A Translator from BML annotated Java Bytecode to BoogiePL

Ovidio José Mallo

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

March 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

The goal of this master project is to extend an existing formalization of a translation from Java bytecode to BoogiePL and to provide an implementation for that formalization. Possible improvements to the existing translation thereby include the extension of the set of bytecode instructions supported by the formalization as well as the the translation of various semantic properties of the Java Virtual Machine to BoogiePL. In addition, the translation of BML specifications to BoogiePL shall be formalized and implemented.

Contents

Chapter 1

Introduction

1.1 The Java Virtual Machine

The Java Virtual Machine (JVM) is the execution environment of the Java platform and is described in detail in the JVM specification [13]. The core part of the JVM consists of the bytecode instruction set which defines the units of execution in a bytecode program. The JVM offers a safe execution environment in that all the bytecode instructions which impose some semantic constraints on their operands and on the context in which they are executed will always check for those constraints at runtime and throw an appropriate runtime exception if any of them is violated.

1.2 The Bytecode Modeling Language

The Bytecode Modeling Language (BML) is a specification language designed to be the counterpart of the Java Modeling Language (JML) [12] at bytecode level. As such, BML allows to specify the behavior of a Java bytecode program by annotating it using a subset of the JML specifications. Currently, BML supports all the specifications defined as part of JML Level 0, the subset of JML which should be understood and checked by all JML tools. In addition, it contains several constructs drawn from JML Level 1, which were considered to be indispensable for writing meaningful specifications. The latter include – but are not limited to – loop specifications and history constraints.

Just as much of the popularity of JML stems from the fact that it uses a Java-like notation, BML introduces a slightly modified syntax which better meets the requirements of a bytecode environment. In particular, references to fields are represented as indices into the constant pool and special, bytecode-specific expressions are introduced which for example allow to access values on a method's operand stack. For the actual storage of BML specifications in a class file, the standard JVM mechanism for attaching meta-data to a class and its members is used, namely the use of user-defined class file attributes.

1.3 The BoogiePL language

BoogiePL is an intermediate language for program verification. The language features a range of built-in types as well as a set of basic arithmetic operators and provides several features which make it particularly suitable for expressing the semantics of many of the modern programming languages such as Java. The main flexibility of the language thereby stems from the fact that it provides means for specifying a global theory based on the definition of mathematical functions and axioms as well as an imperative part characterized by its support for procedures and implementations which make it easy to model the semantics of many imperative programming languages.

Chapter 2

Translation of Java bytecode

In this chapter, we present our main contributions to an already existing translation from Java bytecode to BoogiePL [19]. For a better understanding, we first provide a short review of previous work before proceeding to the elaboration of our own contributions.

2.1 Preliminaries and previous work

2.1.1 Sound verification of loops

During static program verification, an annotated input program is transformed into a logical expression which represents the weakest precondition of that program with respect to its specification. The generated expression is usually referred to as the program's *verification condition* and it consists of a first-order logical formula whose validity implies that the program meets its specification. While constructing such a logical expression for an imperative program is straightforward, the presence of loops tends to pose a major challenge for a *sound* verification of programs since the generation of a verification condition requires the input program to be *loop-free*. The latter implies that a transformation must be found which removes all the loops from a given program while preserving the soundness of the verification process.

A straightforward way of removing a given loop in a program is to unroll it to a fixed depth and to replace the remaining iterations by code that terminates without ever producing an error. This is the approach taken by some static program verification systems such as ESC/Java2[1] [9] which, however, is obviously unsound since only errors appearing in the explicitly represented loop iterations will ever be detected.

In our work, we use a novel technique first introduced in [5] which is also used in the Spec# [6] static program verifier and which allows for a sound verification of loops. As we will see later in this document, loops and in particular the aforementioned loop verification methodology employed in our translator are key components in the specification and verification of programs which – interestingly enough – may even have an impact on other parts of the verification process such as the verification of object invariants[2]. For that reason, we shall give a brief overview of the mentioned loop verification technique at this point while also providing a simple example in BoogiePL of the applied loop transformations in order to give a better understanding and intuition to the reader as of how the technique works and what implications it may have on the verification methodology as a whole.

The loop transformations applied as part of the here presented loop verification methodology are probably best understood as a transformation from a *reducible* control flow graph to an *acyclic* control flow graph. A control flow graph is said to be reducible if and only if it is possible to

[1] Actually, the latest version of ESC/Java2 optionally provides a sound verification for loops but, by default, it still handles loops by unrolling them a fixed number of times.

[2] see Section 3.3 for more details

identify a unique loop header for each loop. A loop in turn is uniquely identified by a so-called *back edge* which is defined as an edge in the control flow graph whose target node *dominates* its source node. One node dominating another node thereby means that all paths to the latter pass through the former. This implies that a back edge can be defined more intuitively as being an edge from a node inside a loop to that loop's header node. Since a loop header may have several loops associated to it, we will always refer to a loop in terms of its unique back edge and we define the set of nodes constituing that loop as the *natural loop* of the back edge, thus following standard compiler terminology [3].

The core idea of the loop verification methodology is to transform every loop body in such a way that it represents an arbitrary iteration of the loop which in turn allows for the loop being safely eliminated from the program by removing its back edge. In order for a loop body to be representative for any particular iteration of that loop, all the variables modified within the loop – the so-called *loop targets* – must be given a value that they might hold on any iteration of the loop. This can be done in BoogiePL by computing the set of loop targets for a given natural loop and by introducing a `havoc` command for every such loop target at the beginning of the loop's header node. This ensures that every loop target is assigned an arbitrary value before entering the loop meaning that any information which might be specific to a particular loop iteration is wiped out. While this technique ensures that the loop can now be eliminated without compromising the soundness of the verification process, it inevitably results in a gross overapproximation of the original input program whose verification may now fail even if the program is correct. For that reason, the verification methodology allows for the specification of a set of *loop invariants* for every loop. Since a loop invariant expresses a condition which must hold at the beginning of *every* loop iteration, it can be used in our context to recover part of the information previously lost on the values of the loop targets.

In BoogiePL, we define the expressions of all the `assert` commands appearing within a prefix of *passive* commands of a loop header node as consituting the set of loop invariants of the corresponding loop. Such a definition of implicitly declared loop invariants is justified by the fact that the reducibility property we are assuming on the input control flow graph guarantees us that the sole entry point to a loop is its unique loop header node meaning that any property which can be asserted at the beginning of that node is guaranteed to hold for any particular loop iteration, just as the definition of a loop invariant requires. Unfortunately, the assert commands containing the loop invariants will fail to be validated if they depend on the value of a loop target (what they typically do) as the loop targets are havoc'ed right before the loop invariants are asserted. In order to circumvent this problem, we perform a special transformation which moves the assertions of the loop invariants from the loop header node to the end of all its immediate predecessor nodes in the control flow graph. This in turn justifies that the very same set of invariants can now be *assumed* to hold at the beginning of the loop header node since their validity is ensured along every path leading to a new iteration of the loop. After this transformation, we can safely havoc all the loop targets at the beginning of a loop since the loop's invariants are now checked right *before* entering the loop. In addition, the information contained in the loop invariants is preserved inside the loop body by the introduced set of assumptions which is what we were aiming at.

An example

An example of the entire loop transformation as applied to an excerpt of the implementation of a BoogiePL procedure is given in Figure 2.1.

In the example, a variable `x` is given the initial value 1 and it is then continuously incremented inside a loop until its value becomes 100. After the loop has terminated, we would like to be able to verify that the variable's value indeed is 100 which we ensure by inserting an adequate assertion inside the `exit` block. Since the variable's value is less than 100 before we enter the loop and since we always increase its value by one unit only at each loop iteration, we know that the value of `x` will never be greater than 100. This information is added as a loop invariant to the code by asserting it at the beginning of the `loop` block. The first step of our transformation now consists in moving the loop invariant assertion from the loop header block to all its predecessors while

```
entry:
  x := 1;
  goto loop;

loop:
  assert x <= 100;
  goto body, exit;

body:
  assume x < 100;
  x := x + 1;
  goto loop;

exit:
  assume !(x < 100);
  assert x == 100;
  return;
```

```
entry:
  x := 1;
  assert x <= 100;
  goto loop;

loop:
  assume x <= 100;
  goto body, exit;

body:
  assume x < 100;
  x := x + 1;
  assert x <= 100;
  goto loop;

exit:
  assume !(x < 100);
  assert x == 100;
  return;
```

```
entry:
  x := 1;
  assert x <= 100;
  goto loop;

loop:
  havoc x;
  assume x <= 100;
  goto body, exit;

body:
  assume x < 100;
  x := x + 1;
  assert x <= 100;
  assume false;
  return;

exit:
  assume !(x < 100);
  assert x == 100;
  return;
```

Figure 2.1: Consecutive stages (from left to right) in the sound elimination of a loop

turning the assertion at the loop block itself into an equivalent assumption. As we can see in the middle column of the figure, the loop invariant is now checked right before entering the loop for the first time as well as along the back edge which terminates the current loop iteration. As a next step, we introduce a `havoc` command for the variable `x` at the beginning of the `loop` block since the variable is modified inside the loop and, finally, we remove the loop's back edge as illustrated in the right column of the figure. The `assume false;` command introduced at the end of the loop body indicates that the paths which do not leave the loop can be considered to have terminated succesfully once the loop's invariant has been re-established.

By looking at the final program as resulting from our transformation, one can easily see that the original assertion inside the `exit` block can still be verified based on the combination of the information provided by the loop guard condition and the loop invariant. If no invariant had been specified for the loop, however, the validation of that assertion by the here presented verification methodology would have failed.

2.1.2 Existing translation

Our work is largely based on the translation from Java bytecode to BoogiePL presented in [19]. Since in this document we will usually limit ourselves to the discussion of our own contributions to the existing translation, it is sometimes inevitable that some basic knowledge about the previous translation process and the corresponding heap model is assumed. For that reason, we encourage the reader to study the relevant parts of the cited work which will certainly allow for a better understanding of the here presented extensions.

2.2 Types and subtyping

In this section, we describe how types and the subtype relationships among them are translated to BoogiePL. In addition, we define precisely what parts of the static type information contained directly in a class file or obtained during the dataflow analysis of a bytecode method need to be explicitly translated to BoogiePL in order to provide enough information to the program verifier about the types of values appearing in the program.

Throughout this section, we will often make use of two simple but important helper functions which we frequently use to reason about the types of values in BoogiePL:

```
// Returns whether a value is of the given type, or else, it is the null value.
function isOfType(Value, name) returns (bool);
axiom (forall v: Value, t: name :: isOfType(v, t) <==> v == rval(null) || typ(v) <: t);

// Returns whether a value is not null and of the given type.
function isInstanceOf(Value, name) returns (bool);
axiom (forall v: Value, t: name :: isInstanceOf(v, t) <==> v != rval(null) && typ(v) <: t);
```

The `isOfType` predicate expresses whether a value is of a given type, or else, it is the `null` value. This function will be used to translate much of the type information contained in a class file. As we will see later, the explicit treatment of the null value is often crucial since it avoids that any concrete type is ever associated to the null value by the second term `typ(v) <: t` in the function's axiomatization. This is important since, otherwise, the usual JVM semantics of the null value being of *any* reference type would lead to a contradiction in conjunction with the type axiomatization presented later in this section. For that reason, the use of this function should always be favored over the direct application of the built-in operator `<:` when it comes to expressing that a certain value, which may include the null value, is of a given type. Note also, that the function can also be applied to integer values and value types what we will often do in order to treat integer and reference values uniformly.

The `isInstanceOf` predicate expresses wheter a value is not `null` and of the given type. This function will only ever be used on reference values and class types when it comes to reasoning about instance objects which are of a given type.

2.2.1 Class types

As in [19], every class type referenced during the translation is represented by a `name` constant in the BoogiePL program. The built-in semantics for constants in BoogiePL thereby automatically ensures that the individual types are indeed distinct from each other.

In addition to the declaration of a constant representing the class type, every type reference will result in a set of axioms being generated which define the core properties of the type such as its supertype hierarchy. Throughout the following discussion, we will assume that the declaration of the class type being referenced has the following generic form[3]:

class C **extends** D **implements** J, K, ...

For every such class type referenced during the translation, we generate a simple set of axioms which express what its *immediate* supertypes are by using the built-in `<:` operator. Defining the immediate supertypes only is enough since the partial order operator in BoogiePL is defined to be transitive by its very nature:

```
axiom C <: D;
axiom C <: J;
axiom C <: K;
  ⋮
```

The declaration of those axioms already suffices for successfully verifying that an existing subtype relationship among two class types indeed holds. During our work, however, we have often encountered the necessity for being able to also verify that a class type is *not* a subtype of another class type. Having this kind of information turned out to be of particular importance in conjunction with our translation of object invariants to BoogiePL[4] and may also be beneficial

[3]Declarations of interfaces are translated analogously and, therefore, are not explicitly mentioned at this point.
[4]see Section 3.3.3 for more details

in general for the performance of the verification process since non-existing subtype relationships could then be ruled out from the very beginning, thus avoiding unnecessary case splits being performed by the theorem prover. As will become immediately apparent, however, the above axioms are not expressive enough to show that the class `C` indeed is not a subtype of any other class type not explicitly mentioned by the axioms. This is because axioms on functions (and on built-in operators as well, as in our case) in BoogiePL only *partially* define a function's value. This means that, as in our example, the mere *absence* of a subtype definition between the class `C` and some other class does not automatically imply that this subtype relationship does not hold. Therefore, the following additional axiom is generated for every translated class type reference `C` as defined above:

```
axiom (forall t: name :: { C <: t } C <: t ==> t == C || D <: t || J <: t || K <: t || ···);
```

This axiom automatically rules out any unintended subtype relationship between the class type `C` and any other class type by explicitly defining the set of possible supertypes (again, taking advantage of the transitivity property of the partial order operator).

If the class type `C` is declared to be `final`, it additionally triggers the generation of the following axiom:

```
axiom (forall t: name :: { t <: C } t <: C ==> t == C);
```

2.2.2 Array types

Unlike class types, array types do not give rise to the declaration of a BoogiePL constant representing them but, instead, array types are explicitly constructed by applying the `arrayType` function on a given element type of the array.

In the following, we define how the subtyping rules involving array types are translated to BoogiePL. In a first step, we express which *class types* are defined by the JVM as being supertypes of any array type:

```
// Define the class types which are supertypes of any array type.
axiom (forall t: name :: arrayType(t) <: $java.lang.Object);
axiom (forall t: name :: arrayType(t) <: $java.lang.Cloneable);
axiom (forall t: name :: arrayType(t) <: $java.io. Serializable );
```

By contrast, the subtyping among pairs of array types as defined by the JVM is given by the following axioms:

```
// Constructing array subtypes from existing subtypes.
axiom (forall t1: name, t2: name :: t1 <: t2 ==> arrayType(t1) <: arrayType(t2));

// The subtypes of an array type T[] are the array types whose element types are subtypes of T.
axiom (forall t1: name, t2: name :: { t1 <: arrayType(t2) }
    t1 <: arrayType(t2) ==> t1 == arrayType(elementType(t1)) && elementType(t1) <: t2);
```

In the second axiom above, the subexpression `t1 == arrayType(elementType(t1))` is a convenient and effective way of expressing that the type `t1` is an array type. Keeping this in mind, that axiom simply states that any subtype of an array type is itself an array type and that the corresponding element types of the two array types are themselves in a subtype relationship. A concrete application of that axiom and how it interacts with other axioms presented earlier in this section can be seen in the simple example the following listing.

```
public final class T {

  public void foo(T[] array, T element) {
    array [0] = element;
  }
}
```

Verifying this method requires ensuring that the type of `element` is a subtype of the element type of `array` in order to rule out any possible `ArrayStoreException` occurring at runtime. More formally, this means that we must be able to show that the expression

```
typ(rval(element)) <: elementType(typ(rval(array)))
```

can be derived from the static type information of the method parameters which can be expressed as follows:

```
typ(rval(array)) <: arrayType(T)
typ(rval(element)) <: T
```

To that end, we first use the above term `typ(rval(array)) <: arrayType(T)` to trigger our array-subtyping axiom which yields the following relevant information:

```
elementType(typ(rval(array))) <: T
```

Since the class `T` is declared to be `final`, this expression can be used to trigger our axiom for final class types which in turn allows us to derive the following:

```
elementType(typ(rval(array))) == T
```

Finally, this equivalence on the type `T` can be combined with the above static type information about the method parameter `element`, namely the expression `typ(rval(element)) <: T`, in order to ultimately derive that the type of `element` indeed is a subtype of the element type of `array`, meaning that the occurrence of an `ArrayStoreException` at runtime can be successfully ruled out, as one would expect.

2.2.3 Primitive types

In [19], the support for the JVM's primitive types is limited to the type `int` and no handling of the value ranges of integer values is provided. In the following, we present a simple extension to the translation which provides support for all the integral primitive types and their associated value ranges. For representing the individual primitive types, we introduce appropriate constants of type `name` in BoogiePL and define them as being the only primitive types.

```
// Define the set of value types.
const $long: name, $int: name, $short: name, $byte: name, $boolean: name, $char: name;
axiom (forall t: name :: isValueType(t) <==>
    t == $long || t == $int || t == $short || t == $byte || t == $boolean || t == $char);
```

In order to add support for the value ranges of the individual primitive types, we introduce a new predicate function `isInRange` which expresses whether a given number is within the value range of a certain primitive type as defined by the JVM. Associating that value range information to the actual type of an integer value finally allows for a seamless integration of our value range support into the already existing heap axiomatization.

```
// Returns whether an integer constant lies within the range of a given value type.
function isInRange(int, name) returns (bool);

// Define the value ranges of the individual value types.
axiom (forall i: int :: isInRange(i, $long) <==> −922337··· <= i && i <= 922337···);
axiom (forall i: int :: isInRange(i, $int) <==> −2147483648 <= i && i <= 2147483647);
axiom (forall i: int :: isInRange(i, $short) <==> −32768 <= i && i <= 32767);
axiom (forall i: int :: isInRange(i, $byte) <==> −128 <= i && i <= 127);
axiom (forall i: int :: isInRange(i, $boolean) <==> 0 <= i && i <= 1);
axiom (forall i: int :: isInRange(i, $char) <==> 0 <= i && i <= 65535);
```

```
// Associate the types of integer values to their corresponding value ranges.
axiom (forall i: int, t: name :: typ(ival(i)) <: t <==> isInRange(i, t));
```

2.2.4 Static type information

After having discussed how the JVM type system is axiomatized in BoogiePL, we would like to describe which parts of the static type information directly contained in a class file or computed during the dataflow analysis of the individual bytecode methods need to be translated to BoogiePL in order to provide enough information about the types of values appearing in a program. Note that for associating a type to a given value, we will always make use of the functions `isOfType` and `isInstanceOf` as introduced in Section 2.2 instead of applying the built-in operator `<:` directly. This ensures a proper handling of the `null` value as described in the aforementioned section.

Field and array element types

Whenever a value is retrieved from a given location in the heap, we must provide information about the value's type. We should thereby cover not only the types of fields but also of individual array elements. In addition, if the value's type is a primitive type, we should make sure that the integer value is known to lie within the corresponding value range. Since fields and array elements are handled uniformly in the heap axiomatization by the notion of a `Location` and since the value ranges are immediately associated to the individual primitive types we have defined in BoogiePL, all the above type information can be expressed very concisely by a single axiom:

```
// Get always returns either null or a value whose type is a subtype of the
// (static) location type.
axiom (forall h: Store, l: Location :: isOfType(get(h, l), ltyp(l)));
```

Frame types

In addition to the types of values stored in the heap, we must also provide enough type information about the types of values on the operand stack and in the local registers of a stack frame at specific points in a method's body. More precisely, the following static type information directly stored in the class file or computed during the dataflow analysis is translated to BoogiePL for a given method:

- At the beginning of every method, we assume the static type information of the individual parameters of the method which are then used to initialize the local registers of the method's initial frame. The only reference parameter which is assumed to be non-`null` is the implicit `this` parameter of instance methods. Note that the appropriate type information is also assumed on primitive type parameters which in turn defines the corresponding value range of the integer values.

- At every bytecode instruction invoking a non-`void` method, we assume the static type information of the method's return type which is then pushed on the top of the operand stack.

- At every bytecode instruction invoking a method which is declared to throw an exception (or multiple exceptions), we assume that the exception object being thrown is non-`null` and of the declared exception's type.

- At every bytecode instruction which may implicitly throw a runtime exception, we assume that the exception object representing that runtime exception is non-`null` and of the appropriate type. Note that this point is only of relevance if runtime exceptions are explicitly modeled by the translation as described in Section 2.6.

- At the beginning of every loop, we assume the type information of all the local registers initialized at that point as well as the type information of all the types on the operand stack. This type information is added to the translation in order to preserve enough information on the types of possible loop targets when it comes to applying the loop verification methodology as described in Section 2.1.1.

2.3 Literals

Some of the JVM instructions take on literals as their parameters. In the case of integer numbers, those literals are sometimes encoded as an inherent part of the instruction's opcode but, in general, they are indirectly referenced by an index into the constant pool where an appropriate entry encoding the nature as well as the value of the literal can be found. In the following, we present the kinds of literals which are supported by our translation and we briefly describe how they are mapped to BoogiePL.

Integer literals

While the most natural and straightforward way of translating an integer literal to BoogiePL consists in mapping it to the very same integer number, this approach may in some cases lead to problems, as described in [11]. In particular, if the integer literal is of a large magnitude, one may not always assume that the underlying theorem prover is capable of adequatly handling it. In addition, having an explicit representation of large integers may lead to an important negative impact on the performance of the verification process in conjunction with some theorem provers such as Simplify.

In order to avoid these kinds of issues, integer literals whose magnitude is larger than a given threshold are not explicitly represented by their actual value in BoogiePL but, instead, we introduce a symbolic constant of type `int` for them and use that constant whenever the corresponding integer literal is encountered. In addition, once the translation of the whole program has terminated, we generate a set of axioms for all such symbolic constants which provide information about the relative ordering of the individual integer values represented by the constants. By that approach, the underlying theorem prover is able to reason about the equality of such large integer constants and about their relative magnitude without, however, being able to reason about the concrete value of the constant.

String literals

The `ldc` and `ldc_w` bytecode instructions can be used to extract a string literal from the constant pool and push it on the operand stack of the current method frame. Once the string object is on the stack, it can be operated on as one would do with any other kind of instance object.

In our translation, every string literal encountered in the program triggers the declaration of a constant of type `ref` in BoogiePL which is used whenever the string literal is referenced. The constant represents the string object itself whose properties are given by the following axiom which is generated for every such constant representing a string:

```
axiom (forall h: Store ::
    isInstanceOf(rval($stringLiteral_i), $java.lang.String)
    && alive(rval($stringLiteral_i), h));
```

Note that the above axiom and the fact that strings are represented by constants in BoogiePL only ensures that the objects are known to be instances of type `String`, that they are always alive and that different string constants are indeed distinct from each other. However, we can e.g. not easily provide any information on the length of a string since a string's length in the JVM is always referred to by a normal method call and not by a special-purpose bytecode instruction as is e.g. the case for retrieving the length of an array. This is different from other languages such as C# where the length of a string is represented by a special `length` attribute.

Class literals

As of J2SE 5.0, the bytecode instructions `ldc` and `ldc_w` can also be used to load class literals of the form `Integer.class` from the constant pool and push them on the operand stack of the current method frame. In prior versions of the Java platform, such class literals were instead translated to bytecode by generating a synthetic method which loaded the appropriate class and which returned the single object instance representing it.

In our translation, we support class literals in very much the same way strings are supported, namely by introducing a BoogiePL constant of type `ref` for every class literal which is then given the following properties:

```
axiom (forall h: Store ::
    isInstanceOf(rval( $classLiteral_i ), $java.lang.Class)
    && alive(rval( $classLiteral_i ), h));
```

2.4 Constructors

At the beginning of every constructor, some implicit properties about the state of the `this` object are guaranteed by the JVM. In the following, we present two such properties and how they are handled by our translation.

Unique `this` reference

In the JVM, the allocation and subsequent initialization of a new object is not accomplished by a single bytecode instruction but, instead, the two instructions `new` and `invokespecial` are used for that purpose. The `new` instruction thereby performs the actual object allocation and returns a reference to the new object which can then be used to invoke a constructor on it by using the `invokespecial` instruction. While this implies that a reference to an uninitialized object may be pushed on the JVM's operand stack and that bytecode instructions may operate on such uninitialized objects, the JVM's static bytecode verifier will always ensure that no reference to an uninitialized object is ever passed to the heap or used as a method parameter before a constructor has been invoked on it. This in turn guarantees that whenever a constructor is invoked on a new object, that object is *not aliased*.

This is something we can take advantage of inside a constructor by explicitly stating that the only reference to the this object at the beginning of a constructor is given by the value of the first local register of the constructor's initial frame. Assuming the implicit this parameter to the constructor is represented by the variable `param0` in BoogiePL, this can easily be accomplished by the following two steps:

- For every parameter $\texttt{param}_i$ other than the this parameter, we produce the assumption

  ```
  assume param0 != param_i;
  ```

- In addition, we state that the this object has no aliases in the heap:

  ```
  assume (forall l: Location :: rval(param0) != get(heap, l));
  ```

Note that while having the above information explicitly available for the verification process may seem to be of limited importance at first sight, we have found several important use cases whose successful verification depends on the above information being available. A simple example illustrating such a use case is given in the following listing.

```
public class ListHolder {

  //@ invariant list != null;
  protected List list;
```

```
  //@ requires other != null;
  public ListHolder(ListHolder other) {
    this. list  = other. list ;
  }
}
```

In the example, a simple copy constructor is implemented which assigns the field `list` of a given object to the same field of the `this` object. As we can see, the class' invariant expresses that the `list` field should always contain a non-null reference. The key point is that at the beginning of a constructor, we may assume the invariants of all allocated objects *but* the `this` object to hold. This implies that if a theorem prover is not given enough information to deduce that the object `other` is indeed distinct from the `this` object, the former's invariants cannot be safely assumed in the constructor's prestate, meaning that the object invariant of the `this` object cannot be guaranteed to hold at the end of the above constructor. Passing the information about the uniqueness of the `this` reference at the beginning of a constructor to the theorem prover, by contrast, allows for a successful verification of the above example.

Initial values of variables

Another fact which is ensured by the JVM is that at the beginning of every constructor, all the *instance* fields of the `this` object are initialized to their default value.

One possible way of translating this information to BoogiePL consists in producing an explicit assumption for *every* single field of the this object (including the set of inherited fields) which states that the field indeed holds the appropriate default value. This is the approach taken by Spec#. What we do, instead, is defining a function which returns the initial value for a variable of a given type and use that function to declare all the fields of the `this` object as being initialized to that value at the beginning of every constructor which can be done by a single axiom. In order to define what the initial value for a variable of a given type is, we use a slight variation of the `init` function introduced in [19] which is now axiomatized as follows:

```
// Returns the initial value for a variable of the given type.
function init(name) returns (Value);

// Define the default values of value types, class types, and array types.
axiom (forall t: name :: isValueType(t) ==> init(t) == ival(0));
axiom (forall t: name :: isClassType(t) ==> init(t) == rval(null));
axiom (forall t: name :: init(arrayType(elementType(t))) == rval(null));
```

Note that the expression `arrayType(elementType(t))` is an effective way we use for representing the set of array types.

Using that function, all the instance fields of the `this` object can be assumed to hold their initial values by the following simple quantification:

```
assume (forall f: name :: get(heap, fieldLoc(param0, f)) == init(fieldType(f)));
```

2.5 Selected bytecode instructions

As part of our work, we have considerably extended the set of bytecode instructions supported by the translation of Java bytecode. Since the translation of many of them is rather straightforward and similar to instructions supported in previous work [19], we only cover a selected set of instructions at this point. An overview of all supported bytecode instructions can be found in Appendix A.

2.5.1 Value cast instructions

In the JVM, a set of instructions are provided for casting among different primitive types, where those instructions are also used to simulate arithmetic operations on low-precision integer values for which no dedicated bytecode instructions exist. If, for example, one wants to perform a `short` arithmetic addition, this is simulated by first performing the addition using `int` arithmetic (`iadd`) and then casting the result back to the value range of a `short` (`i2s`).

Based on our previous axiomatization of value ranges for different primitive types (see Section 2.2.3), we can provide support for value cast instructions by introducing a special casting function which casts a given integer number to the value range of a specified primitive type. The definition and axiomatization of that function is given as follows:

```
// Casts an integer value to the value range of the given value type.
function icast(int, name) returns (int);

// A cast value always lies within the value range of the target type.
axiom (forall i: int, t: name :: isInRange(icast(i, t), t));

// Values which already are within the target value range are not affected by a cast.
axiom (forall i: int, t: name :: isInRange(i, t) ==> icast(i, t) == i);
```

Note that the axiomatization of the casting function does not cover all the aspects specified by the JVM. In particular, if the value we are casting is not within the target value range, we do not say anything about the result of the cast since, in that case, the properties of the cast value are not easily expressible in BoogiePL. The above function is immediately used to translate any value cast instruction encountered in a method.

Implicit value casts

A problem related to the translation of explicit value cast instructions is the handling of value range casts which are specified implicitly by the JVM on the different arithmetic instructions. For example, the JVM defines the instruction `iadd` to perform an `int` addition, i.e. the result is always mapped back to a 32-bit value range. In our current translation, however, those implict casts are *not* considered for practicability reasons since, otherwise, one would almost never know whether an overflow has occurred, meaning that it is very difficult to verify anything based on the result of arithmetic expressions. Note, however, that our decision to not model those implicit casts is a possible source of unsoundness since we are always assuming that no overflow occurs unless the value cast is made explicit. Nevertheless, we believe that the advantages in terms of practicability outweigh the possible soundness issues, a claim which may be considered to be underlined by the fact that other static program verification systems such as Spec# [6] and ESC/Java2 [9] follow the same principle.

2.5.2 Instructions of type `long`

In [19], bytecode instructions operating on values of type `long` have not been included in the translation, mainly due to some technical problems related to the representation of such values in a method frame. In the following, we provide a brief description about the handling of values of type long in the JVM, based on which we will see that support for `long`s can be easily added to the translation what has been done as part of our work.

The only difference between values of type `long` and other integer values is that the former are defined by the JVM specification as occupying two local registers as well as two slots on the operand stack of a method frame. This, in turn, inevitably raises the question whether some special handling is required to account for this subtlety. However, as one can read up in the JVM specification [13], the bytecode verifier only ever allows that a value of type `long` stored in two local registers is accessed by addressing the register with the lower index. In addition, `long` values on the stack are always treated as a single element. Only when it comes to computing the stack height in a given frame, values of type `long` on the stack are considered to indeed consist of two elements.

Based on these points, it should become apparent, that instructions operating on values of type `long` can be easily supported in the translation.

2.5.3 Multidimensional array allocation

In the following, we will describe how the `multianewarray` instruction for allocating multidimensional arrays is translated to BoogiePL. As we will see, the translation of this instruction is more involved if we try to model all the semantic properties it entails.

The `multianewarray` instruction of the JVM is used to allocate multidimensional arrays. Its direct operands are the array's type and the number of dimensions of the array which should be initialized by appropriate sub-arrays. In addition, the array lengths for the individual array dimensions are given by an appropriate number of integer values on the stack.

Since in the existing heap formalization [19], the core abstraction for a newly created object is an `Allocation`, we first of all define a new function in BoogiePL which defines an allocation for multidimensional arrays.

```
// An allocation of a multidimensional array for a given array element type, a given
// array length, and a given allocation describing the element values of the new array.
function multiArrayAlloc(name, int, Allocation) returns (Allocation);

// The type of a multidimensional array allocation.
axiom (forall t: name, i: int, a: Allocation ::
    allocType(multiArrayAlloc(t, i, a)) == arrayType(t));

// The array length of a multidimensional array allocation.
axiom (forall h: Store, t: name, i: int, a: Allocation ::
    arrayLength(new(h, multiArrayAlloc(t, i, a))) == i);
```

Note that the `multiArrayAlloc` function is parameterized by the type of its elements and not of the array itself. While this may be somewhat counter-intuitive, we found it more consistent with the already existing allocation used for one-dimensional arrays given by the function `arrayAlloc` which is also parameterized by the array's element type. The more interesting aspect is that an allocation for a multidimensional array is itself parameterized by an allocation which (recursively) describes the properties of the next dimension of the array. Using the above function, an allocation for the expression `new C[3][4]` could be expressed as follows in BoogiePL:

multiArrayAlloc(arrayType(C), 3, arrayAlloc(C, 4))

By the definition of the above allocation abstraction, it would already be possible to express the allocation of a new multidimensional array in BoogiePL. However, this does not say anything about the initialization of some dimensions of the array which is performed by the JVM by allocating further sub-arrays. In particular, the actual array and all its sub-arrays allocated by the JVM should be known to be (a) new, distinct objects (b) of the correct type, and (c) of the desired length. The function which will be used to attach all those properties to the new arrays is defined as follows:

```
// Returns whether the given value represents a multidimensional array newly created
// in the given heap and described by the given allocation.
function isNewMultiArray(Value, Store, Allocation) returns (bool);

// Associate the isNewMultiArray function to the new function which is then used
// in the actual translation of the bytecode instruction.
axiom (forall h: Store, t: name, i: int, a: Allocation ::
    isNewMultiArray(new(h, multiArrayAlloc(t, i, a)), h, multiArrayAlloc(t, i, a)));
```

Before we proceed to the actual axiomatization of the `isNewMultiArray` function, we must first think about how we want to show that all the allocated sub-arrays are actually distinct from each other, which is the more difficult part to express in BoogiePL. To that end, we notice that every array allocated in the context of a multidimensional array allocation can be uniquely defined by the conjunction of the parent array to which it belongs and its position inside that array. The idea is that if we attach that information to every allocated array, BoogiePL will be able to derive *by contradiction* that all the arrays are indeed pairwise different. For that purpose, we introduce the following two functions:

```
// Defines the parent array of one of the arrays created during the allocation
// of a multidimensional array.
function multiArrayParent(Value) returns (Value);

// Defines the array element index of one of the arrays created during the allocation
// of a multidimensional array.
function multiArrayPosition(Value) returns (int);
```

Finally, we proceed to the axiomatization of the `isNewMultiArray` function. If the function is applied to an `arrayAlloc`, the array it refers to must be one of the leaf arrays which are not further initialized. In that case, the only properties we have to define on the array are that it is a new array allocated in the given heap, and that its type and length are given by the specified allocation.

```
// Define the properties of a leaf array which is created during the allocation
// of a multidimensional array but which, itself, is not further initialized.
axiom (forall v: Value, h: Store, t: name, i: int ::
    isNewMultiArray(v, h, arrayAlloc(t, i))
    <==> !alive(v, h) && typ(v) == arrayType(t) && arrayLength(v) == i);
```

If, on the other hand, the function `isNewMultiArray` is applied to a `multiArrayAlloc`, the array it refers to will itself be further initialized. In that case, again, we define that the array itself is a new array allocated in the given heap, and that its type and length are given by the specified allocation. In addition, we recurse on all the elements of the array and uniquely identify them by their parent array (the current array) and their position inside the array. The latter will ensure that all the allocated arrays are known to be pairwise distinct.

```
// Define the properties of an array which is created during the allocation
// of a multidimensional array and which, itself, is also further initialized.
axiom (forall v: Value, h: Store, t: name, i: int, a: Allocation ::
    isNewMultiArray(v, h, multiArrayAlloc(t, i, a))
    <==> !alive(v, h) && typ(v) == arrayType(t) && arrayLength(v) == i
            && (forall e: int ::
                    isNewMultiArray(get(h, arrayLoc(toref(v), e)), h, a)
                    && multiArrayParent(get(h, arrayLoc(toref(v), e))) == v
                    && multiArrayPosition(get(h, arrayLoc(toref(v), e))) == e));
```

2.6 Runtime exceptions

The JVM offers a safe execution environment in that all the bytecode instructions which impose some semantic constraints on their operands and on the context in which they are executed will always check for those constraints at runtime and throw an appropriate *runtime exception* if any of them is violated. In addition, the execution of any bytecode instruction may give rise to an asynchronous *JVM error* which signals a more severe problem in the platform itself rather than in the actual program being executed. Since JVM errors often indicate that an unrecoverable failure has occurred, they are typically not caught and handled by an application but, instead, lead to the termination of the running program. Therefore, they are only of little interest in our context. Runtime exceptions, by contrast, are often modeled as part of the execution of a program by explicitly catching and handling them or by declaring them in the throws clause of a method's signature.

In [19], runtime exceptions are handled by inserting an appropriate assertion before the translation of the bytecode instruction signaling the exception which rules out the possibility of the runtime exception being thrown. If this assertion fails to be verified, the user can be informed accordingly. While this is also the approach taken by default in our implementation, in what follows, we present an extension to the existing translation which allows for an explicit modeling of runtime exceptions and the program flow they cause.

In order to model the normal and exceptional outcome of the execution of a given bytecode instruction which might throw a runtime exception, we create a synthetic BoogiePL block for each of them whenever such an instruction is encountered in the input program. In case the execution of the instruction triggers a runtime exception, the semantics of the instruction itself should not have any impact on the program state since the instruction has not been successfully executed. Therefore, a branch to the above synthetic blocks is inserted in the BoogiePL program *before* translating the actual bytecode instruction. In the block representing the exceptional execution path, we can then assume the condition under which the runtime exception occurs. This is dependent on the concrete instruction and runtime exception and is defined in the JVM specification for individual bytecode instructions[5]. In addition, we assume the information on the exception object representing the runtime exception which is defined by the JVM specification as being the only object left on the operand stack. Finally, a branch to the appropriate exception handler for the exception is inserted. In the block representing the normal execution of the bytecode instruction, on the other hand, we simply assume the condition under which the runtime exception does not occur and proceed with the translation of the actual bytecode instruction, as usual. Note that if the instruction may throw several runtime exceptions, the same translation process can simply be repeated at this point for the next runtime exception.

An example of this translation scheme for an integer division instruction which might throw an `ArithmeticException` is illustrated in Figure 2.2.

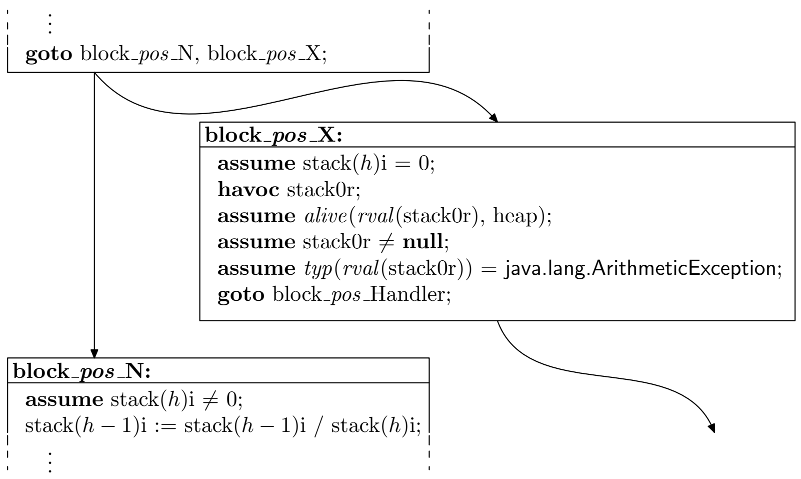
2.7 Exception handlers

The code attribute of every bytecode method contains a possibly empty exception table which lists the set of exception handlers protecting specific code ranges of the method body. In the following, we present a simple extension to the existing translation of Java bytecode [19] which makes the type information inherently contained in every such exception handler available to the verification process.

To that end, we include an additional level of indirection in the translation of every bytecode instruction explicitly throwing an exception by performing the following simple steps:

- At every bytecode instruction explicitly throwing an exception, we generate a new BoogiePL block for every exception handler reachable in the program flow.
- Every branch to an exception handler in the bytecode is translated to an equivalent branch to the corresponding BoogiePL block representing that handler.
- Inside every handler block, we assume the exception object being an instance of the handler's type. In addition, we assume that the exception object is *not* an instance of the type of any previous exception handler for the current instruction.

[5]The set of runtime exceptions modeled by our translation for the individual bytecode instructions can be found in Appendix A.



```
  ⋮
goto block_pos_N, block_pos_X;

block_pos_X:
  assume stack(h)i = 0;
  havoc stack0r;
  assume alive(rval(stack0r), heap);
  assume stack0r ≠ null;
  assume typ(rval(stack0r)) = java.lang.ArithmeticException;
  goto block_pos_Handler;

block_pos_N:
  assume stack(h)i ≠ 0;
  stack(h − 1)i := stack(h − 1)i / stack(h)i;
  ⋮
```

Figure 2.2: An example translation of runtime exceptions to BoogiePL

- For every handler block, the program flow is continued at the first instruction representing the actual handler code in the bytecode method.

Note that the type information assumed inside every handler block is immediately justified by the lookup process employed by the JVM for finding the matching exception handler for a given exception object at runtime [13].

Chapter 3

Translation of BML specifications

In this chapter, we present the translation of BML specifications to BoogiePL. We do this by first precisely defining what the semantics of the individual specifications is and then describing how this semantics is realized by the actual translation to BoogiePL.

3.1 Method specifications

While JML features a rich set of syntactic sugar constructs which are designed to make method specifications more expressive [12], BML only supports a very limited subset of them. In essence, a method specification in BML consists of a set of lightweight *specification cases*. A specification case can be regarded as the unit of specification which describes what the behavior of a method is for a given prestate of that method. More precisely, a specification case in BML is made up of the following standard specification features:

- The method's *precondition* for a specification case is defined by a `requires` clause. If the precondition does not hold in the method's prestate, the method's implementation is not required to satisfy any of the specifications contained in the corresponding specification case.

- The method's *frame condition* for a specification case is defined by a `modifies` clause and it defines the set of heap locations the method's implementation may assign to during its execution. Apart from these explicitly specified locations, a method implementation is always implicitly allowed to modify the state of objects which have been allocated during the current execution of the method's body. Note that the method's frame condition must be satisfied even if the method terminates by throwing an exception. The individual locations specified in a modifies clause are usually referred to as *store references* in BML (and also in JML).

- The method's *normal postcondition* for a specification case is defined by an `ensures` clause. Such a normal postcondition is a two-state predicate which expresses a relationship between the prestate and the poststate of the method which must hold whenever the method's execution terminates normally (i.e. without throwing an exception).

- A method's *exceptional postcondition* for a given exception type is defined by a `signals` clause. As with a normal postcondition, an exceptional postcondition also represents a two-state predicate which, however, expresses the relationship between the prestate and the poststate of the method which must hold whenever the method's execution terminates by throwing an exception of a given type. Note that a method's specification case may have several signals clauses in order to specify different exceptional postconditions for individual exception types.

3.1.1 Desugaring

In the following, we briefly describe a simple desugaring process which transforms a method specification consisting of several specification cases to an equivalent one defined by a single specification case. This allows us to precisely capture the intended semantics of a BML method specification by deriving a single predicate for each specification aspect of a method, namely for its precondition, its postcondition, and its exceptional postcondition. Note that the resulting semantics is equivalent to the semantics of a corresponding subset of a method specification in JML [18]. For the actual notational purposes, we make use of the JML syntax [12].

Our starting point for the desugaring process is given by the following general form of a method specification consisting of several specification cases:

```
/*@
  @   requires $P_1$;
  @   modifies $W_1$;
  @   ensures $Q_1$;
  @   signals ($E_1$ e) $S_1$;
  @ also ··· also
  @   requires $P_n$;
  @   modifies $W_n$;
  @   ensures $Q_n$;
  @   signals ($E_n$ e) $S_n$;
  @*/
```

Note that we provide no explicit treatment for the parts of a method specification which a method *inherits* from its set of overridden methods since inherited specification cases are simply included by aggregation in the overriding method and, thus, are assumed to already be contained in the above method specification. Note also that, for the sake of notational brevity, we assume each specification case containing a single `signals` clause only. A generalization to multiple signals clauses, however, is straightforward.

Before proceeding to the presentation of the desugared form of the above method specification, we try to give a more intuitive explanation as of what its expected semantics is:

- A single specification case can be regarded as a single use case of the method which defines the behavior of that method, given a method's prestate satisfying the `requires` clause of the specification case. In particular, a method's implementation must satisfy its specification for *any* such use case, meaning that the *effective precondition* of a method consists of the *disjunction* of the requires clauses of the individual specification cases. Note that this automatically ensures that a method's effective precondition becomes only ever weaker when its specification is extended by overriding methods in subclasses.

- Any postcondition or frame condition for a given specification case needs only be satisfied in the method's poststate if the corresponding precondition of the specification case holds in the method's prestate.

- A method's frame condition must be satisfied whenever that method's execution terminates, irrespective of whether the method has terminated normally or by throwing an exception. As a consequence, the frame condition of each specification case can be thought of as being conjoined to the postcondition of the corresponding specification case as well as to all its exceptional postconditions. In JML, this can be done by wrapping the frame's store references in an `\assigns_only` expression which allows to incorporate frame conditions in specification expressions.

- A method's implementation must satisfy the normal and exceptional postconditions of *every* specification case whose precondition is satisfied in the method's prestate meaning that a method's *effective postcondition* is esentially the *conjunction* of the postconditions of the

```
/*@
  @ requires $P_1 \vee \cdots \vee P_n$;
  @ modifies $W_1, \cdots, W_n$;
  @ ensures (\old($P_1$) $\Rightarrow Q_1 \wedge$ \assigns_only($W_1$)) $\wedge \cdots \wedge$ (\old($P_n$) $\Rightarrow Q_n \wedge$ \assigns_only($W_n$));
  @ signals (Exception e) (\old($P_1$) $\wedge$ ($e$ instanceof $E_1$) $\Rightarrow S_1 \wedge$ \assigns_only($W_1$))
  @                       $\wedge \cdots \wedge$
  @                       (\old($P_n$) $\wedge$ ($e$ instanceof $E_n$) $\Rightarrow S_n \wedge$ \assigns_only($W_n$));
  @*/
```

Figure 3.1: Desugared form of the original method specification.

individual specification cases. Note that this automatically ensures that a method's effective postcondition becomes only ever stronger when its specification is extended by overriding methods in subclasses.

- A method's exceptional postcondition needs only be satisfied if the exception object being thrown is of the type specified in the `signals` clause. In JML, this additional condition can be specified directly in a specification expression by using the `instanceof` operator in order to avoid requiring the use of `signals` clauses for different exception types.

All of this is subsumed and formalized by the desugared form of the original method specification given in Figure 3.1. Note thereby that while the desugared method specification still contains a `modifies` clause, this information is not required anymore since the actual frame conditions of the individual specification cases have been directly integrated into the method's normal and exceptional postconditions.

3.1.2 Translation to BoogiePL

In [19], a translation of method implementations and method calls has been presented which also models the program flow caused by exceptions which are thrown during a method's execution. That translation thereby allows for a seamless integration of our desugared form of a method specification since it introduces a special BoogiePL block for the normal and all the exceptional poststates of the method which can be used to assert or assume the corresponding postconditions, as required. For that reason, we will not go into further detail in the translation of method specifications as a whole but, instead, we briefly describe how individual frame conditions are translated to BoogiePL.

As was already mentioned, a method is only allowed to modify a given heap location if it is either specified in the method's frame condition, or else, if the location is part of an object which has been allocated during the current execution of the method's body. Assuming that the heap in the method's prestate is denoted by `oldHeap` and the current heap by `heap`, this can be expressed as follows in BoogiePL:

```
assert (forall l: Location ::
    alive(rval(obj(l)), oldHeap) && l $\notin$ W ==> get(heap, l) == get(oldHeap, l));
```

Since fields and array elements are handled uniformly in the heap axiomatization by the notion of a `Location`, the above expression can account for all kinds of store references which might be specified in the modifies clause denoted by `W`.

3.2 Loop specifications

BML supports a very rich set of loop specifications which include some features which are particularly important for the static verification of programs. More precisely, the following aspects of a loop's behavior can be specified in BML:

- A loop *invariant* specifies a predicate which must hold at the beginning of every loop iteration.
- A loop *variant function* (or decreasing function) is used to help prove termination of a loop. It specifies an expression of type `int` which must never be negative at the beginning of a loop iteration and which must decrease by at least one unit between consecutive loop iterations.
- A loop *frame condition* specifies a set of store references which define the set of heap locations the loop's body may assign to during its execution. Apart from these explicitly specified locations, a loop is always implicitly allowed to modify the state of objects which have been allocated inside the loop. Note that loop frame conditions are not supported in JML.

3.2.1 Semantics

In order to provide a more precise definition of the semantics of loop specifications in BML, we use a JML-like syntax to define how the dedicated loop specification features can be desugared to a set of simple assertions, as illustrated in Figure 3.2.

```
//@ loop_invariant I;
//@ decreasing V;
//@ loop_modifies W;
while (G) {
  S;
}
```

```
while (true) {
  //@ assert I;
  //@ assert 0 <= V;
  //@ assert (* W *);
  int loopVariant = V;
  if (!G) {
    break;
  }
  S;
  //@ assert V < loopVariant;
}
```

Figure 3.2: Supported BML loop specifications (left) and the corresponding desugared form (right)

On the left hand side of the figure, we have a generic representation of a loop with the loop guard condition `G` and the sequence of loop body statements `S`. The loop is annotated by an invariant `I`, a variant function `V`, and a modifies clause providing the frame condition `W`. On the right hand side, a transformed but equivalent loop is presented which is better suited to our purposes of annotating it using simple assertions. As one can see, the loop invariant and the non-negativity of the loop variant function are ensured to hold at the beginning of every loop iteration by an appropriate assertion. In addition, the loop variant function is checked to have decreased at least by one at the end of a loop iteration by comparing the current value of the function to the one at the beginning of the current iteration which has been previously stored in a dedicated variable when starting the running loop iteration. The semantics imposed for the loop frame condition is that it must be satisfied at *each* loop iteration. Note that an alternative – and probably equally valid – semantics would be to only require that the loop frame condition is satisfied when we break out of the loop. However, for our purposes of static program verification, the former approach seems far more adequate since it results in the frame condition being semantically equivalent to a loop invariant which allows to retain precious information about the heap's state in conjunction with the loop verification methodology employed in our translation (see Section 2.1.1).

An example

In order to illustrate a typical application of the loop specifications supported in BML, we would like to briefly discuss the simple example in the following listing.

```
public class ArrayShift {

  //@ requires array != null && array.length > 1;
  public static void shift(int[] array) {
    //@ loop_modifies array[0 .. i - 1];
    //@ loop_invariant 0 <= i;
    //@ loop_invariant (\forall int l; 0 <= l && l < i;
    //@                             array[l] == \old(array[l + 1]));
    //@ decreasing (array.length - 1) - i;
    for (int i = 0; i < array.length - 1; i++) {
      array[i] = array[i + 1];
    }
  }
}
```

In the example, we shift all the elements of an array – except the first one – one position to the left, a fact which is expressed as part of the loop's invariant. We believe that the following specification aspects of the example are noteworthy:

- Beside the invariant expressing the actual semantics of the loop body, we also need to explicitly specify the more boring invariant `0 <= i` since the variable `i` is used as an array index which must not be negative. The need for this invariant comes from the fact that the variable `i` is modified inside the loop, meaning that by our loop verification methodology, we are forced to wipe out all the information about the variable's value. This implies that any information which might be required for the verification process must be recovered by the specification of adequate loop invariants. Note that an interesting alternative to the explicit specification of such semantically poor invariants is a process known as *abstract interpretation* [7] which allows for an automatic *inference* of many of those more repetitive loop invariants. This is a technique employed by the Boogie [4] static program verifier which, however, is beyond the scope of our work.

- The specified loop variant function is very simple and rather common for loops that use a running variable which is incremented up to a given upper bound. Note that the non-negativity of a loop variant function always implicitly defines a loop invariant which we might take advantage of during the verification process. In our particular example, the loop variant function implies that `(array.length - 1) - i >= 0` which, in turn, tells us what the upper bound of the running variable is.

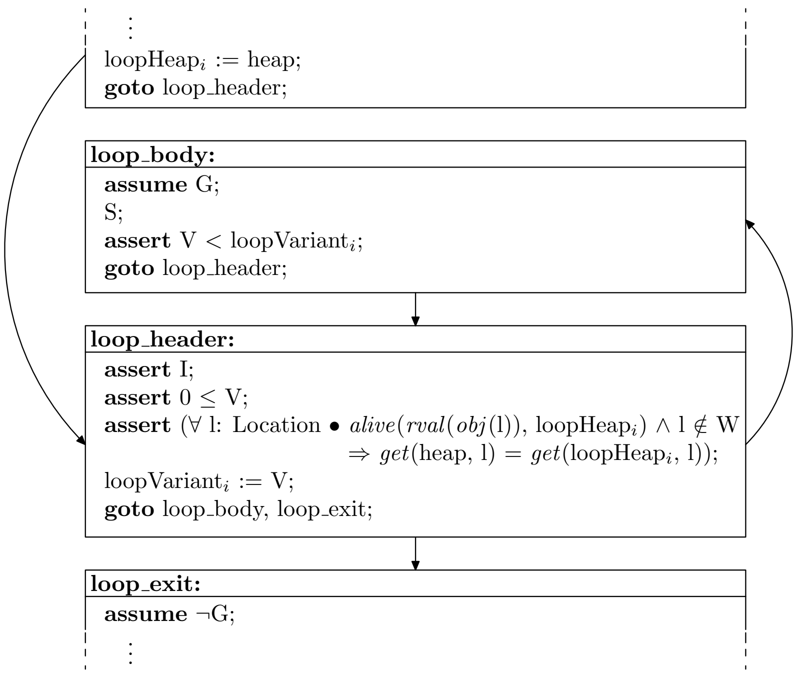
- A rather subtle but notedly important aspect of the loop's specification lies in the definition of the frame condition. The interesting point is that we reference the variable `i` inside the frame condition which, however, has not a constant value during the loop's execution. Intuitively, such a frame condition expresses that at any particular loop iteration, only array elements to the left of the of the current running variable will ever have changed. In particular, this ultimately ensures that for every subsequent loop iteration, the value of the array element `array[i + 1]` is the same as in the method's prestate which is a key requirement for the successful verification of the whole loop. In fact, if we replace the above frame condition by `loop_modifies array[0 .. array.length - 2]`, the loop's verification fails while, otherwise, it succeeds. The definition of such *dynamic* frame conditions as in the above example is an interesting feature we have often taken advantage of such as for the verification of the *insertion sort* algorithm.

- As the attentive reader might already have noticed, the fact that the variable `i` is modified inside the loop is not reflected in the loop's frame condition. As a general rule, we do *not* expect the user to explicitly express his intent of modifying any local variable as part of a loop's frame condition since there is no direct benefit from such specifications, neither for the verification process nor for the user.

3.2.2 Translation to BoogiePL

Based on the desugared form of loop specifications in BML as given in Figure 3.2, their translation to BoogiePL is rather straightforward. In essence, the translation can be summarized by the following individual steps:

- For every loop, we introduce a fresh heap variable $\texttt{loopHeap}_i$ in BoogiePL which is used to store a copy of the heap *before* initiating the loop's execution. This variable is set to the current `heap` at the end of every predecessor block of the loop header block which itself is *not* inside the loop.
- For every loop, we introduce a fresh variable $\texttt{loopVariant}_i$ in BoogiePL which is used to store the value of the loop's variant function at the beginning of a loop iteration.
- At the beginning of every loop header block, we insert an assertion for the loop invariant, for the non-negativity of the loop variant function, and for the loop's frame condition, just as suggested by our desugared loop specification.
- At the end of a loop's body, we check that the loop variant function has decreased at least by one by comparing its current value to the value previously stored in the variable $\texttt{loopVariant}_i$ which holds the function's value as evaluated at the beginning of the current loop iteration.



```
  ⋮
  loopHeap_i := heap;
  goto loop_header;

loop_body:
  assume G;
  S;
  assert V < loopVariant_i;
  goto loop_header;

loop_header:
  assert I;
  assert 0 ≤ V;
  assert (∀ l: Location • alive(rval(obj(l)), loopHeap_i) ∧ l ∉ W
                             ⇒ get(heap, l) = get(loopHeap_i, l));
  loopVariant_i := V;
  goto loop_body, loop_exit;

loop_exit:
  assume ¬G;
  ⋮
```

Figure 3.3: Translating BML loop specifications to BoogiePL

The whole translation is illustrated in Figure 3.3 where we assume a loop with a body represented by `S` and whose guard condition is `G`. In addition, the translation of the loop's invariant (`I`), its variant function (`V`), and its frame condition (`W`) is illustrated.

3.3 Object invariants

In this section, we describe a verification methodology for object invariants as provided by BML. We will thereby mainly concentrate on the actual proof technique for invariants and also discuss how it relates to other parts of the verification process such as the verification of loops. To conclude, we describe how invariants are translated to BoogiePL.

3.3.1 Verification methodology

According to [15], a technique for specifying and reasoning about invariants should address the following fundamental points:

- **Encapsulation:** What parts of a program may assign to the variables used in an invariant?
- **Admissibility:** What variables may an invariant depend on?
- **Semantics:** When do invariants have to hold?
- **Modular proof techniques:** How can one show that objects satisfy their invariants, without examining the entire program?

In the following, we will answer the individual questions above in the context of object invariants as provided by BML. Note, however, that the aspects related to the encapsulation of objects and the admissibility of invariants as described below are currently *not* explicitly enforced by our translator. Instead, our proof technique is applied to the specified object invariants while leaving it up to the user to satisfy the encapsulation and admissiblity properties, if desired. While enforcing those properties directly in the translator is certainly desirable, this could not be done as part of our work in the given time frame.

Object encapsulation

For a sound and modular verification of invariants, one usually needs to provide some kind of object encapsulation which ensures that the fields on which an invariant may depend are protected from unwanted modifications. A basic requirement on the encapsulation policy thereby is that whenever we allow for an object field to be assigned in a given context, the invariants which may depend on that field's value must be ensured to be re-established within the same context in case they had been broken by the field assignment. Some encapsulation properties might be guaranteed by the concrete programming language at hand and its type system, or else, they may be explicitly enforced by only allowing field assignments which satisfy a set of criteria.

In Java, the encapsulation properties guaranteed by the type system itself are very poor or even inexistent, depending on whether we look at the problem from the point of view of a class as a whole, or, of a single object. In fact, the strongest protection on the accessibility of a field which the Java programming language provides is the `private` access modifier which guarantees that a field which is declared to be private can only ever be referenced within a method declared in the same class as the field[1]. This fact can be regarded as a class-level encapsulation property guaranteed by the Java language itself which has been exploited for defining a verification methodology for invariants in which invariants are only allowed to depend on private fields of the invariant's declaring class [14]. What the Java programming language lacks, however, is a notion of encapsulation on an object-level which is what ownership type systems [16] provide.

Since in our verification methodology we always check the invariants of all objects of a method's receiver type, our encapsulation policy can be regarded as a class-level encapsulation which we define as follows: A location $o.f$ for an object o of type C and a field f is said to be encapsulated if the only assignments to $o.f$ occur in methods declared in the class C.

Note that this can be seen as an extension to the classical encapsulation discipline [15] which allows for the assignment of the same set of class fields while also permitting that those fields are

[1]Note that we do not consider inner classes in Java at this point for which this statement is not totally true.

modified on objects other than the `this` object. Note also that we do not impose any kind of restrictions on the access modifiers of the field f nor on whether that field is declared in the class C itself or in any of its superclasses, just as is the case for the classical encapsulation technique.

Admissible invariants

In order for an invariant to be admissible in the context of a sound verification methodology, one must ensure that no location on which the invariant depends is subject to uncontrolled modifications, where the latter largely depends on the encapsulation discipline imposed.

For our verification methodology, we define an invariant declared in a class C as being admissible if each of the access expressions it contains has one of the following forms:

- The access expression represents a field which is declared in the class C.
- The access expression represents some constant expression such as an array length expression or a constant field.

Note that dependencies on fields whose value is constant can be safely permitted since if a newly allocated object is able to once establish its invariant containing such a constant field, the field's value will never change and, thus, it can never lead to the invariant being broken.

A notable restriction of the above admissibility rules is that an invariant may not depend on the elements of an array since arrays in Java are not directly incorporated in an object but, instead, are represented by separate references into the heap. In order to overcome this kind of restriction (and other, similar restrictions), one would require to have some kind of *aliasing control* which is provided by typical ownership type systems [16]. Unfortunately, BML does not offer any support for aliasing control as is e.g. the case in JML.

Invariant semantics

An invariant's semantics defines the points in the program flow at which an object's invariant must hold. For our verification methodology, we use a visible state semantics [17] which essentially defines every pre- and post-state of a method as a visible state, except the pre-state of a constructor which is assumed to be *no* visible state *for the object being initialized*.

Based on this notion of a visible state, we define the invariant semantics for our verification methodology as the requirement that all objects must satisfy their invariants in all their visible states.

Proof technique

In the following, we describe the actual verification of invariants by explicitly stating when the invariants of individual objects are *checked* to hold and at which points in the program we may in turn *assume* that the invariants of certain objects hold. Note that while most aspects of the verification are rather standard, there is an interesting point in the verification of invariants which is immediately related to the loop verification methodology employed in our translation. This will be described below in more detail.

For the subsequent discussion, we assume that the class we are verifying is C and that $inv(C)$ denotes the conjunction of the invariants declared in C and its supertypes. In addition, we will restrict ourselves to the discussion of methods while omitting constructors.

For each method m declared in C, one must *show* the following:

- In the poststate of each execution of m, $inv(C)$ is checked to hold for all objects of type C.
- In the prestate of every method call appearing in m, $inv(C)$ is checked to hold for all objects of type C.
- At the beginning of every iteration of a loop appearing in m, $inv(C)$ is checked to hold for all objects which have been allocated inside that loop.

For each method m declared in C, one may in turn *assume* the following:

- In m's prestate, the invariants of all allocated objects are assumed to hold.
- In the poststate of every method call appearing in m, the invariants of all allocated objects are assumed to hold.

The probably more interesting and less obvious aspect of the above proof technique is that we require that objects allocated inside a loop satisfy their invariants when initiating the next loop iteration. The need for these additional proof obligations comes from the loop verification methodology (see Section 2.1.1) employed in our translation and is best illustrated by looking at the example in the following listing.

```
public class T {

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

  public void foo() {
    for (int i = 0; i < 100; i++) {
      T t = new T();
      t.x = −1;
    }
  }
}
```

In the above example, a new object is allocated inside a loop whose invariant is then broken by an assignment to its field `x`. Since our loop verification methodology requires that all the values modified within a loop are assigned an arbitrary value, however, we will loose all the information about the concrete value of the field `t.x`. This, in turn, implies that we will not be able to reason about that field's value once we break out of the loop, meaning that it is also not possible to check whether such objects allocated and modified inside the loop indeed satisfy their invariants when the method terminates. The key difference between objects allocated *inside* the loop and objects which were already allocated *before* entering the loop thereby is that for the latter, the information lost during the application of our loop verification methodology can be recovered by using invariants while this is obviously not possible for objects which were not allocated when entering the loop. For that reason, we enforce that objects allocated inside a loop satisfy their invariants at the beginning of every loop iteration, which, technically, results in the declaration of an implicit loop invariant which could otherwise not be explicitly declared by the programmer.

3.3.2 Invariants and constructors

In the following, we would like to briefly discuss an example related to the verification of invariants inside a constructor which needs some special handling to be verifiable by our translator. The example is illustrated in Figure 3.4.

On the left hand side of the example, we see a simple class with a constructor which correctly establishes the class' invariant by assigning a new instance of an `ArrayList` to its field `list`. However, the problem is that the constructor call `new ArrayList()` is executed *before* the assignment is performed and, thus, before the invariant has been established. By our verification methodology, however, the invariant of the `this` object should already hold in the prestate of the constructor call for the verification to succeed. Note thereby that, in general, it would be *unsound* to allow for any such method call without enforcing that the invariant of the `this` object holds since at the beginning of every method body, we always assume that the invariants of *all* objects are satisfied. In our particular example, however, the latter is no real issue since we know that the implementation of the `ArrayList()` constructor cannot have a reference to the freshly allocated `this` object, so the latter's invariant need not be enforced in our case. A simple way to provide

```
public class PriorityQueue {

  //@ invariant list != null;
  protected List list;

  //@ modifies this.list;
  public PriorityQueue() {
    super();
    // invariant assertion fails !
    this. list  = new ArrayList();
  }
}
```

```
public class PriorityQueue {

  //@ invariant list != null;
  protected List list;

  //@ modifies this.list;
  public PriorityQueue() {
    this(new ArrayList());
  }

  //@ requires inputList != null;
  //@ modifies this.list;
  //@ ensures this.list == inputList;
  protected PriorityQueue(List inputList) {
    super();
    this. list  = inputList;
  }
}
```

Figure 3.4: Establishing an object invariant by method calls.

enough information to our translator for the example to be verifiable is thereby illustrated on the right hand side of the figure. There, we see that the initialization of the `this` object is now split over two constructors while the problematic constructor call `new ArrayList()` is now invoked *before* the `this` object has been initialized. Since the JVM guarantees us that no reference to the `this` object will ever be passed out to the heap before the `this` object has been initialized by a super-constructor call, we can exploit this information to avoid enforcing the invariant on the `this` object before invoking the constructor `new ArrayList()`. This is a special feature we have implemented in our translator since we believe that an example as the one presented at this point is frequently used in practice.

3.3.3 Translation to BoogiePL

What follows, is the more technical description of the translation of BML invariants and their associated proof technique to BoogiePL.

Translation of invariant declarations

For the representation of the actual invariant declarations, a special BoogiePL predicate function is introduced whose signature reads as follows:

function inv(C: **name**, o: **ref**, h: Store) **returns** (**bool**);

The parameters to the function represent a class `C`, an object `o`, and a heap `h` and the function's intended semantics is that it returns `true` whenever the invariant of the class `C` is satisfied by the object `o` in the given heap `h`. Using this function, every class C referenced during the translation results in the generation of the following axiom in BoogiePL:

axiom (**forall** o: **ref**, h: Store ::
  inv(C, o, h) <==> isInstanceOf(rval(o), C) ==> $Tr[inv(C)]$);

In the above axiom, C is used as a place holder for the BoogiePL `name` constant representing the type C and $Tr[inv(C)]$ stands for the translation of C's invariant from BML to BoogiePL. Note that $inv(C)$ thereby represents the invariants declared not only in C itself but also in all its supertypes. As we can see, such an axiom essentially associates the value of the function `inv(C, o, h)` to the expression of C's invariant. In addition, if `inv(C, o, h)` is applied to an object `o` which is either `null` or not an instance of type C, the function simply returns `true`. The latter is particularly important for ensuring that we never make any assumption on the state of an object which might be `null` since this could be a possible source of unsoundness.

Translation of the proof technique

Using the above function and its axiomatization, it is straightforward to translate the proof technique for invariants presented earlier in this section to BoogiePL. In particular, for ensuring that the invariants of all objects of type C hold in the poststate of every method and in the prestate of every method call, we generate the following proof obligation at the appropriate points in the program during the translation process:

```
assert (forall o: ref :: inv(C, o, heap));
```

Likewise, in the prestate of every method as well as in the poststate of every method call, we may assume that the invariants of all objects hold which can be expressed in BoogiePL as follows:

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

In this case, we quantify not only over all objects but also over all types in order to assume the invariants of all types on all their instances.

Finally, the requirement imposed by the proof technique which states that, at the beginning of every loop, the invariants of all the objects allocated inside that loop must satisfy their invariants, can be expressed by inserting the following assertion at the beginning of every loop header block of the resulting BoogiePL program:

```
assert (forall o: ref :: !alive(rval(o), loopHeap) ==> inv(C, o, heap));
```

Thereby, the above variable `loopHeap` represents the state of the heap as encountered right before entering the execution of the loop.

3.4 Ghost fields

A ghost field [8] is a specification-only field whose value can be explicitly set by a special set specification statement which is similar to a normal field assignment. Thus, ghost fields provide a convenient mechanism for carrying and manipulating some state which is only relevant for specification purposes and to do that in a way which is similar to the handling of normal fields declared in a class.

Since ghost fields can be handled like any other field in our translation and since set statements are easily translated by a normal heap update, we will not go into more detail at this point but, instead, we refer the interested reader to some standard work [8, 12] which covers some use cases and applications of ghost fields.

3.5 Translation example

To better illustrate the translation of BML specifications to BoogiePL, we present an example translation of an annotated class which contains many of the features presented in this chapter. To that end, we first briefly introduce the actual class in Java and annotate it using a JML-like syntax. Subsequently, we present the bytecode for the individual methods which we will translate before finally discussing the resulting BoogiePL code as generated by our translator.

The annotated Java class used for the translation example is given in Listing 3.1.

Listing 3.1: Account class used as the translation example

```
public class Account {
  //@ invariant balance >= 0;
  private int balance;

  //@ requires initial >= 0;
  //@ modifies balance;
  //@ ensures balance == initial;
  public Account(int initial) {
    this.balance = initial ;
  }

  //@ requires amount > 0;
  //@ modifies balance;
  //@ ensures balance == \old(balance) + amount;
  public void deposit(int amount) {
    //@ loop_modifies balance;
    //@ loop_invariant 0 <= i;
    //@ loop_invariant balance == \old(balance) + i;
    //@ decreasing amount - i;
    for (int i = 0; i < amount; i++) {
      balance++;
    }
  }
}
```

3.5.1 Translating the invariant declaration

The translation of the invariant declaration of the `Account` class leads to the generation of a global axiom which defines the invariant in terms of our `inv` function, as illustrated in the following listing:

```
// invariant balance >= 0;
axiom (forall o: ref, h: Store :: inv($Account, o, h) <==>
    isInstanceOf(rval(o), $Account) ==> toint(get(h, fieldLoc(o, Account.balance))) >= 0);
```

3.5.2 Translating `.init`

First of all, we will have a look at the translation of the constructor of the class. The constructor is particularly interesting in terms of the verification of invariants since there are many subtleties behind the correct handling of the verification of invariants in conjunction with the prestate of a constructor and also in conjunction with the invocation of superconstructors. Since those details have been partially omitted during the discussion of the verification methodology for invariants, we want to look at them in more detail at this point. Before that, however, we present the bytecode for the constructor as resulting from the compilation with the Sun Java compiler:

```
public Account(int);
  Code:
    0:   aload_0
    1:   invokespecial    java.lang.Object.<init>()
    4:   aload_0
    5:   iload_1
    6:   putfield         Account.balance
    9:   return
```

Below, the BoogiePL code as generated by our translator is presented. As already mentioned, the more interesting aspects of this part of the code are related to the verification of the invariants on the this object at the beginning of the constructor. The more interesting aspects are thereby mentioned by appropriate comments in the BoogiePL code and will therefore not be further discussed at this point.

```
procedure Account..init.int(param0: ref, param1: int)
{
  var reg0r: ref, reg0i: int;
  var reg1r: ref, reg1i: int;
  var stack0r: ref, stack0i: int;
  var stack1r: ref, stack1i: int;
  var callResultr: ref, callResulti: int;
  var swapr: ref, swapi: int;
  var heap: Store, oldHeap: Store, preHeap: Store;

init:
  // Keep a copy of the old heap to refer to the method's prestate.
  oldHeap := heap;
  // Assume the parameter type information and set up the initial method frame.
  assume param0 != null;
  assume alive(rval(param0), heap);
  assume isOfType(rval(param0), $Account);
  reg0r := param0;
  assume isOfType(ival(param1), $int);
  reg1i := param1;
  // Assume the this object is not aliased.
  assume (forall l: Location :: rval(param0) != get(heap, l));
  // Assume the instance fields of the this objects have their default values.
  assume (forall f: name :: get(heap, fieldLoc(param0, f)) == init(fieldType(f)));
  goto pre;

pre:
  // Assume the invariants on all objects but the this object.
  assume (forall t: name, o: ref :: o != param0 ==> inv(t, o, heap));
  // Assume the method's precondition.
  assume param1 >= 0;
  goto block_2;

block_2:
  stack0r := reg0r;
  // super();
  assert stack0r != null;
  // The this object is not initialized yet, so its invariant need not hold.
  assert (forall o: ref :: o != param0 ==> inv($Account, o, heap));
  assert true; // precondition of the superconstructor
  preHeap := heap;
  havoc heap;
  assume (forall v: Value :: alive(v, preHeap) ==> alive(v, heap));
  // the frame condition of the superconstructor
  assume true ==>
      (forall l: Location :: alive(rval(obj(l)), preHeap)
          && true ==> get(heap, l) == get(preHeap, l));
  // On the this object, we may only assume the invariant of
  // the supertype after a superconstructor call.
  assume (forall t: name, o: ref :: o != param0 ==> inv(t, o, heap));
  assume inv($java.lang.Object, param0, heap);
  assume true ==> true; // postcondition of the superconstructor
  stack0r := reg0r;
  stack1i := reg1i;
  // this.balance = initial;
  assert stack0r != null;
  heap := update(heap, fieldLoc(stack0r, Account.balance), ival(stack1i));
  goto post;

post:
  // Assert the postcondition.
  assert param1 >= 0 ==> toint(get(heap, fieldLoc(param0, Account.balance))) == param1;
  goto exit;

exit:
  // Check the invariants on all instances of the class Account.
  assert (forall o: ref :: inv($Account, o, heap));
  // Assert the frame condition.
  assert param1 >= 0 ==>
      (forall l: Location :: alive(rval(obj(l)), oldHeap)
          && l != fieldLoc(param0, Account.balance) ==> get(heap, l) == get(oldHeap, l));
  return;
}
```

3.5.3 Translating `deposit`

In what follows, we present the translation of the `deposit` method of our example. Again, we wirst of all provide a listing containing the corresponding Java bytecode before proceeding to the discussion of the actual BoogiePL code.

```
public void deposit(int);
  Code:
   0:   iconst_0
   1:   istore_2
   2:   goto    18
   5:   aload_0
   6:   dup
   7:   getfield        Account.balance
   10:  iconst_1
   11:  iadd
   12:  putfield        Account.balance
   15:  iinc    2, 1
   18:  iload_2
   19:  iload_1
   20:  if_icmplt       5
   23:  return
```

In the code below, we mainly see how loop specifications are translated to BoogiePL. Again, the relevant parts are commented inline in the code and, thus, will not be discussed in further detail at this point.

```
procedure Account.deposit.int(param0: ref, param1: int)
{
  var reg0r: ref, reg0i: int;
  var reg1r: ref, reg1i: int;
  var reg2r: ref, reg2i: int;
  var stack0r: ref, stack0i: int;
  var stack1r: ref, stack1i: int;
  var stack2r: ref, stack2i: int;
  var callResultr: ref, callResulti: int;
  var swapr: ref, swapi: int;
  var heap: Store, oldHeap: Store, preHeap: Store;
  var loopHeap0: Store;
  var loopVariant0: int;

init:
  // Keep a copy of the old heap to refer to the method's prestate.
  oldHeap := heap;
  // Assume the appropriate parameter type information and set up the
  // initial method frame.
  assume param0 != null;
  assume alive(rval(param0), heap);
  assume isOfType(rval(param0), $Account);
  reg0r := param0;
  assume isOfType(ival(param1), $int);
  reg1i := param1;
  goto pre;

pre:
  // Assume the invariants of all objects.
  assume (forall t: name, o: ref :: inv(t, o, heap));
  // Assume the method's precondition.
  assume param1 > 0;
  goto block_2;

block_2:
  stack0i := 0;
  reg2i := stack0i;
  // Make a copy of the heap before entering the loop.
  loopHeap0 := heap;
  goto block_4_Loop;

block_3:
  stack0r := reg0r;
  stack1r := stack0r;
  assert stack1r != null;
  stack1i := toint(get(heap, fieldLoc(stack1r, Account.balance)));
  stack2i := 1;
  stack1i := stack1i + stack2i;
  assert stack0r != null;
  heap := update(heap, fieldLoc(stack0r, Account.balance), ival(stack1i));
  reg2i := reg2i + 1;
  // Check that the value of the loop variant function has decreased at least
  // by 1 during the current loop iteration.
  assert reg1i − reg2i < loopVariant0;
  goto block_4_Loop;

// Here, the loop starts.
```

```
block_4_Loop:
  // Assume the type information of the current method frame at the beginning
  // of the loop.
  assume isOfType(rval(reg0r), $Account);
  assume isOfType(ival(reg1i), $int);
  assume isOfType(ival(reg2i), $int);
  // Assume that objects remain alive inside the loop.
  assume (forall v: Value :: alive (v, loopHeap0) ==> alive(v, heap));
  // Ensure that the objects allocated inside the loop satisfy their invariants
  // at each loop iteration .
  assert (forall o: ref :: ! alive (rval (o), loopHeap0) ==> inv($Account, o, heap));
  // Check the actual loop invariant as specified in the source class .
  assert 0 <= reg2i
       && toint(get(heap, fieldLoc(param0, Account.balance)))
       == toint(get(oldHeap, fieldLoc(param0, Account.balance))) + reg2i;
  // Check for the non−negativity of the loop variant function.
  assert 0 <= reg1i − reg2i;
  // Check the loop's frame condition.
  assert (forall l: Location :: alive (rval (obj(l )), loopHeap0)
       && l != fieldLoc(param0, Account.balance) ==> get(heap, l) == get(loopHeap0, l));
  // Keep a copy of the value of the loop variant function at the beginning of
  // a loop iteration .
  loopVariant0 := reg1i − reg2i;
  stack0i := reg2i;
  stack1i := reg1i;
  goto block_4_Loop_True, block_4_Loop_False;

block_4_Loop_True:
  assume stack0i < stack1i;
  goto block_3;

block_4_Loop_False:
  // Break out of the loop.
  assume stack0i >= stack1i;
  goto block_5;

block_5:
  goto post;

post:
  // Assert the method's postcondition.
  assert param1 > 0 ==>
       toint (get(heap, fieldLoc (param0, Account.balance)))
       == toint(get(oldHeap, fieldLoc(param0, Account.balance))) + param1;
  goto exit;

exit:
  // Check that the appropriate invariants hold.
  assert (forall o: ref :: inv($Account, o, heap));
  // Check that the method frame condition holds.
  assert param1 > 0 ==>
       (forall l: Location :: alive (rval (obj(l )), oldHeap)
            && l != fieldLoc(param0, Account.balance) ==> get(heap, l) == get(oldHeap, l));
  return;
```

```
}
```

Chapter 4

Implementation

In this chapter, we present the implementation of a new tool for translating BML annotated Java bytecode to BoogiePL, which features the translation on which this work builds [19] as well as all the contributions and extensions presented in the previous chapters of this document.

Note that apart from the here provided description of the implementation, we also provide an extensive JavaDoc which accompanies the code itself and which is thought to be the main source of information when a detailed understanding of the implementation is required.

4.1 Bytecode classes and BML specifications

In this section, we describe the parts of the translator which are responsible for reading and representing bytecode classes. In particular, we will first present the bytecode library which we are using for the more low-level aspects of bytecode handling while also clearly specifying to which extent our own code depends on the library. Other aspects such as the dataflow analysis performed on the bytecode will also be covered.

4.1.1 The ASM bytecode library

For the actual parsing of a class file, we use the ASM bytecode library [1]. While other popular libraries such as BCEL [2] are also available, we have opted for ASM, mainly based on the following criteria:

- ASM has a simple, well designed and modular API which is intuitive to use. This mainly comes from the fact that, unlike other bytecode libraries such BCEL, ASM provides an API which abstracts away the constant pool of a class file. This is achieved by only ever passing the *content* of the constant pool to the user instead of the corresponding constant pool index which must then be painfully maintained by the user to retrieve the actual data.

- The core API of the ASM bytecode library is based on a clean, visitor-like API which requires no in-memory representation of a class file. As we will see later, we use our own abstract syntax tree representation of a class file and, thus, we found it beneficial to avoid building up different in-memory representations of the same class file.

- ASM features a dataflow analysis framework which we use for computing the method frames for the individual bytecode instructions which are then used for the actual translation process.

- Unlike BCEL, ASM is actively maintained and it provides support for the latest Java version, Java 6. Furthermore, it is open source and released under the liberal BSD license.

4.1.2 BML class file attributes

ASM provides a simple mechanism for accessing the contents of custom class file attributes by declaring the class `Attribute` which should be extended by every class representing a custom class file attribute provided by the user. A *prototype instance* of the extending class can then be passed to the class reader of ASM which will be used to read the attribute's raw bytecode data when an appropriate attribute is encountered in the class file being parsed. The newly generated attribute is then passed to the user through the usual event based API.

Based on this mechanism, we define a separate class for each of the BML class file attributes recognized by our translator. Every such class thereby contains the code specific to the actual format of the class file attribute which is responsible for decoding the attribute stored in the class file and creating a new instance of the attribute class representing it. All the code which is independent of a concrete BML attribute is shared in the `BMLAttributeReader` class. All the classes representing the individual BML attributes or otherwise related to the parsing of those attributes can be found in the `b2bpl.bytecode.attributes` package.

4.1.3 Class file loading

Our translator uses its own representation for a class file and its members. A class file itself is thereby represented by the class `JClassType`. For actually creating an instance of that class in the context of our translator, the API of the `TypeLoader` class should always be used which internally maintains a repository of such class file objects in order to avoid that referenced classes are loaded multiple times. In addition to maintaining such a repository, the `TypeLoader` provides several high-level mechanisms which allow for an effective optimization of the class file loading process:

- The `TypeLoader` has a notion of so-called *application types* which are assumed to represent the set of types which will be translated to BoogiePL and not just referenced by other class types. This information can be used to heavily optimize the process of loading a class file by skipping the actual bytecode instructions in class files which will never be translated to BoogiePL. For the latter, only the interface including the BML specifications will ever be loaded. Note that this optimization not only considerably reduces the amount of time required to load a class file which is not part of the set of application types but it also has an important impact on the set of classes which are ever referenced and, thus, eventually loaded.

- The `TypeLoader` offers the possibility to not directly load a class when it is first referenced but only when some information on that class is accessed which indeed requires the class file data to be loaded. This lazy loading mechanism is implemented by requiring that the `JClassType` itself checks whether the data which is queried on it is already available and, if not, it requests it from the `TypeLoader`. This mechanism is thereby totally transparent to classes other than the `TypeLoader` and `JClassType` classes themselves.

Since the lazy loading mechanism implemented by the `TypeLoader` class makes it impossible to cleanly separate the phase of class file loading from any semantic analysis and from the actual translation, we require that every class type returned by the type loader should have passed the semantic analysis in order to ensure that no unexpected errors occur during the translation process due to some inconsistency in the class file. Therefore, the `TypeLoader` class is immediately responsible for performing the semantic analysis on every loaded class type by delegating the actual analysis to the `SemanticAnalyzer` class.

4.1.4 Flow analysis of bytecode methods

The actual dataflow analysis on bytecode methods is performed by the `FlowAnalyzer` class which in turn delegates part of its work to the dataflow analysis framework provided by the ASM bytecode library. As a result of the dataflow analysis, every bytecode instruction will have a valid

`StackFrame` associated to it which represents the method frame containing the types of the local registers and elements of the operand stack at the given instruction.

4.2 Translation to BoogiePL

The actual translation from Java bytecode to BoogiePL is split over different classes for better modularity. In the following, we will describe which classes are involved in the generation of the resulting BoogiePL program and how they interact with each other.

4.2.1 Classes

The main entry point to the translation of a set of bytecode classes to BoogiePL is the `Translator` class. Some aspects of the translation process can be configured by passing an appropriate instance of the class `Project` containing the desired translation settings upon creating the `Translator`. The `Translator` class is immediately responsible for generating the following parts of the resulting BoogiePL program:

- The part of the background predicate which is the same for every translation. This mainly includes the heap and core type system axiomatization.
- The global theory part which depends on the concrete set of bytecode classes being translated. Among others, this includes the set of axioms which are for example generated when a given class type is refernce and which define the type's subtyping relationships.

By contrast, the following aspects of the translation are not handled directly by the `Translator` class but, instead, are delegated to other classes:

- The BML specifications coming from the declaration of invariants are translated by an instance of the class `SpecificationTranslator`.
- The individual bytecode methods are translated by a `MethodTranslator`.

Since BML specifications and bytecode methods may contain type references and other references which may require global axioms to be generated, we introduce the special interface `TranslationContext` which declares methods for translating special references which cannot be resolved locally. Such a context interface can then be used to delegate the translation of those reference to the `Translator` which is responsible for the global section of the BoogiePL program being generated.

4.2.2 Methods

The main entry point to the translation of a bytecode method to a BoogiePL procedure is the `MethodTranslator` class. The `MethodTranslator` is responsible for the following aspects of the translation process:

- The translation of the individual bytecode instructions and the method's program flow.
- The translation of local BML annotations such as assertions, assumptions, and loop specifications.
- The translation of method specifications and invariants.

The actual translation of BML specification expressions and store references is thereby not performed directly by the `MethodTranslator` but, instead, is delegated to the `SpecificationTranslator` and `ModifiesFilter` classes, respectively.

4.2.3 BML specifications

The `SpecificationTranslator` class is responsible for translating BML specification expressions to expressions in BoogiePL. Internally, it is realized as an implementation of the `BMLExpression` interface. Likewise, the `ModifiesFilter` class is responsible for translating BML store references contained inside modifies clauses to a predicate expression in BoogiePL which is `true` if and only if a given location does not refer to any of the specified store references.

Chapter 5

Conclusion and future work

5.1 Conclusion

In this report, we have presented several extensions to an already existing translation of Java bytecode to BoogiePL, the input language of the Boogie program verifier. Notable contributions include the extension of the set of bytecode instructions supported by the translation and an axiomatization for the JVM type system. In addition, the translation to BoogiePL now also supports a set of BML specifications such as object invariants as well as method and loop specifications. Furthermore, an implementation of the existing translation and all our contributions has been provided as part of this work.

5.2 Future work

5.2.1 History constraints

History constraints represent an interesting specification feature which allows to relate the current state of an object to a state as observed earlier in the program flow and, thus, are an interesting generalization of invariants. Since BML already provides support for history constraints, we regard it as a natural future step to support them in our translation, too.

5.2.2 Loop invariant inference

Abstract interpretation [7] allows for an automatic inference of many loop invariants which are required for the program being successfully verified without. For that reason, it would be interesting to see how an abstract interpretation framework could be integrated into our translator and how the two components would interact with each other.

Appendix A

Supported bytecode instructions

In the following, we shall define the set of bytecode instructions supported by the current translation. For every instruction, we specify the set of runtime exceptions which are modeled in the translation as well as the Java class by which the instruction is represented in the actual implementation. The opcode notation employed as well as the below categorization of the instruction set into groups of instructions follows the conventions of the JVM specification [13].

Load and store instructions

Opcodes	**Runtime exceptions**	**Java class**
`iload, iload_<n>`	-	`ILoadInstruction`
`lload, lload_<n>`	-	`LLoadInstruction`
`aload, aload_<n>`	-	`ALoadInstruction`
`istore, istore_<n>`	-	`IStoreInstruction`
`lstore, lstore_<n>`	-	`LStoreInstruction`
`astore, astore_<n>`	-	`AStoreInstruction`
`bipush, sipush, iconst_m1, iconst_<i>, lconst_<l>`	-	`VConstantInstruction`
`ldc, ldc_w, ldc2_w`	-	`LdcInstruction`
`aconst_null`	-	`AConstNullInstruction`
`wide`	-	-[1]

Arithmetic instructions

Opcodes	**Runtime exceptions**	**Java class**
`iadd, isub, imul, idiv, irem`	`ArithmeticException`[2]	`IBinArithInstruction`
`ladd, lsub, lmul, ldiv, lrem`	`ArithmeticException`[2]	`LBinArithInstruction`
`ineg`	-	`INegInstruction`
`lneg`	-	`LNegInstruction`
`ishl, ishr, iushr, iand, ior, ixor`	-	`IBitwiseInstruction`
`lshl, lshr, lushr, land, lor, lxor`	-	`LBitwiseInstruction`
`iinc`	-	`IIncInstruction`
`lcmp`	-	`LCmpInstruction`

[1]The `wide` instruction only serves as a *modifier* for other instructions and, as such, has no explicit representation.
[2]for the division and remainder operations only

Type conversion instructions

Opcodes	Runtime exceptions	Java class
i2s, i2b, i2c, i2l, l2i	-	VCastInstruction

Object creation and manipulation

Opcodes	Runtime exceptions	Java class
new	-	NewInstruction
newarray	NegativeArraySizeException	NewArrayInstruction
anewarray	NegativeArraySizeException	ANewArrayInstruction
multianewarray	NegativeArraySizeException	MultiANewArrayInstruction
getfield	NullPointerException	GetFieldInstruction
putfield	NullPointerException	PutFieldInstruction
getstatic	-	GetStaticInstruction
putstatic	-	PutStaticInstruction
baload, caload, saload, iaload, laload	NullPointerException, ArrayIndexOutOfBoundsException	VALoadInstruction
aaload	NullPointerException, ArrayIndexOutOfBoundsException	AALoadInstruction
bastore, castore, sastore, iastore, lastore	NullPointerException, ArrayIndexOutOfBoundsException	VAStoreInstruction
aastore	NullPointerException, ArrayIndexOutOfBoundsException, ArrayStoreException	AAStoreInstruction
arraylength	NullPointerException	ArrayLengthInstruction
instanceof	-	InstanceOfInstruction
checkcast	ClassCastException	CheckCastInstruction

Operand stack management instructions

Opcodes	Runtime exceptions	Java class
pop	-	PopInstruction
pop2	-	Pop2Instruction
dup	-	DupInstruction
dup2	-	Dup2Instruction
dup_x1	-	DupX1Instruction
dup2_x1	-	Dup2X1Instruction
dup_x2	-	DupX2Instruction
dup2_x2	-	Dup2X2Instruction
swap	-	SwapInstruction

Control transfer instructions

Opcodes	Runtime exceptions	Java class
ifeq, iflt, ifle, ifne, ifgt, ifge	-	IfInstruction
ifnull	-	IfNullInstruction
ifnonnull	-	IfNonNullInstruction
if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge	-	IfICmpInstruction
if_acmpeq, if_acmpne	-	IfACmpInstruction
tableswitch	-	TableSwitchInstruction
lookupswitch	-	LookupSwitchInstruction
goto, goto_w	-	GotoInstruction

Method invocation and return instructions

Opcodes	Runtime exceptions	Java class
invokevirtual	NullPointerException	InvokeVirtualInstruction
invokeinterface	NullPointerException	InvokeInterfaceInstruction
invokespecial	NullPointerException	InvokeSpecialInstruction
invokestatic	-	InvokeStaticInstruction
invokestatic	-	InvokeStaticInstruction
ireturn	-	IReturnInstruction
lreturn	-	LReturnInstruction
areturn	-	AReturnInstruction
return	-	ReturnInstruction

Throwing exceptions

Opcodes	Runtime exceptions	Java class
athrow	NullPointerException	AThrowInstruction

Appendix B

BML Abstract Syntax Tree

In the following, we specify a BNF-like grammar definition which describes the abstract syntax tree for BML specifications as implemented as part of our work. The grammar definition shall thereby serve as a precise and compact description of the concrete implementation of the abstract syntax tree without referring to the actual code. This is achieved since every non-terminal symbol used in the grammar directly corresponds to an equally-named[1] class in the actual implementation. More precisely, the grammar is to be interpreted as follows:

- We use the meta-level symbols *, $^+$, and $^?$ to denote a sequence, a non-empty sequence, and an optional element, respectively. The angle brackets $\langle\cdot\rangle$ are used for grouping and the standard operator | separates different alternatives in a single production.
- All the terminal symbols denoting either keywords in BML or punctuation symbols are written in a **bold** font.
- Non-terminal symbols denoting *abstract* classes in the implementation of the abstract syntax tree are written in an *italic* font. Every production for such an abstract class consists of a set of alternatives representing all the subclasses of the abstract class.
- Non-terminal symbols denoting *concrete* classes in the implementation of the AST are written in a normal font. Every production for such a concrete class may consist of a set of terminal and non-terminal symbols, where the latter fully describe the corresponding concrete class.

Type specifications

$$\text{Invariant} \quad ::= \quad \textbf{invariant}\ \text{Predicate}\ \textbf{;}$$

Method specifications

$$\begin{array}{rcl}
\text{MethodSpecification} & ::= & \text{RequiresClause SpecificationCase}^* \\
\text{SpecificationCase} & ::= & \text{RequiresClause ModifiesClause EnsuresClause ExsuresClause}^* \\
\text{RequiresClause} & ::= & \textbf{requires}\ \text{Predicate}\ \textbf{;} \\
\text{ModifiesClause} & ::= & \textbf{modifies}\ \textit{StoreRef}^*\ \textbf{;} \\
\text{EnsuresClause} & ::= & \textbf{ensures}\ \text{Predicate}\ \textbf{;} \\
\text{ExsuresClause} & ::= & \textbf{exsures}\ \textbf{(}\textit{JType}\ \textbf{e}\textbf{)}\ \text{Predicate}\ \textbf{;}
\end{array}$$

Loop specifications

[1]up to the prefix BML which is prepended to every node class of the abstract syntax tree in the actual implementation while omitted in the grammar and in the following discussion

LoopSpecification	::=	LoopModifiesClause LoopInvariant LoopVariant
LoopModifiesClause	::=	**loop_modifies** *StoreRef** **;**
LoopInvariant	::=	**loop_invariant** Predicate **;**
LoopVariant	::=	**decreasing** *Expression* **;**

Assertions and assumptions

AssertStatement	::=	**assert** Predicate **;**
AssumeStatement	::=	**assume** Predicate **;**

Specification expressions

Expression	::=	*BinaryExpression*
	$\mid$	*UnaryExpression*
	$\mid$	ArrayAccessExpression
	$\mid$	ArrayLengthExpression
	$\mid$	BoundVariableExpression
	$\mid$	CastExpression
	$\mid$	ElemTypeExpression
	$\mid$	FieldAccessExpression
	$\mid$	FieldExpression
	$\mid$	FreshExpression
	$\mid$	InstanceOfExpression
	$\mid$	*Literal*
	$\mid$	LocalVariableExpression
	$\mid$	OldExpression
	$\mid$	Predicate
	$\mid$	QuantifierExpression
	$\mid$	ResultExpression
	$\mid$	StackCounterExpression
	$\mid$	StackElementExpression
	$\mid$	ThisExpression
	$\mid$	TypeOfExpression
BinaryExpression	::=	BinaryArithmeticExpression
	$\mid$	BinaryBitwiseExpression
	$\mid$	BinaryLogicalExpression
	$\mid$	EqualityExpression
	$\mid$	RelationalExpression
Literal	::=	BooleanLiteral
	$\mid$	IntLiteral
	$\mid$	NullLiteral
UnaryExpression	::=	LogicalNotExpression
	$\mid$	UnaryMinusExpression

BinaryArithmeticExpression	::=	*Expression* $\langle + \mid - \mid * \mid / \mid \% \rangle$ *Expression*
BinaryBitwiseExpression	::=	*Expression* $\langle << \mid >> \mid >>> \mid \& \mid \vert \mid \hat{} \rangle$ *Expression*
BinaryLogicalExpression	::=	*Expression* $\langle \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \rangle$ *Expression*
EqualityExpression	::=	*Expression* $\langle = \mid \neq \rangle$ *Expression*
RelationalExpression	::=	*Expression* $\langle < \mid > \mid \leq \mid \geq \rangle$ *Expression*
LogicalNotExpression	::=	$\neg$ *Expression*
UnaryMinusExpression	::=	$-$ *Expression*
QuantifierExpression	::=	$(\langle \forall \mid \exists \rangle$ *JType** $\bullet$ *Expression*$)$

Store references

$$
\begin{array}{rcl}
\textit{StoreRef} & ::= & \text{EverythingStoreRef} \\
 & | & \text{NothingStoreRef} \\
 & | & \textit{StoreRefExpression}
\end{array}
$$

$$
\begin{array}{rcl}
\text{EverythingStoreRef} & ::= & \backslash\textbf{everything} \\
\text{NothingStoreRef} & ::= & \backslash\textbf{nothing}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{StoreRefExpression} & ::= & \text{ArrayAllStoreRef} \\
 & | & \text{ArrayElementStoreRef} \\
 & | & \text{ArrayRangeStoreRef} \\
 & | & \text{FieldStoreRef} \\
 & | & \text{FieldAccessStoreRef} \\
 & | & \text{LocalVariableStoreRef} \\
 & | & \text{ThisStoreRef}
\end{array}
$$

Appendix C

BoogiePL Abstract Syntax Tree

In the following, we will give a brief description of the abstract syntax tree used to represent a BoogiePL program. To that end, we use a BNF-like grammar definition which describes the abstract syntax of a BoogiePL program. Note that while similar grammars are also given elsewhere [10, 4], the main purpose of the definition provided here is to clearly specify the features of BoogiePL supported by our implementation. In addition, the grammar definition shall serve as a precise and compact description of the concrete implementation of the abstract syntax tree without referring to the actual code. This is achieved since every non-terminal symbol used in the grammar directly corresponds to an equally-named[1] class in the actual implementation. More precisely, the grammar is to be interpreted as follows:

- We use the meta-level symbols *, $^+$, and $^?$ to denote a sequence, a non-empty sequence, and an optional element, respectively. The angle brackets $\langle\cdot\rangle$ are used for grouping and the standard operator | separates different alternatives in a single production.
- All the terminal symbols denoting either keywords of the BoogiePL language or punctuation symbols are written in a **bold** font.
- Non-terminal symbols denoting *abstract* classes in the implementation of the abstract syntax tree are written in an *italic* font. Every production for such an abstract class consists of a set of alternatives representing all the subclasses of the abstract class.
- Non-terminal symbols denoting *concrete* classes in the implementation of the abstract syntax tree are written in a normal font. Every production for such a concrete class may consist of a set of terminal and non-terminal symbols, where the latter fully describe the corresponding concrete class.

Programs and declarations

A BoogiePL program consists of a set of declarations:

$$\begin{array}{rcl} \text{Program} & ::= & \textit{Declaration}^* \\ \textit{Declaration} & ::= & \text{VariableDeclaration} \\ & | & \text{ConstantDeclaration} \\ & | & \text{TypeDeclaration} \\ & | & \text{Function} \\ & | & \text{Axiom} \\ & | & \text{Procedure} \\ & | & \text{Implementation} \end{array}$$

[1] up to the prefix BPL which is prepended to every node class of the abstract syntax tree in the actual implementation while omitted in the grammar and in the following discussion

Variable and constant declarations are treated uniformally in that they both introduce a set of `Variables`:

$$\begin{array}{rcl}
\text{VariableDeclaration} & ::= & \textbf{var}\ \text{Variable}^{+}\ \textbf{;} \\
\text{ConstantDeclaration} & ::= & \textbf{const}\ \text{Variable}^{+}\ \textbf{;}
\end{array}$$

A type declaration can be used in BoogiePL to declare a set of user-defined types:

$$\begin{array}{rcl}
\text{TypeDeclaration} & ::= & \textbf{type}\ \text{String}^{+}\ \textbf{;}
\end{array}$$

Since such a declaration merely defines the names of the introduced types but does not contain any further information, we represent the individual types as normal `String`s instead of introducing a special purpose class for them.

Every function declaration defines exactly one function symbol (which, however, may have several names associated to it). Therefore, we do not make any special distinction between the declaration and the actual function symbol which can then be referenced in expressions:

$$\begin{array}{rcl}
\text{Function} & ::= & \textbf{function}\ \text{String}^{+}\textbf{(}\text{FunctionParameter}^{*}\textbf{)} \\
 & & \quad \textbf{returns}\ \textbf{(}\text{FunctionParameter}\textbf{)} \\
\text{FunctionParameter} & ::= & \langle \text{String}\ \textbf{:}\rangle^{?}\ \textit{Type}
\end{array}$$

In- and out-parameters to a function are represented by the special class `FunctionParameter` which simply represents an optionally named type.

An axiom is fully defined by an expression of type **bool** which specifies a constraint on the symbolic constants and functions:

$$\begin{array}{rcl}
\text{Axiom} & ::= & \textbf{axiom}\ \textit{Expression}\ \textbf{;}
\end{array}$$

Procedures and implementations mainly differ from each other in that a procedure may have a specification attached to it:

$$\begin{array}{rcl}
\text{Procedure} & ::= & \textbf{procedure}\ \text{String}\textbf{(}\text{Variable}^{*}\textbf{)}\ \langle \textbf{returns}\ \textbf{(}\text{Variable}^{*}\textbf{)}\rangle^{?} \\
 & & \quad \text{Specification}^{?}\ \text{ImplementationBody}^{?} \\
\text{Implementation} & ::= & \textbf{implementation}\ \text{String}\textbf{(}\text{Variable}^{*}\textbf{)}\ \langle \textbf{returns}\ \textbf{(}\text{Variable}^{*}\textbf{)}\rangle^{?} \\
 & & \quad \text{ImplementationBody}
\end{array}$$

The in- and out-parameters are in both cases represented by a set of `Variables`. A procedure's specification consists of a number of **requires**, **modifies**, and **ensures** clauses, where a new feature recently added to BoogiePL allows requires and ensures clauses to be marked as **free** in which case the corresponding conditions need not be verified by Boogie but instead can just be assumed. A modifies clause is defined by an optional set of `VariableExpressions` (discussed later) which represent identifiers referencing variables:

$$\begin{array}{rcl}
\text{Specification} & ::= & \textit{SpecificationClause}^{+} \\
\textit{SpecificationClause} & ::= & \text{RequiresClause} \\
 & | & \text{ModifiesClause} \\
 & | & \text{EnsuresClause} \\
\text{RequiresClause} & ::= & \textbf{free}^{?}\ \textbf{requires}\ \textit{Expression}\ \textbf{;} \\
\text{ModifiesClause} & ::= & \textbf{modifies}\ \text{VariableExpression}^{*}\ \textbf{;} \\
\text{EnsuresClause} & ::= & \textbf{free}^{?}\ \textbf{ensures}\ \textit{Expression}\ \textbf{;}
\end{array}$$

Finally, the implementation body belonging to a procedure or an implementation consists of an optional set of variable declarations followed by a number of basic blocks:

$$\begin{array}{rcl}
\text{ImplementationBody} & ::= & \text{VariableDeclaration}^{*}\ \text{BasicBlock}^{+}
\end{array}$$

Types

Types are mainly used in the declarations of variables and function parameters but also in cast expressions. Currently, four kinds of types are supported in BoogiePL:

$$\begin{array}{rcl} \textit{Type} & ::= & \text{BuiltInType} \\ & | & \text{TypeName} \\ & | & \text{ArrayType} \\ & | & \text{ParameterizedType} \end{array}$$

The built-in types are represented by the single class `BuiltInType`. In the actual implementation, we use a typesafe enumeration pattern to represent the individual types:

$$\text{BuiltInType} \quad ::= \quad \textbf{bool} \mid \textbf{int} \mid \textbf{ref} \mid \textbf{name} \mid \textbf{any}$$

User-defined types are represented by the `TypeName` class and are fully determined by their name which corresponds to a BoogiePL identifier:

$$\text{TypeName} \quad ::= \quad \text{String}$$

Array types are defined by a set of index types used to access the array together with the actual element type:

$$\text{ArrayType} \quad ::= \quad [\textit{Type}^{+}]\ \textit{Type}$$

Note that even though our grammar allows array types to have an arbitrary number of index types, BoogiePL only supports up to two-dimensional arrays.

A new feature recently added to BoogiePL is the support for what we call *parameterized types* which allows to parameterize some type by another type:

$$\text{ParameterizedType} \quad ::= \quad <\textit{Type}>\ \textit{Type}$$

In order to see what parameterized types can be used for, let us assume we want to model a heap in BoogiePL by using a two-dimensional array which is indexed by a **ref** type denoting an object and a **name** type representing an object's field (as is e.g. done in Spec#). Since the heap may contain objects of different types, the array's element type must be declared to be of type **any**. Therefore, whenever we extract an element from such an array, we must usually insert an appropriate cast to the actual type of the field. This is illustrated on the left hand side of the following listing:

```
var Heap: [ref, name]any;
const C.f: name;
var o: ref, i: int;
Heap[o, C.f] := 3;
i := cast(Heap[o, C.f], int);
```

```
var Heap: [ref, <t>name]t;
const C.f: <int>name;
var o: ref, i: int;
Heap[o, C.f] := 3;
i := Heap[o, C.f];
```

On the right hand side, by contrast, we see how one might achieve the same result by taking advantage of parameterized types: we see that the **name** type used to index the heap array is now parameterized by a type parameter `t` which is also used as the array's element type. As can be seen in the declaration of the constant `C.f` denoting the field being accessed, the actual type to be inserted for the type parameter `t` can then be specified individually for every **name** constant. If we now use the constant `C.f` to access an element of the array, the type **int** is automatically substituted for the type parameter `t` and no explicit cast is required anymore, thus improving the readability of the code. In addition – and more importantly – the fact that the field modeled by the constant `C.f` is of type **int** can be made explicit in the constant's declaration by using parameterized types.

Variables

We use the single class `Variable` to represent variables and constants, in- and out-parameters of procedures and implementations, as well as expression bound variables introduced by quantification expressions (discussed later). Therefore, a `Variable` covers exactly those elements which may be referenced by a BoogiePL identifier in expressions and procedure specifications. Our grammar defines a variable as follows:

$$\text{Variable} \quad ::= \quad \text{String } \mathbf{:} \ \textit{Type} \ \langle \mathbf{where} \ \textit{Expression} \rangle^{?}$$

As we can see, a variable can have a so-called *where clause* associated to it. The expression of such a where clause must be of type **bool** and it specifies a unary constraint on the variable's value. Where clauses are a convenience construct which allows to specify some properties of a variable's value along with its declaration instead of scattering that information over different points in the program. In addition, whenever such a variable is havoc'ed, its value becomes not totally arbitrary but is still constrained by the expression provided in the where clause. However, note that where clauses may only be used in variable declarations and for parameters of procedures and implementations but not in constant declarations or for expression bound variables. Note also that the concept of a variable as used in our context does not completely correspond to the notion of variables described in [10] where constant symbols are not considered to be variables.

Basic blocks and commands

The body of a procedure or implementation contains a set of basic blocks, where each of them consists of a label and a sequence of commands, followed by a single transfer command:

$$\text{BasicBlock} \quad ::= \quad \text{String}\mathbf{:} \ \textit{Command}^{*} \ \textit{TransferCommand}$$

The set of commands and transfer commands should be self-explanatory and, thus, they are not discussed further at this point:

$$\begin{array}{rcl}
\textit{Command} & ::= & \text{AssertCommand} \\
 & \mid & \text{AssertCommand} \\
 & \mid & \text{HavocCommand} \\
 & \mid & \text{AssignmentCommand} \\
 & \mid & \text{CallCommand} \\
\text{AssertCommand} & ::= & \mathbf{assert} \ \textit{Expression} \ \mathbf{;} \\
\text{AssumeCommand} & ::= & \mathbf{assume} \ \textit{Expression} \ \mathbf{;} \\
\text{HavocCommand} & ::= & \mathbf{havoc} \ \text{VariableExpression}^{+} \ \mathbf{;} \\
\text{AssignmentCommand} & ::= & \textit{Expression} \ \mathbf{:=} \ \textit{Expression} \ \mathbf{;} \\
\text{CallCommand} & ::= & \mathbf{call} \ \langle \text{VariableExpression}^{+} \ \mathbf{:=} \rangle^{?} \ \text{String}\mathbf{(}\textit{Expression}^{*}\mathbf{)} \ \mathbf{;} \\
\textit{TransferCommand} & ::= & \text{GotoCommand} \\
 & \mid & \text{ReturnCommand} \\
\text{GotoCommand} & ::= & \mathbf{goto} \ \text{String}^{+} \ \mathbf{;} \\
\text{ReturnCommand} & ::= & \mathbf{return;}
\end{array}$$

Expressions

All the classes used to represent the different kinds of expressions are given in the following grammar productions:

$$\begin{array}{rcl}
\textit{Expression} & ::= & \text{VariableExpression} \\
 & \mid & \textit{BinaryExpression} \\
 & \mid & \textit{UnaryExpression}
\end{array}$$

$$\begin{array}{rcl}
 & \mid & \text{QuantifierExpression} \\
 & \mid & \textit{Literal} \\
 & \mid & \text{ArrayExpression} \\
 & \mid & \text{CastExpression} \\
 & \mid & \text{FunctionApplication} \\
 & \mid & \text{OldExpression} \\
\textit{BinaryExpression} & ::= & \text{BinaryArithmeticExpression} \\
 & \mid & \text{BinaryLogicalExpression} \\
 & \mid & \text{EqualityExpression} \\
 & \mid & \text{PartialOrderExpression} \\
 & \mid & \text{RelationalExpression} \\
\textit{Literal} & ::= & \text{BoolLiteral} \\
 & \mid & \text{NullLiteral} \\
 & \mid & \text{IntLiteral} \\
\textit{UnaryExpression} & ::= & \text{LogicalNotExpression} \\
 & \mid & \text{UnaryMinusExpression}
\end{array}$$

A `VariableExpression` is the most common expression and simply represents a BoogiePL identifier referencing a variable (as modeled by the already discussed `Variable` class):

$$\text{VariableExpression} \quad ::= \quad \text{String}$$

The set of arithmetic and first-order logical expressions supported by BoogiePL are defined by the following grammar productions:

$$\begin{array}{rcl}
\text{BinaryArithmeticExpression} & ::= & \textit{Expression} \ \langle + \mid - \mid * \mid / \mid \% \rangle \ \textit{Expression} \\
\text{BinaryLogicalExpression} & ::= & \textit{Expression} \ \langle \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \rangle \ \textit{Expression} \\
\text{EqualityExpression} & ::= & \textit{Expression} \ \langle = \mid \neq \rangle \ \textit{Expression} \\
\text{PartialOrderExpression} & ::= & \textit{Expression} <: \textit{Expression} \\
\text{RelationalExpression} & ::= & \textit{Expression} \ \langle < \mid > \mid \leq \mid \geq \rangle \ \textit{Expression} \\
\text{LogicalNotExpression} & ::= & \neg \ \textit{Expression} \\
\text{UnaryMinusExpression} & ::= & - \ \textit{Expression} \\
\text{QuantifierExpression} & ::= & \mathbf{(} \langle \ \forall \mid \exists \ \rangle \ \text{Variable}^{*} \ \text{Trigger}^{*} \bullet \textit{Expression} \mathbf{)}
\end{array}$$

As we can see, in order to reduce the number of nodes in the abstract syntax tree, not every expression is represented by its own class but, instead, we group operations according to their operands and types. In the actual implementation, a special enumeration type denoting the individual operators is introduced for every group of operations in order to distinguish among them.

Triggers 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, as described in [4]. As specified above, triggers can be used in quantification expressions and consist of a non-empty sequence of expressions:

$$\text{Trigger} \quad ::= \quad \mathbf{\{} \ \textit{Expression}^{+} \ \mathbf{\}}$$

The boolean, reference, and integer literals supported by BoogiePL are represented as follows:

$$\begin{array}{rcl}
\text{BoolLiteral} & ::= & \mathbf{true} \mid \mathbf{false} \\
\text{NullLiteral} & ::= & \mathbf{null} \\
\text{IntLiteral} & ::= & \cdots \mid \mathbf{-1} \mid \mathbf{0} \mid \mathbf{1} \mid \cdots
\end{array}$$

The remaining expressions, which should be self-explanatory and, thus, are not further discussed at this point, are the following:

$$\text{ArrayExpression} \quad ::= \quad \textit{Expression}[\textit{Expression}^{+}]$$

CastExpression	::=	**cast**(*Expression*, *Type*)
FunctionApplication	::=	String(*Expression**)
OldExpression	::=	**old**(*Expression*)

Implementation notes

The actual implementation of the abstract syntax tree is fully described by the above discussed grammar. The individual nodes of the tree are implemented as simple classes which do not contain any specific functionality. Instead, operations on the abstract syntax tree are implemented using the visitor pattern. In particular, the implementation of the abstract syntax tree is completely self-contained in that the individual nodes of the tree only reference other nodes and the abstract visitor provided for the tree.

Decorating the abstract syntax tree

As a convenience, we provide the ability to decorate some of the nodes with node-specific data. As an example, the `Variable` referenced by a `VariableExpression` can be directly stored in the latter and the types to which expressions evaluate can be set on the corresponding `Expression` object. This allows to conveniently store information which might be used frequently in the nodes themselves. Note, however, that even though this information is kept in the nodes, the information itself is never computed by the nodes but is always set from the outside (e.g. by an appropriate visitor performing a semantic analysis on the tree).

Bibliography

[1] The ASM bytecode library. http://asm.objectweb.org/.

[2] The ASM bytecode library. http://jakarta.apache.org/bcel/.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111, pages 364–387, 2006.

[5] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Pnstructured Programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM Press.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview.

[7] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract Interpretation with Alien Expressions and Heap Structures. In *VMCAI*, pages 147–163, 2005.

[8] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language.* PhD thesis, Department of Computer Science, Iowa State University, Ames, April 2003.

[9] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML - Progress and issues in building and using ESC/Java2, 2004.

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

[11] Joseph R. Kiniry *et al.* The Logics and Calculi of ESC/Java2, November 2004. Available from http://secure.ucd.ie/products/opensource/ESCJava2/.

[12] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML reference manual. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org, August 2006.

[13] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[14] Barbara Liskov and John Guttag. *Abstraction and specification in program development.* MIT Press, Cambridge, MA, USA, 1986.

[15] Peter M&#252;ller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[16] Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.

[17] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* PhD thesis, Habilitation thesis, Technical University of Munich, January 1997. Available.

[18] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03e, Iowa State University, Department of Computer Science, May 2005.

[19] Alex Suzuki. Translating Java Bytecode to BoogiePL. Master's thesis, ETH Zurich, 2006.