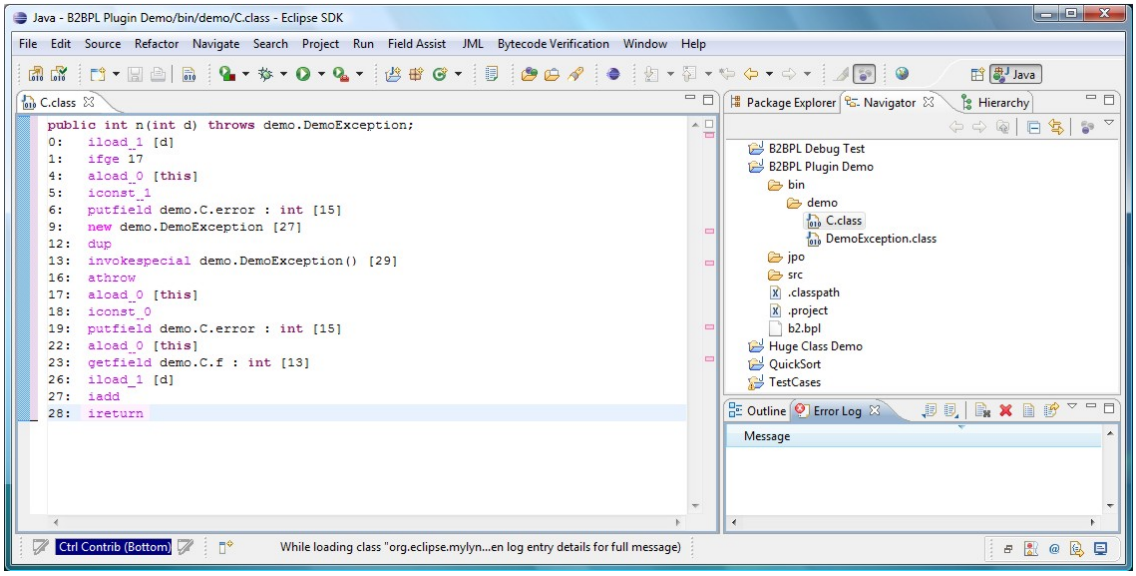

Figure 5.2: Bytecode text editor in Eclipse

The Bytecode is displayed with common instruction labels such as `getfield` and `invokevirtual` rather than cryptic byte values. Unfortunately, the plugin was not yet ready to be integrated into an automatic verifier for BoogiePL because it underwent same heavy changes during this thesis which would have made it quite difficult to adapt to. Nevertheless, Umbra has the potential to become a central component of the aspired framework for automatic Bytecode verification. The clean and concise presentation of the Java Bytecode makes modifications as easy as in Java source code itself. Furthermore, Umbra provides commands to disassemble existing Bytecode back to Java code.

5.3 B2BPL

At the beginning of this thesis, we wanted to implement a B2BPL component as part of the Umbra plugin. However, we finally decided the create an independent Eclipse plugin to translate and verify Java Bytecode for the reasons above. In this section, we want to quickly recapitulate the essential classes of B2BPL (including the modifications accomplished during this thesis) before we explain how the Eclipse plugin encapsulating the translator actually works.

5.3.1 Class Structure

Main

The *Main* class in package *b2bpl* is the main entry point of the translator. In order to parameterize the translation process, *Main* can be initialized with either a list of command line arguments (see Section 5.4.4), or with an object instance of type *Project*.

Project

The class *Project* in package *b2bpl* capsulates a set of properties which can be set to influence the translation process. The translator can be initialized either with a fully instantiated object of type *Project*, or it creates such an object implicitely via its command line arguments (see Section 5.4.4). Its most important properties are the *base directory*, which is used to look for the

given classes if they are not found in Java's class path, the *output file* which contains the final BoogiePL code, and of course a list of *class names* (including their full package path) which are going to be translated.

Other properties contained in the *Project* class are either obsolete or experimental and are therefore not used by the B2BPL plugin.

MethodTranslator

The class *MethodTranslator* located in package *b2bpl.translation* is the actual core of the translator. The class converts a single Bytecode method into BoogiePL by creating a sequence of BoogiePL blocks, each of which is made up of consecutive BoogiePL commands.

Every block starts with a block label which has to be unique in the current method and ends with a list of block labels. The former are used as jump labels, and the latter define undeterministic branches to the corresponding jump labels.

We will not go into further detail here as we did not change the mechanisms behind block- and command generation. Instead, we will focus on two new methods which are responsible for compiling the method specifications: `getRequiresClauses` and `getEnsuresClauses` (see Section 5.4).

5.4 Pre- and Postcondition Generation

5.4.1 Prepare Requires Clauses

Method `getRequiresClauses` is responsible for gathering the predicates which are part of the method's precondition. As explained in Sections 2.3 and 3.6, the correct composition is governed by the signature of the translated method. If the generation of preconditions has to be changed in the future, this is the place to make adjustments.

5.4.2 Prepare Ensures Clauses

Similarly to the method just described, `getEnsuresClauses` is responsible for gathering the predicates for the method's postcondition, with a special focus on exceptional postconditions.

5.4.3 Prepare Invariants

There are three different methods implemented in the *MethodTranslator* to prepare the predicates for invariant checks; one method for every possible occurrence: both invariant checks which are part of a method's pre- or postcondition respectively, as well as the invariant which the caller of a method can safely assume after a the call.

5.4.4 User Guide

B2BPL can be executed from the command line with the options shown below. As mentioned above, the translator can also be invoked in Java directly by creating a new instance of the *Main* class and invoking the method *compile*.

Listing 5.1: B2BPL command line options

```
1 Usage: java b2bpl.Main [<options>] [<files>]
2
3 <options>:
4   -h             Print a help message.
5   -o <outfile>   The output file which contains the resulting BoogiePL code.
6   -s             Translate every class into a separate file.
7   -t             Adds triggers to the axioms (where applicable).
8   -i             Includes invariant checks.
9   -sl            Remove redundancy from logical formulas during translation.
```

```
  -this          Verify the object invariants of the this object only.
  -l             Perform a sound elimination of loops in the BoogiePL program.
  -r             Model runtime exceptions of Bytecode instructions
                 (instead of ruling them out)
  -c <constant>  The magnitude of the largest integer constant to
                 be represented explicitely.
  -basedir <path> Base directory where classfiles are located
                 (if it differs from Java's class path)

<files>
  The class files or type names of the classes to verify.
  They have to be either located in Java's class path, or
  the option -basedir has to be set.
```

Listing 5.2: Calling B2BPL on the command line

```
java b2bpl.Main -o "c:\output.bpl" -basedir "c:\workspace\car\bin" demo.Car demo.Wheel
```

Listing 5.3: Calling B2BPL in your own application

```
// Address translator with command line options...
String[] args = /* ... */
(new b2bpl.Main(args)).compile();

// ...or alternatively with an explicit Project object
Project project = new Project();
(new b2bpl.Main(project)).compile();
```

5.4.5 ASM Framework vs. BCEL

The ASM Framework [7] was chosen to create abstract syntax trees from the Java Bytecode at hand because of its small size and good performance. Umbra however is based on the Byte Code Engineering Library (BCEL) [15], a similar library developed by the Apache Software Foundation. A better integration of Umbra and the B2BPL Eclipse plugin described in Section 5.5 might be achieved by either migrating the latter from ASM to BCEL, or by implementing a translator for converting abstract syntax trees from BCEL to ASM. This way, Java Bytecode would not have to be parsed from Bytecode in order to be verified. Instead, the parsed tree could be directly adopted from Umbra which would prevent the Bytecode from being parsed twice during the verification process and might speed up said operation considerably (supposing that developers edit their Bytecode in Umbra and subsequently invoke B2BPL to verify it).

5.5 B2BPL Eclipse plugin

Converting an existing Java Bytecode file into BoogiePL in Eclipse should be as easy as compiling the original source code. Invoking a command line tool by hand every time you want to verify your Bytecode seems not to be an appropriate solution. As part of an automated verifier for Java Bytecode, we created a plugin which provides two basic options, *conversion* from Bytecode into BoogiePL, and *verification* of Bytecode. The latter assumes that an adequate BoogiePL file already exists in the current project directory. Otherwise, the translation is performed implicitely. Both options are available as toolbar icons in the main toolbar of Eclipse as well as separate menu items.

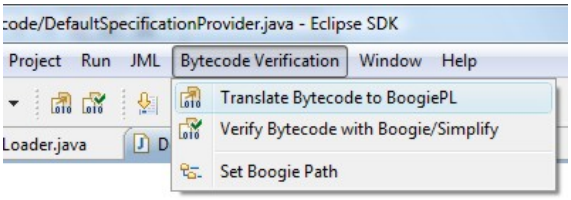

Figure 5.3: B2BPL plugin (menu and toolbar commands) for translating Bytecode to BoogiePL and verifying the generated BoogiePL code in Boogie. The third command is currently used to set the directory where Boogie (and Simplify) are located.

5.5.1 ConversionHandler Class

The *ConversionHandler* converts a given Java project and all its classes into a BoogiePL program. This options can be convenient if you want to further process your BoogiePL code without sending it to Boogie directly.

At this point, it is necessary to include all classes of a Java project because of the obvious dependency between individual classes (e.g. if a class A calls method m of class C, the declaration of $C.m$ has to be available). However, this could be improved in the future. Imagine a Java project consisting of thousands of classes. If you just want to verify a single class C, it would be perfectly fine to only translate this particular class into BoogiePL and use only the specifications of the remaining methods to verify C, omitting their specific implementations altogether.

In fact, this optimization seems to be inevitable when we start including classes from the Java Runtime Library, where the actual implementation is not required (or not even available) for the translation. Only the specifications contained in the method declarations are vital.

However, classes from the Java Runtime Library are not yet incorporated in the translation process. This is another issue which should be tackled in the future (see also Section 6.3).

5.5.2 VerificationHandler Class

The *VerificationHandler* uses the services provided by the *ConversionHandler* to translate Bytecode into BoogiePL first if an appropriate BoogiePL file does not already exist. After the translation process, the translated BoogiePL file is then passed to Boogie. We basically create a new process and invoke Boogie with the previously created BoogiePL file as its only argument.

Because Boogie is primarily designed to run under Windows, we assume that the B2BPL plugin is also used in a Windows environment. Therefore, we start Boogie from Eclipse via the *Runtime* class:

```
Runtime.getRuntime().exec("cmd /c start /b boogie.exe <args>");
```

Because the path to the Boogie executable might not be contained in the system's *PATH* variable, we set the working directory explicitely. The Boogie directory can be customized via the menu *BytecodeVerification* > *SetBoogiePath*.

5.6 The Mobius Workspace

Recently, a preconfigured development environment was introduced which is aimed at simplifying the development on different components of the Mobius project. The new workspace contains all Java projects which are part of Mobius, such as *ESC/Java2*, *Umbra*, and of course our *B2BPL* translator. The workspace is easy to use because hardly any configuration is needed. A couple of plugins are already set up, i.e. *JUnit*, *JMLUnit*, *Checkstyle* and *SVN/CVS*.

All members of the Mobius project have been encouraged to use the Mobius workspace to develop their projects and plugins. Apart from the inherently absence of setup overhead, this will also greatly contribute to more homogeneous source code and targets at satisfying all Mobius standards.

Chapter 6

Conclusion

The achievements of this thesis can be divided into two parts. In Chapters 2 and 3 respectively, we have worked out some unobtrusive improvements addressing method calls and invariant checks by taking the current implementation of B2BPL, formalizing the solutions and extending the existing code base according to those formalizations. In Chapter 5, we have been focusing more on the general issue of an automatic Bytecode verifier and its integration into the development workflow in Eclipse.

6.1 Challenges

Unfortunately, most of the time went into debugging the results from the translation in Boogie, mainly for two reasons. First of all, Boogie lacks some decent debugging capabilities. Although Boogie tells you that a given assumption might not be satisfied, it does not specify what subclause exactly may be broken. Hence, it took a lot of time to track down all the errors and to find their original source. However, we understand there is an ongoing Master Project addressing the problem of insufficient error reporting in Boogie which might mitigate this issue in the future. On the other hand, the heap model in the existing translation framework was quite different from the one used by the native translation from Spec#. In particular, there were some intricate axioms involved in the allocation of new objects on the heap which required extreme caution.

6.2 Achievements

Translating method calls from Java Bytecode to BoogiePL has become much clearer now by the use of Boogie's own *call* statement. Not only has the generated BoogiePL code become easier to read for developers and machines alike (which also makes it easier for the latter to disassemble existing BoogiePL code if required in the future), but it is also more likely that future improvements in Boogie itself (in terms of method calls) can be exploited without further ado, because we let Boogie do all the necessary transformations already.

Invariant checks have also been improved noticeably. Instead of checking only the invariant of the current object (or all objects with the same type as the current object), all relevant invariants are now considered. Invariant admissibility is restricted considerably though, but after all we are dealing with a sound and modular approach, i.e. all invariants which potentially might be broken are checked on any account, and methods can be checked independently (provided that at least the specification of all involved methods are available).

Finally, we made the first move towards a fully automated program verifier for Java Bytecode by integrating B2BPL into an easy-to-use plugin, obviating an unnecessarily complex process to convert compiled Java code into BoogiePL and starting the verifyer manually. It will be exciting to see the plugin being integrated even deeper in Umbra and the Mobius workspace, allowing developers to verify JML annotated Java source code and BML annotated Java Bytecode alike.

6.3 Future Work

Although B2BPL has been developed over a couple of Master Projects now, there is still room for improvement. Some of the many possible (or necessary) extensions are briefly mentioned in this section.

6.3.1 BML annotated Java Runtime Library

This issue has already been mentioned in Chapter 3. In order to be able to use classes from the Java Runtime Library, they need to be BML annotated too. Otherwise, they interfere with the existing theory, because they would not retain all information needed .

As there is already a JML annotated version of the Java Runtime Library available, it would be possible to take this existing source code and convert it to BML annotated Bytecode with JACK (according to Section 5.1).

6.3.2 Extend Object Invariants

Current objects invariants are constricted too much to verify really interesting programs. One of the next steps might be to relax those restrictions and to allow fields of other objects to be included in invariants as well, which, of course, requires substantial modifications on the existing algorithms for invariant checking.

6.4 Acknowledgement

I would like to express my sincere thanks to my supervisor, Hermann Lehner, who supported me patiently during this thesis and provided a lot of valuable help and support. I would also like to thank Prof. Peter Müller for letting me participate in one of his group's projects and giving me the opportunity to dip into the current state of research in software verification and to contribute to one of the ongoing projects.

Appendix A

Extended Background Theory

In this section, we will give you a list of all axioms in our background theory to which we have added one or possibly several triggers. Function declarations are omitted. Triggers are given in curly brackets directly after the corresponding quantifier declaration. Note that for the sake of completeness, we added even those triggers which would be derived automatically.

```
1  axiom (forall i: int, j: int :: { ival(i), ival(j) } ival(i) == ival(j) <==> i == j);
2
3  axiom (forall v: Value :: { ival(toint(v)) } ival(toint(v)) == v);
4
5  axiom (forall i: int :: { toint(ival(i)) } toint(ival(i)) == i);
6
7  axiom (forall o1: ref, o2: ref :: { rval(o1), rval(o2) }
8          rval(o1) == rval(o2) <==> o1 == o2);
9
10 axiom (forall v: Value :: { rval(toref(v)) } rval(toref(v)) == v);
11
12 axiom (forall o: ref :: { toref(rval(o)) } toref(rval(o)) == o);
13
14 axiom (forall i: int, o: ref :: { ival(i), rval(o) } ival(i) != rval(o));
15
16 axiom (forall i: int :: { isValueType(typ(ival(i))) } isValueType(typ(ival(i))));
17
18 axiom (forall t: name :: { isValueType(t), init(t) }
19         isValueType(t) ==> init(t) == ival(0));
20
21 axiom (forall t: name :: { isClassType(t), init(t) }
22         isClassType(t) ==> init(t) == rval(null));
23
24 axiom (forall t: name :: { init(arrayType(elementType(t))) }
25         init(arrayType(elementType(t))) == rval(null));
26
27 axiom (forall v: Value :: { isValueType(typ(v)) }
28         static(v) <==> isValueType(typ(v)) || v == rval(null));
29
30 axiom (forall v: Value :: { arrayLength(v) } 0 <= arrayLength(v));
31
32 axiom (forall o1: ref, f1: name, o2: ref, f2: name ::
33         { fieldLoc(o1, f1), fieldLoc(o2, f2) }
34         fieldLoc(o1, f1) == fieldLoc(o2, f2) <==> o1 == o2 && f1 == f2);
35
36 axiom (forall o1: ref, i1: int, o2: ref, i2: int ::
37         { arrayLoc(o1, i1), arrayLoc(o2, i2) }
```

```
38          arrayLoc(o1, i1) == arrayLoc(o2, i2) <==> o1 == o2 && i1 == i2);
39
40  axiom (forall o1: ref, f1: name, o2: ref, i2: int ::
41          { fieldLoc(o1, f1), arrayLoc(o2, i2) } fieldLoc(o1, f1) != arrayLoc(o2, i2));
42
43  axiom (forall o: ref, f: name :: { obj(fieldLoc(o, f)) } obj(fieldLoc(o, f)) == o);
44
45  axiom (forall o: ref, i: int :: { obj(arrayLoc(o, i)) } obj(arrayLoc(o, i)) == o);
46
47  axiom (forall o: ref, f: name :: { ltyp(fieldLoc(o, f)) }
48          ltyp(fieldLoc(o, f)) == fieldType(f));
49
50  axiom (forall o: ref, i: int :: { ltyp(arrayLoc(o, i)) }
51          ltyp(arrayLoc(o, i)) == elementType(typ(rval(o))));
52
53  axiom (forall t: name :: { typeObject(t) } typeObject(t) != null);
54
55  axiom (forall h: Store, t: name :: { alive(rval(typeObject(t)), h) }
56          alive(rval(typeObject(t)), h));
57
58  axiom (forall t: name :: { allocType(objectAlloc(t)) } allocType(objectAlloc(t)) == t);
59
60  axiom (forall t: name, i: int :: { allocType(arrayAlloc(t, i)) }
61          allocType(arrayAlloc(t, i)) == arrayType(t));
62
63  axiom (forall t: name, i: int, a: Allocation :: { allocType(multiArrayAlloc(t, i, a)) }
64          allocType(multiArrayAlloc(t, i, a)) == arrayType(t));
65
66  // [SW]: A value is alive if it is put onto the heap
67  axiom (forall l: Location, h: Store, v: Value :: { alive(v, update(h, l, v))) }
68          alive(v, update(h, l, v)));
69
70  // Field stores do not affect the values stored in other fields.
71  axiom (forall l1: Location, l2: Location, h: Store, v: Value ::
72          { get(update(h, l1, v), l2) }
73          l1 != l2 ==> get(update(h, l1, v), l2) == get(h, l2));
74
75  // Field stores are persistent.
76  axiom (forall l: Location, h: Store, v: Value :: { get(update(h, l, v), l) }
77          alive(rval(obj(l)), h) && alive(v, h) ==> get(update(h, l, v), l) == v);
78
79  // Object allocation does not affect the existing heap.
80  axiom (forall l: Location, h: Store, a: Allocation :: { get(add(h, a), l) }
81          get(add(h, a), l) == get(h, l));
82
83  // [SW]: Object allocation does not affect existing invariants.
84  axiom (forall o: ref, t: name, h: Store, a: Allocation :: { inv(t, o, add(h, a)) }
85          inv(t, o, add(h, a)) == inv(t, o, h));
86
87  // Field stores do not affect object liveness.
88  axiom (forall l: Location, h: Store, v1: Value, v2: Value ::
89          { alive(v1, update(h, l, v2)) } alive(v1, update(h, l, v2)) <==> alive(v1, h));
90
91  // [SW]: Field stores do not affect the invariants of other fields.
92  axiom (forall l: Location, h: Store, o: ref, t: name, v: Value ::
93          { inv(t, o, update(h, l, v)) }
94          o != obj(l) ==> inv(t, o, update(h, l, v)) == inv(t, o, h));
95
```

```
96  // Alive objects remain alive when a newly allocated object is added to the heap.
97  axiom (forall h: Store, v: Value, a: Allocation :: { alive(v, add(h, a)) }
98          alive(v, h) ==> alive(v, add(h, a)));
99
100 // A newly allocated object becomes alive in the heap it is added to.
101 axiom (forall h: Store, a: Allocation :: { alive(new(h, a), add(h, a)) }
102         alive(new(h, a), add(h, a)));
103
104 // Values reachable from alive objects are themselves alive.
105 axiom (forall l: Location, h: Store :: { alive(get(h, l), h) }
106         alive(rval(obj(l)), h) ==> alive(get(h, l), h));
107
108 // Static values are always alive.
109 axiom (forall h: Store, v: Value :: { alive(v, h) } static(v) ==> alive(v, h));
110
111 // A newly allocated object is not alive in the heap it was created in.
112 axiom (forall h: Store, a: Allocation :: { alive(new(h, a), h) } !alive(new(h, a), h));
113
114 // Allocated objects retain their type.
115 axiom (forall h: Store, a: Allocation :: { typ(new(h, a)) }
116         typ(new(h, a)) == allocType(a));
117
118 // Creating an object of a given type in two heaps yields the same result if liveness
119 // of all objects of that type is identical in both heaps.
120 axiom (forall h1: Store, h2: Store, a: Allocation :: { new(h1, a), new(h2, a) }
121         new(h1, a) == new(h2, a) <==> (forall v: Value ::
122         { alive(v, h1), alive(v, h2), allocType(a) }
123         typ(v) == allocType(a) ==> (alive(v, h1) <==> alive(v, h2))));
124
125 // Two heaps are equal if they are indistinguishable by the alive and get functions.
126 axiom (forall h1: Store, h2: Store :: (forall v: Value :: { alive(v, h1), alive(v, h2) }
127         alive(v, h1) <==> alive(v, h2)) && (forall l: Location ::
128         { get(h1, l), get(h2, l) } get(h1, l) == get(h2, l)) ==> h1 == h2);
129
130 // [SW]: Object allocations preserve existing invariants
131 axiom (forall o: ref, t: name, pre_h: Store, new_h: Store ::
132         { new(pre_h, objectAlloc(t)), inv(t, o, new_h) }
133         new(pre_h, objectAlloc(t)) == rval(o) && new_h == add(pre_h, objectAlloc(t)) ==>
134           !inv(t, o, new_h) && (forall o2: ref, t2: name ::
135             { inv(t2, o2, new_h), inv(t2, o2, pre_h) }
136             t2 != t || o2 != o ==> inv(t2, o2, new_h) == inv(t2, o2, pre_h)));
137
138 // Get always returns either null or a value whose type is a subtype of the (static)
139 // location type.
140 axiom (forall h: Store, l: Location :: { isOfType(get(h, l), ltyp(l)) }
141         isOfType(get(h, l), ltyp(l)));
142
143 // New arrays have the allocated length.
144 axiom (forall h: Store, t: name, i: int :: { arrayLength(new(h, arrayAlloc(t, i))) }
145         arrayLength(new(h, arrayAlloc(t, i))) == i);
146
147 axiom (forall h: Store, t: name, i: int, a: Allocation ::
148         { arrayLength(new(h, multiArrayAlloc(t, i, a))) }
149         arrayLength(new(h, multiArrayAlloc(t, i, a))) == i);
150
151 axiom (forall h: Store, t: name, i: int, a: Allocation ::
152      { isNewMultiArray(new(h, multiArrayAlloc(t, i, a)), h, multiArrayAlloc(t, i, a)) }
153        isNewMultiArray(new(h, multiArrayAlloc(t, i, a)), h, multiArrayAlloc(t, i, a)));
```

```
154
155 axiom (forall v: Value, h: Store, t: name, i: int ::
156         { isNewMultiArray(v, h, arrayAlloc(t, i)) }
157         isNewMultiArray(v, h, arrayAlloc(t, i)) <==> !alive(v, h) &&
158         typ(v) == arrayType(t) && arrayLength(v) == i);
159
160 axiom (forall v: Value, h: Store, t: name, i: int, a: Allocation ::
161         { isNewMultiArray(v, h, multiArrayAlloc(t, i, a)) }
162         isNewMultiArray(v, h, multiArrayAlloc(t, i, a)) <==> !alive(v, h) &&
163         typ(v) == arrayType(t) && arrayLength(v) == i &&
164         (forall e: int :: { isNewMultiArray(get(h, arrayLoc(toref(v), e)), h, a) }
165           isNewMultiArray(get(h, arrayLoc(toref(v), e)), h, a) &&
166           multiArrayParent(get(h, arrayLoc(toref(v), e))) == v &&
167           multiArrayPosition(get(h, arrayLoc(toref(v), e))) == e));
168
169 axiom (forall b: bool :: { bool2int(b) } bool2int(b) == 0 <==> b == false);
170
171 axiom (forall b: bool :: { bool2int(b) } bool2int(b) != 0 <==> b == true);
172
173 axiom (forall i: int :: { int2bool(i) } int2bool(i) == false <==> i == 0);
174
175 axiom (forall i: int :: { int2bool(i) } int2bool(i) == true <==> i != 0);
176
177 axiom (forall v: Value, t: name :: { isOfType(v, t), typ(v) <: t }
178         isOfType(v, t) <==> v == rval(null) || typ(v) <: t);
179
180 axiom (forall v: Value, t: name :: { isInstanceOf(v, t), typ(v) <: t }
181         isInstanceOf(v, t) <==> v != rval(null) && typ(v) <: t);
182
183 axiom (forall v: Value, t: name :: { isInstanceOf(v, t) }
184         isInstanceOf(v, t) ==> isOfType(v, t));
185
186 axiom (forall b: bool, x: any, y: any :: { ifThenElse(b, x, y) }
187         b ==> ifThenElse(b, x, y) == x);
188
189 axiom (forall b: bool, x: any, y: any :: { ifThenElse(b, x, y) }
190         !b ==> ifThenElse(b, x, y) == y);
191
192 // Defines the set of value types.
193 axiom (forall t: name :: { isValueType(t) }
194         isValueType(t) <==> t == $long || t == $int || ... || t == $char);
195
196 axiom (forall i: int :: { isInRange(i, $long) } isInRange(i, $long) <==>
197         $int#m9223372036854775808 <= i && i <= $int#9223372036854775807);
198
199 axiom (forall i: int :: { isInRange(i, $int) } isInRange(i, $int) <==>
200         $int#m2147483648 <= i && i <= $int#2147483647);
201
202 axiom (forall i: int :: { isInRange(i, $char) } isInRange(i, $char) <==>
203         0 <= i && i <= 65535);
204
205 // Associate the types of integer values to their corresponding value ranges.
206 axiom (forall i: int, t: name :: { isInRange(i, t) }
207         typ(ival(i)) <: t <==> isInRange(i, t));
208
209 // A cast value is in the value range of the target type.
210 axiom (forall i: int, t: name :: { isInRange(icast(i, t), t) }
211         isInRange(icast(i, t), t));
```

```
212
213 // Values which already are in the target value range are not affected by a cast.
214 axiom (forall i: int, t: name :: { isInRange(i, t) }
215         isInRange(i, t) ==> icast(i, t) == i);
216
217 axiom (forall t: name :: { elementType(arrayType(t)) }
218         elementType(arrayType(t)) == t);
219
220 axiom (forall t: name :: { $java.lang.Object <: t }
221         $java.lang.Object <: t ==> t == $java.lang.Object);
222
223 axiom (forall o: ref, h: Store, t: name ::
224         t <: $java.lang.Object ==> (inv(t, o, h) <==>
225         isInstanceOf(rval(o), t) ==> true));
226
227 axiom (forall t: name :: { $java.lang.Cloneable <: t, $java.lang.Object <: t }
228         $java.lang.Cloneable <: t ==>
229         t == $java.lang.Cloneable || $java.lang.Object <: t);
230
231 axiom (forall o: ref, h: Store, t: name :: { inv(t, o, h) }
232         t <: $java.lang.Cloneable ==> (inv(t, o, h) <==>
233         isInstanceOf(rval(o), t) ==> true));
234
235 // ... the same for £java.io.Serializable, £java.lang.Throwable
236
237 axiom (forall t: name :: { arrayType(t) }
238         arrayType(t) <: $java.lang.Object && arrayType(t) <: $java.lang.Cloneable &&
239         arrayType(t) <: $java.io.Serializable && arrayType(t) <: $java.lang.Throwable);
240
241 axiom (forall t1: name, t2: name :: { t1 <: t2 }
242         t1 <: t2 ==> arrayType(t1) <: arrayType(t2));
243
244 axiom (forall t1: name, t2: name :: { t1 <: arrayType(t2) }
245         t1 <: arrayType(t2) ==> t1 == arrayType(elementType(t1)) &&
246         elementType(t1) <: t2);
247
248 axiom (forall s: ReturnState :: { isNormalReturnState(s)
249          s != $normal <==> !isNormalReturnState(s));
250
251 axiom (forall s: ReturnState :: { isExceptionalReturnState(s) }
252         s != $exceptional <==> !isExceptionalReturnState(s));
253
254 axiom (forall i: int, j: int :: { i % j, i / j } i % j == i - i / j * j);
255
256 axiom (forall i: int, j: int :: { i % j } 0 <= i && 0 < j ==> 0 <= i % j && i % j < j);
257
258 axiom (forall i: int, j: int :: { i % j } 0 <= i && j < 0 ==> 0 <= i % j && i % j < 0-j);
259
260 axiom (forall i: int, j: int :: { i % j } i <= 0 && 0 < j ==> 0-j < i % j && i % j <= 0);
261
262 axiom (forall i: int, j: int :: { i % j } i <= 0 && j < 0 ==> j < i % j && i % j <= 0);
263
264 axiom (forall i: int, j: int :: { (i+j) % j } 0 <= i && 0 < j ==> (i+j) % j == i % j);
265
266 axiom (forall i: int, j: int :: { (j+i) % j } 0 <= i && 0 < j ==> (j+i) % j == i % j);
267
268 axiom (forall i: int, j: int :: { (i-j) % j } 0 <= i-j && 0 < j ==> (i-j) % j == i % j);
269
```

```
270 axiom (forall a: int, b: int, d: int :: { a % d, b % d }
271         2 <= d && a % d == b % d && a < b ==> a + d <= b);
272
273 axiom (forall i: int :: { shl(i, 0) } shl(i, 0) == i);
274
275 axiom (forall i: int, j: int :: { shl(i, j) } 0 < j ==> shl(i, j) == shl(i, j - 1) * 2);
276
277 axiom (forall i: int :: { shr(i, 0) } shr(i, 0) == i);
278
279 axiom (forall i: int, j: int :: { shr(i, j) } 0 < j ==> shr(i, j) == shr(i, j - 1) / 2);
280
281 axiom (forall i: int, j: int :: { ushr(i, j) } 0 <= i ==> ushr(i, j) == shr(i, j));
282
283 axiom (forall i: int, j: int :: { ushr(i, j) } 0 < j ==> 0 <= ushr(i, j));
284
285 axiom (forall i: int, j: int :: { and(i, j) } 0 <= i || 0 <= j <==> 0 <= and(i, j));
286
287 axiom (forall i: int, j: int :: { and(i, j) }
288         (0 <= i) == (0 <= j) ==> and(i, j) <= i && and(i, j) <= j);
289
290 axiom (forall i: int, j: int :: { or(i, j) } 0 <= i && 0 <= j <==> 0 <= or(i, j));
291
292 axiom (forall i: int, j: int :: { or(i, j) } 0 <= i && 0 <= j ==> or(i, j) <= i + j);
293
294 axiom (forall i: int, j: int :: { xor(i, j) } (0 <= i) == (0 <= j) <==> 0 <= xor(i, j));
```

Appendix B

Benchmark Program

The benchmark program used in Section 4.2 was written in C#. The source code is listed below (instructions for console output have been omitted).

Listing B.1: Benchmark source code

```
1  using System;
2  using System.Diagnostics;
3  using System.ComponentModel;
4
5  namespace BoogieBenchmark {
6
7    public class BoogieBenchmark {
8
9      public static void Main(string[] args) {
10
11       const int Repetitions = 10;
12
13       Process boogie = new Process();
14       boogie.StartInfo.FileName = "Boogie.exe";
15       boogie.StartInfo.Arguments = String.Join(" ", args);
16
17       try {
18         double overall_time = 0;
19
20         for (int i = 0; i < Repetitions; i++) {
21           boogie.Start();
22
23           while (!boogie.HasExited) System.Threading.Thread.Sleep(2000);
24
25           TimeSpan duration = boogie.ExitTime - boogie.StartTime;
26           overall_time += duration.TotalMilliseconds;
27         }
28
29         double average_runtime = (overall_time / Repetitions);
30       }
31       catch (Win32Exception) { /* error */ }
32     }
33   }
34 }
```

Appendix C

Sample Programs

The following section contains sample programs that were used in Section 4.2 to conduct performance tests. Note that the actual translation was performed directly from Java Bytecode. For the sake of clarity, we will list the original JML annotated Java source code only, which was then compiled with the standard Java SDK compiler (version 5.0) and annotated with JACK (according to Section 5.1) to obtain BML annotated Java Bytecode.

C.1 Simple Demonstration Program

Listing C.1: Demonstration program performing some object instantiations, method calls and field assignments.

```
1  public class C {
2
3    public C() {
4      for (int i = 0; i < 20; i++) {
5        (new A()).setValue(i);
6      }
7    }
8
9  }
10
11 public class A {
12
13   int value;
14
15   //@ modifies value;
16   //@ ensures value == v;
17   public void setValue(int v) {
18     value = v;
19   }
20
21 }
```

C.2 QuickSort

Listing C.2: QuickSort algorithm

```
1  public class QuickSortDemo {
2
3    private static int[] a;
4
5    //@ requires a0 != null && a0.length > 0;
6    public static void sort(int[] a0) {
7      a = a0;
8      quicksort(0, a.length - 1);
9    }
10
11   //@ requires a != null && lo >= 0 && lo < a.length &&
12   //@         hi >= 0 && hi < a.length && lo <= hi;
13   private static void quicksort(int lo, int hi) {
14     int i = lo, j = hi;
15     int x = a[(lo + hi) / 2];
16
17     // Divide
18     while (i <= j) {
19       while (a[i] < x) i++;
20       while (a[j] > x) j--;
21       if (i <= j) {
22         swap(i, j);
23         i++;
24         j--;
25       }
26     }
27
28     // Conquer
29     if (lo < j) quicksort(lo, j);
30     if (i < hi) quicksort(i, hi);
31   }
32
33   //@ requires a != null && i >= 0 && i < a.length && j >= 0 && j < a.length;
34   private static void swap(int i, int j) {
35     int t = a[i];
36     a[i] = a[j];
37     a[j] = t;
38   }
39 }
```

Bibliography

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. *Technical Report, Microsoft Research, Redmond, Washington, USA. University of California, Berkeley, California, USA. Katholieke Universiteit Leuven, Belgium*, 2006.

[2] European Commission. *http://www.europa.eu/*.

[3] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. *Technical Report, Microsoft Research, Redmond, WA, USA*, 2005.

[4] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM, Vol. 52*, 2005.

[5] E.W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM 18 (1975)*, 1975.

[6] ESC/Java2. *http://secure.ucd.ie/products/opensource/ESCJava2/*.

[7] ASM Framework. *http://www.objectweb.org/asm*.

[8] FET Global Computing Proactive Initiative. *http://www.cordis.lu/ist/fet/gc.htm*.

[9] Java Applet Correctness Kit. *http://www-sop.inria.fr/everest/soft/Jack/*.

[10] Java Modelling Language. *http://www.jmlspecs.org/*.

[11] H. Lehner. Byte Code Specification Language and Program Logic. *MOBIUS Deliverable D3.1*, 2006.

[12] H. Lehner and P. Müller. Formal Translation of Bytecode into BoogiePL. *Software Component Technology Group, Department of Computer Science, ETH Zurich, Switzerland*, 2007.

[13] K.R.M. Leino. Automatic verification of summations. *IFIP WG 2.3 meeting 46, Sydney, Australia*, 2007.

[14] K.R.M. Leino and Monahan R. Automatic verification of textbook programs that use comprehensions. *Manuscript KRML 175, Microsoft Research, Redmond, WA, USA and National University of Ireland*, 2007.

[15] Byte Code Engineering Library. *http://jakarta.apache.org/bcel/*.

[16] O.J. Mallo. A Translator from BML annotated Java Bytecode to BoogiePL. *Software Component Technology Group, Department of Computer Science, ETH Zurich, Switzerland*, 2007.

[17] R.C. Martin. The Visitor Family of Design Patterns. *The Principles, Patterns, and Practices of Agile Software Development*, 2002.

[18] B. Meyer. Object-Oriented Software Construction. *Prentice Hall*, 1997.

[19] P. Müller, A. Poetzsch-Heffter, and Leavens G.T. Modular Invariants for Layered Object Structures. *Department of Computer Science, ETH Zurich, Switzerland. Technische Universität Kaiserslautern, Germany. Iowa State University, Ames, Iowa, USA*, 2006.

[20] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1979.

[21] Simplify. *http://www.hpl.hp.com/downloads/crl/jtk/*.

[22] SpecSharp. *http://www.research.microsoft.com/specsharp/*.

[23] Inc. Sun Microsystems. The Java Language Specification. *Third Edition*, 2005.

[24] A. Suzuki. Translating Java Bytecode to BoogiePL. *Software Component Technology Group, Department of Computer Science, ETH Zurich, Switzerland*, 2006.

[25] Umbra. *http://mobius.inria.fr/twiki/bin/view/Tools/ToolsUpdates*.