Masterthesis

Generating Boogie Verification Conditions for Backward Compatibility of Class Libraries

Mathias Weber
m_weber@cs.uni-kl.de

October 15, 2012

Department of Computer Science,
University of Kaiserslautern,
Germany

Supervisors:
Prof. Dr. Arnd Poetzsch-Heffter
M. Sc. Yannick Welsch

Erklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema

„Generating Boogie Verification Conditions for Backward Compatibility of Class Libraries“

selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum) (Unterschrift)

Contents

Abstract

When changing the implementation of a library, the behavior of the old implementation has to be retained, such that programs using the old implementation still work as expected with the new implementation. If the behavior exhibited by a new library implementation is the same as the behavior of an old implementation, the new implementation is backward compatible.

This thesis presents a tool which is capable of automatically verifying backward compatibility of two Java library implementations using a coupling invariant relating the internal state of both implementations. Checking the equality of the well-defined interactions between the context and the library is the basis for verifying backward compatibility. The tool called BCVerifier works by generating a model from the byte code of both library implementations in the intermediate verification language Boogie. Support for dynamic method dispatch and recursive computation are the features that distinguish the approach from existing equivalence checking tools. Working examples show the applicability of the approach to non-trivial examples of library evolution.

Zusammenfassung

Änderungen der Implementierung einer Bibliothek dürfen nicht dazu führen, dass Programme, die die alte Implementierung verwenden, nicht mehr wie erwartet funktionieren. Das Verhalten der alten Implementierung muss beibehalten werden. Ist das sichtbare Verhalten der neuen Implementierung identisch mit dem der alten Implementierung, so ist die neue Implementierung abwärts kompatibel.

Diese Arbeit beschreibt ein System, mit dessen Hilfe abwärts Kompatibilität zweier Java Bibliotheken basierend auf einer Invariante, die die Abhängigkeiten des internen Zustands der beiden Implementierungen beschreibt, automatisch geprüft werden kann. Die Prüfung der abwärts Kompatibilität baut auf der Äquivalenz der Interaktionen zwischen dem Kontext und der Bibliothek auf. Die Software, gennant BCVerifier, generiert aus dem Bytecode der beiden Bibliotheksimplementierungen ein Modell in der Zwischensprache Boogie, welche in die Gattung der Verifikationssprachen einzuordnen ist. Die Unterstützung von dynamischen Methoden Bindung und rekursiven Berechnungen hebt den Ansatz von vergleichbaren System zur Prüfung der Äquivalenz von Programmen ab. Beispiele, die mit dem beschriebenen System erfolgreich verifiziert wurden, zeigen die Anwendbarkeit des Ansatzes auf Bibliotheksimplementierungen, die sich nicht nur unwesentlich von einander unterscheiden.

Chapter 1

Introduction

Object-oriented libraries are a means to provide functionality that solves specific tasks that can be reused in many different contexts. The implementation is encapsulated in classes with well-defined interfaces. When libraries are modified, extended and refactored, the developer performing these tasks has a responsibility to the users of the library to not change behavior of the previous functionality [27]. The contexts in which a library is used are not known to the developers working on a library implementation. Backward compatibility is the property of a library implementation to exhibit the same behavior as the old implementation in all possible contexts, the opposite of breaking API changes [13]. Backward compatibility can be reached by using informal guideline or tools, that support the developer during or after the development of the new implementation of a library is finished.

Backward compatibility consists of two layers. Source compatibility means that the new implementation of the library combined with a valid client program based on the old implementation again yields a valid program with respect to the syntax of the programming language. Valid in this regard means that the new library implementation combined with the user code does not result in type errors, if the programming language is a typed language. We focus on the programming language Java [5] which is a typed language. Source compatibility mostly bases on properties of the used interfaces of objects [25, 26]. The second layer is the behavioral layer. The two implementations of the library should behave the same way in all interactions possible in the old implementation. This property can be checked by proving both implementations of the library comply to an abstract specification of the behavior of the library. We postulate that verifying the equivalence of two similar library implementations directly is simpler than building a specification of the full behavior of the library and checking conformance of the new implementation to this specification. First of all a functional specification of the library behavior is not needed to check behavioral equivalence. Additionally, the behavior of both implementations can be compared without loosing accuracy by using an abstract specification. For a more detailed explanation as to why proving behavioral equivalence is simpler then proving the new implementation complies to the specification of the library we refer to the papers by Welsch and Poetzsch-Heffter [27] and Godlin and Strichman [16].

1.1 Contribution

We focus on the behavioral layer of backward compatibility here. The goal of this thesis is to build a tool that is capable of taking two implementations of the same library and automatically check that the two object-oriented library implementations exhibit the same behavior in the setting of unknown contexts. The check should be automatic based on the source code of both implementations and a description of the relation between the two implementations. For this we need a theoretical foundation of equivalent behavior of two object oriented libraries. This theoretical model is implemented in a model of the intermediate verification language Boogie. The model is valid if the new implementation simulates the behavior of the old implementation of the library. Dynamic method dispatch is supported, a limited support for loops and recursive methods exists as well. The byte code of the class libraries is translated into a description that follows the Boogie model. Boogie is an automatic verifier which checks this intermediate model and reports errors encountered during the proving process. The result of this process is either success, which signals backward compatibility of the two library implementations, or counter examples in which scenarios of execution backward compatibility could not be established. In addition to the description of model and implementation as well as the implementation itself, a case study of working examples shows that our approach can be used to prove backward compatibility of non-trivial examples.

1.2 Overview

Chapter 3 gives an introduction into the theoretical model we use as a basis for the backward compatibility check. The automatic verifier Boogie, its syntax and capabilities we use in the implementation of this theoretical model, as well as B2BPL, a framework we borrow part of the translation process from, are described in the same chapter. Chapter 4 describes the implementation of the theoretical model in the intermediate language of the Boogie verifier. The chapter especially focuses on a problem encountered in object oriented languages, dynamic method binding. Additionally the problem to verify equivalent behavior of loops is partially addressed. Chapter 5 gives an overview of the implementation of the Backward Compatibility Verifier (BCVerifier), the process of the translation of the library implementations into the Boogie model, and gives insight into the different components that implement this translation process. Chapter 6 shows examples that are verified using BCVerifier, including some interesting problems like a comparison between an implementation using a loop and an implementation using a recursive method. Chapter 7 introduces some projects that either implemented tools that also compare programs or program executions using behavioral equivalence or could benefit from our implementation.

Chapter 2

Theoretical Model

As one possibility to describe the behavior of a library we give a small introduction into a theoretical model of behavior equivalence that can be used to check backward compatibility. We give an overview of the language considered by this approach and give some important definitions used throughout the description of the model. We clarify how backward compatibility can be proven based on the possible interactions between the library implementation and program context. The interaction are a subset of the method calls and returns and the program context is a program including a main method using the library.

The trace-based semantics given by Welsch and Poetzsch-Heffter [26] is based on a sequential object-oriented language called LPJava (Lightweight Package Java). This language has the standard object-oriented features such as classes, interfaces, inheritance, method calls with dynamic dispatch and mutable state. Classes and interfaces can be public or package local. For simplicity all fields are considered private and all methods public. All classes have a default constructor with the same access modifier as its class. Explicit casts are possible using as special cast operator which returns a default value in case the cast is not successful. This construct removes the need for support of exceptions in this case. LPJava can be seen as a subset of full blown Java.

2.1 Definitions

There are some important definitions we list here. A *codebase* consists of a sequence of packages (i.e. P) and is denoted by the meta-variables K, X or Y. A codebase is called a *library implementation* if it satisfies all the well-formedness conditions of the language, i.e. well-formed type hierarchy, well-typedness of method bodies, etc. Well-formedness of a codebase X is formalized as $\vdash X$. A *program* is a codebase that has a main class with a main method and that satisfies all the well-formedness conditions of the language. If we join a codebase K (with a main method) and a library implementation X to form a program, we call K a *program context* of X. The well-formedness properties of LPJava are all

listed in the paper "A Fully Abstract Trace-based Semantics for Reasoning About Backward Compatibility of Class Libraries" [26] in Fig. 6 on page 8.

2.2 Source Compatibility

To be able to compare two library implementations, a prerequisite is that whenever the first library implementation joined with a program context yields a program, then the second library implementation joined with the same program context also has to yield a program [26]. This property is called *source compatibility*. For a typed language such as Java, this property is needed for the program that is formed by taking the old program context and replacing the old library implementation with the new one to type check. Source compatibility is purely a property of the typing of the program, not its behavior. In terms of the theoretical foundation, implementation Y is source compatible with library implementation X if for any codebase K, $\vdash KX$ implies $\vdash KY$.

To check source compatibility of library implementation Y with library implementation X one can check the following conditions. The package names occurring in X must exactly be those appearing in Y. Every public type in X must appear in Y. For public types in X, every public method which is part of the type (declared or inherited) in X must also have a method with the same signature in Y and vice-versa. The subtype hierarchy between public types of X must be preserved in Y. These rules are taken from the paper of Welsch and Poetzsch-Heffter [26].

2.3 Subtypes and Inheritance

LPJava, as a class based programming language, supports subtyping of classes and inheritance of methods. For two classes C and D, if D inherits from C, written D **extends** C, then all methods that are defined in C are also defined in D. Class D *inherits* all methods from class C. For all methods of a class C, if C includes an implementation of a method then C defines this method. If a method m is defined in a class C then it is valid to call m on an object that is of type C.

To distinguish overloaded methods, methods with the same name but different parameter count or types and possibly different return type, when talking about a method we mean the signature of the method consisting of the name of the method, the types of the parameters, and the the return type. A method m is *member of* class D with implementation in class C if D is subtype of C, D has no implementation of m, C has an implementation of m, and there is no proper subtype of C that is supertype of D that has an implementation of m. If class C has an implementation of method m then m is always member of C with implementation in C. If m is member of C then m is defined in C. When calling a method m that is member of D with implementation in C then the body of the implementation in C is executed.

Interfaces define types as well. A class can implement multiple interfaces. Each type defined by a class implementing an interface is a subtype of the interface type. Interfaces can not be instantiated and include only method signatures but no method implementations.

The types of a program are separated into *library types* and *context types*. The library types are the types defined in an implementation of the library, the context types are the types defined in the program context. The types of the library are closed with respect to supertypes. The supertype of a library type is again a library type. The top of the type hierarchy is java.lang.Object which is defined to be a library type. The context types are not known to either library implementation and can not be used throughout the implementations.

Types are clustered into packages. A type belongs to exactly one package. Packages of the library are sealed, it is not possible for the context to define a type that belongs to a package of the library. The second implementation may define types that are not part of the first implementation. New packages may not be introduced by the second implementation. This fact together with sealed packages avoids conflicts between the context types and the library types. Library types belong to the packages of the first library implementation, context types do not belong to these packages.

2.4 Observable Behavior

To show that two library implementations are backward compatible, one shows that the second implementation simulates the behavior of the first implementation. This property can be checked by showing that the second implementation shows the same observable behavior as the first implementation for each possible program context the first implementation can be used in.

State Abstraction To verify that the behavior of two library implementations is equivalent in all possible contexts, a model of the behavior of each implementation is needed that abstracts from the internal state of heap and stack. The inner workings of both implementations such as the names of variables and the representation of the state can be quite different even though the behavior visible to class contexts is the same. In the theoretical model used in this thesis, the behavior of a class library is described by the possible *interactions* between the library and the program context it is used in. The notion of an interaction is defined based on change of control.

Interactions The library controls execution if code of the library is currently executed, otherwise the context controls execution. As LPJava is a purely sequential subset of Java without parallel execution, the control flow can at a fixed point in the execution either be in code of the library or in code of the context. An

interaction is a switch of control flow from the library to the context or vice-versa. In LPJava, interactions can only happen by method calls and method returns. A method call is said to be a *boundary call* or cross boundary if the code the method invocation appears in belongs to the library and the code of the method called belongs to the context or vice-versa. The same is true for a method return from a method of the library to a method of the context and vice-versa, known as a *boundary return*. A call of method m which is member of class D with implementation in class C is a boundary call into the context if the call site is in the library and C is a context type. A library implementation can neither instantiate nor extend a context type. As such the object the boundary call is performed on has to be created by the context. A call of a method with implementation in the library called from within a method of the context is also a boundary call.

Interaction Frames Both library implementations each have their own stack. If a method belongs to the context then the stack frames used to execute this method also belongs to the context. The same holds for methods of the library whose stack frames belong to the library. Method calls that are no boundary calls, these calls are internal calls, create stack frames that belong to the same entity, either the library or the context. This way an execution of a program creates alternating continues sequences of stack frames that belong to the context and stack frames that belong to the library. These continuous sequences of stack frames that belong to the same entity are clustered into *interaction frames*. Interaction frames are only created by boundary calls and destroyed by boundary returns.

Exposed Objects Objects carry information about if they are created by the context or if they are created by the library. All objects start as internal objects once they are created. If an object participates in an interaction, thereby crossing the boundary between library and context, then this object gets exposed.

Behavior Comparison Library implementation two simulates the behavior of library implementation one if whenever during the execution of implementation one an interaction is possible then the same interaction must also be possible in implementation two. The interactions must be equivalent, meaning the method called or returning must be the same and the objects involved in the interaction, such as the parameters of the method or the method result, must also be observational equivalent. Two objects o_1 and o_2 are behavioral equivalent if they show the same behavior, again judging about the possible interactions, and additionally the types of the objects must not distinguish the objects from each other from the point of view of the context. The context only knows about the public types of library implementation one, so for all super types t of o_1 that are public in implementation one, t must also be super type of o_2. From implementation one and two being source compatible follows that t is public in implementation two if it is public in implementation one. Two method calls are equal if the name of the method is equal, the number of parameters is equal, the public supertypes of the

parameters are equal modulo new public types in implementation two, and the objects involved in the interaction are observational equal. Two method returns are equal if the returning method is equal, the public supertypes of the return type are equal modulo new public types in implementation two, and the returned object is observational equal.

Related Objects Object appearing in the same position in two equal interactions are said to be corresponding objects or in *correspondence relation*. For method calls c_1 and c_2, c_1 being a method call in the first implementation and c_2 being a method call in the second implementation, if c_1 and c_2 are equal then the first parameter of c_1 is related with the first parameter of c_2, the second parameter of c_1 is related with the second parameter of c_2, and so forth for the other parameters as well. For two equal method returns, the returned objects are related.

Backward Compatibility Implementation Y is backward compatible with implementation X, if X and Y are source compatible and for all contexts K the resulting interaction for each interaction between K and either X or Y respectively is equal. Since we use the same context K for the execution of both library implementations, the interactions between K and the respective implementation are equal as long as the answers to interactions of the context are equal.

2.5 Reasoning Method

The reasoning method to prove backward compatibility is described in the following and is based on a simulation of the interactions between possible contexts and both library implementations. For a library implementation to be backward compatible with an old implementation, the new implementation needs to simulate the functionality of the old implementation. The state needed by the computation performed by this functionality needs to be represented in both implementations, even though the concrete representation may be quite different. We know that there always is a relation $\preccurlyeq^{\rho}_{ctxt}$ between the state of the two library implementations if they are backward compatible [26]. The context, so the program that was built to use the old implementation of the library and is supposed to work using the new implementation, is fixed, the runtime configurations or state of this context is related over $\preccurlyeq^{\rho}_{ctxt}$. This means that the interactions made by the context are simulated properly if the new library implementation is backward compatible, since the interactions made by the context depend on the state of the context. The goal is to show that the interactions of the library implementations are simulated properly, as well. To prove this property we need to know the relation between the library parts of the corresponding states. We call this relation the *coupling relation*.

The coupling relation can rely on some properties of $\preccurlyeq^{\rho}_{ctxt}$. The objects used in the interactions made by the context are related, since these interactions are simulated properly. Objects that take part in an interaction get exposed in the process. There is always a correspondence relation ρ from the exposed objects of the first configuration to the exposed objects of the second. This relation can be used to describe the coupling relation.

The coupling relation can be given in form of a *coupling invariant* which uses the correspondence relation to talk about corresponding objects. A coupling relation is adequate if the coupling invariant holds initially, before the first step of the execution of the program is made, steps in the context may not destroy the invariant and the coupling invariant has to hold after each interaction between library and context. This means the coupling invariant has to hold whenever the control flow is outside the library.

The goal of this thesis is to implement a Backward Compatibility Verifier that proves that a specified coupling invariant is an adequate coupling, showing that the two library implementations are backward compatible.

In the following we give a small introduction to the framework of our tool, Boogie, a verification condition generator for modern object-oriented languages, and B2BPL, a classical verifier for abstract specifications based on Java byte code.

Chapter 3

Tool Framework

3.1 Boogie

In this section we give an introduction to the automatic verifier Boogie. The use of an automatic verifier relieves the user from the task of manual interaction with the verifier to discharge proof goals, which is a very complex task not many users are willing to do. Boogie is used in our implementation because of the rather simple mapping of procedural aspects of the Java byte code to the language Boogie uses to describe the model.

3.1.1 Overview

Boogie is used as an intermediate language for automatic program verifiers for modern programming languages. The Boogie translator generates *verification conditions*, logical formulas whose validity imply that the program under consideration has the expected properties. A successful proof attempt of an automatic theorem prover shows the correctness of a program, a failed attempt may originate from an error in the program or a specification that is too weak.

Boogie is already targeted by some specification languages like Spec# [6], Dafny [3] and Chalice [2], to mention only a few. The success of Dafny, the verifier was used for example in the VSTTE 2012 program verification competition, the COST Verification Competition 2011, and the VSComp 2010, shows the potential of this tool to handle even complex problems. We try to use this potential to our advantage to avoid having to generate these verification conditions ourselves.

The Boogie language is rather high level. It offers all needed procedural constructs such as variables, while-loops, jump statements and procedures. Other than this, the language also offers some functional constructs such as functions and custom datatypes. Additional there are the usual logical constructs such as axioms, assumes and asserts as well as pre-, post- and frame conditions of user defined procedures. Axioms introduce global facts, assumptions are used to introduce facts that hold at a specific point in the execution of a program, and assertions

are used to add proof goals. What Boogie generates are formulas for an SMT solver. This intermediate language reduces the work one has to put into the generation process to come from an object-oriented view down to the formula-based view of a theorem prover. At the same time Boogie tries to generate some details that normally have to be generated by our framework, for example triggers for axioms, that tell the solver when to apply a rule, and basic invariants for loops. Those only have to be fine-tuned for performance and completeness. Deducing strong enough loop invariants from the program code only is not always possible. Facts that can not automatically be deduced have to be added for the prover to verify loops correctly. Triggers of axioms are crucial for the performance of the proving process. Reducing the number of cases a rule is used in also reduces the solution space that has to be explored. On the other hand, this might also remove the valid solution, which leads to unsuccessful proofs even though the goals are fulfilled. Fine-tuning triggers of axioms is a very complex problem.

3.1.2 Features

Introduced here is version 2 of Boogie [22], which offers a richer type system compared to the previous version.

Types

Other than the built-in types, **bool** and **int**, parameterized type constructors and polymorphic map types can be used. **type** Barrel α defines a type constructor with one parameter. The name of the parameter is irrelevant, the type constructor can also be written **type** Barrel _. An invocation of the type constructor with an existing type as parameter yields the corresponding concrete type. For an existing type Wicket an instantiation of the constructor Barrel yields a new type Barrel Wicket. [Barrel Wicket] Wicket is the type of a map which has type Barrel Wicket as type of the indices and the type of the values of this map is Wicket. One can use type $<\alpha>$[Barrel α] α to abstract from the type of the values, so this map can be used with any type of Barrel, the type of the value depends on the type instantiation of the parameterized type constructor Barrel. Type synonyms are different names for existing types. **type** MultiSet α= [α] **int** introduces the name MultiSet for a map from some type to **int**.

Boogie offers a partial order <: for each built-in and user-defined type. Being a partial order, <: is reflexive, transitive and antisymmetric. One can use this order for example to describe the type hierarchy of a programming language. For two constants representing classes A and B the expression A <: B tells Boogie that A is subclass of B, A **extends** B in Java notation.

The features of Boogie can be divided into two parts, the procedural and the logics-part.

Procedural Part

The procedural part can be used to describe procedures that can be executed. Procedures are one of the most important entities in Boogie. These entities are the only ones that are actually validated. A procedure consists of two parts, a specification and an implementation. The specification of a procedure again consists of preconditions, postconditions, frameconditions and additional constraints. A typical specification of a procedure looks as in the following example:

```
1 procedure P<α>(ins) returns (outs); spec
```

P is the name of the procedure, ins are the in-parameters and outs are the out-parameters or return values. Parameters may be annotated with **where** clauses. These clauses give a hint on what has to hold for the possible values of the parameter. These clauses are not checked but assist in narrowing down the possible values of the parameters for the proving process. The specification spec can consist of any number of

- **requires** clauses, which have to hold in the initial state of the procedure and have to be established at the call-site. **free requires** clauses are assumed to hold for the implementation but do not have to be established by the caller.
- **ensures** clauses, which have to hold in the post state of the procedure and can be assumed to hold at the call-site.
- **modifies** clauses, which specify which global variables can be modified inside the procedure implementation.

The implementation of a procedure is normally given separately from its specification. It is also possible to combine the implementation with the specification by inlining the implementation body into the specification. The following two definitions are equal:

```
1 procedure P<α>(ins) returns (outs) spec {body}
2
3 procedure P<α>(ins) returns (outs); spec
4 implementation P<α>(ins') returns (outs') {body}
```

The body of an implementation consists of a list of basic blocks. The basic blocks in Boogie are not basic blocks in the strict sense. It is actually possible to write only one basic block without a label and use control constructs such as **if** statements and **while** loops in this block. The Boogie compiler desugars these control statements into basic blocks. The statement **if** (E) Thn **else** Els is desugared into

```
1 goto L0, L1;
2 L0: assume E; <Thn> goto Done;
3 L1: assume !E; <Els> goto Done;
4 Done:
```

L0 and L1 are the labels of the basic blocks generated by Boogie. The command **goto** L0, L1; is a non-deterministic branch to both basic blocks. An **assume** statement introduces facts that hold at specific points in the execution as described in the logics-part of this section. Using this construction both possibilities, the execution of the Thn block assuming E holds and the execution of the Els block assuming that E does not hold, are checked.

Variables can be used to represent the state of the program. Also supported are map assignments and pre- and post conditions. Variables can be declared top level or as local variables inside a procedure. Local variables are only visible inside the body of the implementation they are declared in. A variable can be defined in the following way: If v is a yet unused name, **var** v: **int**; defines a new variable of type **int**. New values can be assigned to variables using the operator :=. For an existing variable i of type **int**, the command i := 0; assigns the value 0 to i.

In addition to variables, Boogie also supports constants. Constants can be declared as top level expressions. Constants that are declared **unique** have a value different from all other constants of the same type that are also declared **unique**. This property avoids having to declare an axiom for the quadratically many distinctions that the value of each unique constant is unequal to the value of each other unique constant of the same type. For a not yet used name null and an existing type Ref the statement **const unique** null: Ref defines a new unique constant of type Ref.

Boogie maps can be seen as a mapping from indices to values that can be changed. Map selection is done using a syntax that reminds of array access in, for example, Java. For b is a variable or constant of a map type with a single index and i is a variable or constant of the index type of b, b[i] is the value of b at index i. Map update is also allowed using the following syntax:

```
1 b[i := v]
```

The map returned by this expression is a map where all values are the same as in the original map b, only the value at index i is equal to the value of v. For variables there is an abbreviation, the following two commands are equal

```
1 b[i] := v
2 b := b[i := v]
```

The **havoc** command assigns arbitrary values to a set of variables. If a variable is annotated with a **where** clause, this restriction is considered when choosing the new value to be assigned. The following example shows the effect.

```
1 function WellformedHeap(Heap) returns (bool);
2 ...
3   var heap: Heap where WellformedHeap(heap);
4 ...
5   havoc heap;
6   assert WellformedHeap(heap);
7 ...
```

Let the function WellformedHeap(Heap) define some properties of heaps that should always hold. The variable heap is declared using a **where** clause, which means Boogie will choose the initial value such that WellformedHeap holds. The assertion directly after **havoc** in line 6, a property that is defined to hold at this particular program point, will always yield **true** in this scenario, because the new arbitrary value will be chosen accordingly.

Logics Part

The logics-part is used to describe the proof goals that have to be fulfilled by the program.

Boogie supports the standard logical operators (<==>, ==>, ||, &&, ==, !=, <=, >=, !) and the standard mathematical operators (+, −, ∗, /, %). The meaning of / and % has to be defined using axioms, because their behavior is highly dependent on the programming language that is used as source language. The standard boolean quantifiers (**forall** and **exists**) are supported as well.

Axioms are facts that hold globally, thus can only be declared as top level elements. Boogie generally will deduce from the validity of the left side of the implication the fact on the right side. Assumes can be used to introduce facts that hold at a specific point of the execution of the program. Asserts are used to introduce proof goals at specific points of the execution. The facts given by assumes are used by Boogie to check proof goals as well as reduce the possible executions of a procedure. A part of a procedure that leads to a contradiction of assumed facts is abandoned.

Boogie also offers a **call** statement, which is intended to simulate calling a given procedure from inside another procedure. What actually happens is that this statement is replaced with checks of the non-free preconditions of the procedure and assumes of the non-free postconditions of the procedure.

```
1 procedure P(args) returns (res);
2 requires Q;
3 modifies F;
4 ensures R;
5 ...
6   call res' := P(args');
```

The call to procedure P in line 5 gets translated more or less into the following statements:

```
1 ...
2   assert Q;
3   havoc F;
4   assume R;
```

As far as the value of variables is not explicitly given, the proving process considers all possible executions of the given procedures. Using assumes, those possible executions can be restricted.

Functions capture an expression parameterized over a set of variables, the parameters of the function. Function name and their signature can be defined separately from their body. An axiom than be used to define the behavior of the function. The two constructs

```
1 function F(args) returns (res) {E}
```

and

```
1 function F(args) returns (res);
2 axiom (forall args . F(args) == E;
```

are equal. The body expression of the function is inlined in the first example and given as an axiom in the second example. Here F is the name of the function, args are the parameters of the function that optionally may have a name and res is the type of the result. E is an expression with args as free variables. A function can have multiple arguments. The types of the parameters can even be parameterized, as in

```
1 function volume<α>(Barrel α) returns (int);
```

This function is meant to return the volume of a barrel. Type type of the content of a barrel is abstracted into the type parameter α. So this function can be used to get the volume of any kind of barrel, independent of the type of its content.

3.1.3 Verification Condition Generation

The Boogie compiler takes models written in the Boogie language and generates conditions that can be processed by an SMT solver, for example Z3 [7]. The different statements and expressions are generated into logical formulas which are checked for validity by an SMT solver. The performance of the proving process is highly dependent on how the axioms are processed. Normally the solver generates triggers for all existential and universal quantified axioms on its own. During the proving process all formulas are used to derive new formulas whenever their trigger matches.

Boogie offers a way to customize the triggers used for the quantified axioms. This can help rule out triggers that do not lead any closer to the proof goal. Let us look at an example.

```
1 (forall x: int :: {h(x)} h(x) < h(k(x)) )
```

In this example it is legal to instantiate `x` with any expression of type **int**. The problem is, if the quantifier is instantiated, the instantiation produces a term with another argument of `h`. If a formula `h(X)` has to be derived during the proof process, the instantiation will produce `h(k(X))` and the axiom can be re-instantiated with `k(X)`. The quantifier will be instantiated with `X`, `k(X)`, `k(k(X))` and so on, causing a matching loop. The trigger `h(k(x))` on the other hand will not lead to a matching loop. More detailed information on this topic is given in the paper of Leino and Monahan [19].

3.2 B2BPL

As the programming language with class oriented features we have chosen Java. The backward compatibility checker needs to translate from Java or Java byte code to Boogie. We give a short introduction to B2BPL, which is a tool specialized to verify Java byte code programs against classical abstract specifications.

3.2.1 Overview

B2BPL [18, 20, 23, 28] is a verification framework targeted towards checking that a byte code implementation complies to a specification written in a special specification language called BML (Bytecode Modeling Language) [12]. BML is a behavioral interface definition language for Java byte code and is inspired by JML, its Java pendant. The main idea behind B2BPL is to check that a given Java implementation of a library compiled to byte code complies to the annotated specification. In Fig. 3.1 we see an overview of the translation process.

The Java sources contain the implementation of the library as well as its behavioral specification. These sources are compiled to class files which are parsed using the ASM framework [1]. During this process the BML specification is extracted and an AST of the byte code is formed. A translator generates a Boogie procedure for each method in each class it encounters. The pre- and postconditions of these procedures originate from the BML specification. The Boogie procedures generated from the remaining byte code are annotated with the corresponding BML specifications. The result of the transformation process is a Boogie AST which is finally written to a file which can be fed to Boogie to check the library satisfies the specification of its behavior.

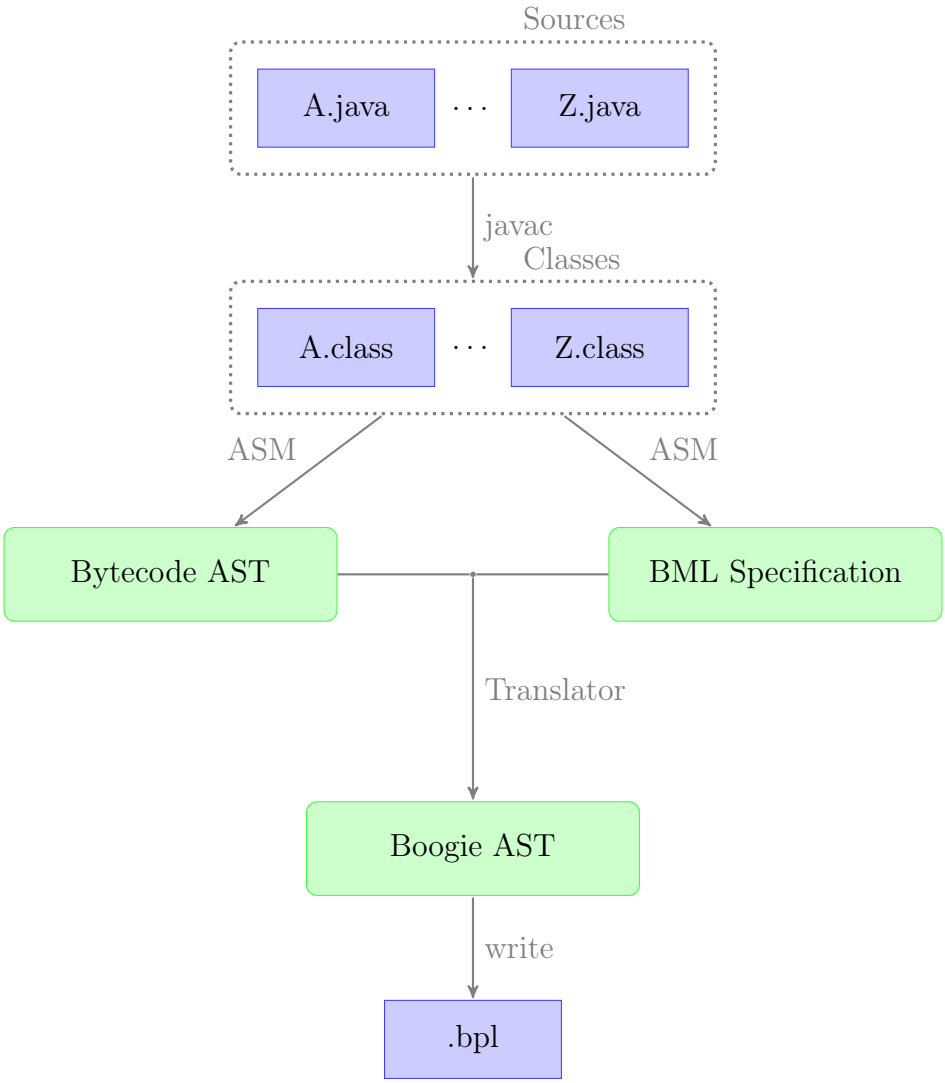


Figure 3.1: Overview of the B2BPL translation process

Since the implementation of B2BPL is older than the introduction of Boogie 2, the implementation still uses Boogie version 1, also called BoogiePL, which lags some of the advanced features outlined in Section 3.1 such as parameterized types and polymorphic maps.

The implementations of the Boogie procedures is purely based on basic blocks. So none of the higher-level constructs of Boogie, for example **while** loops and **if** branches, are used to translate the byte code. In the following we will give a short introduction into how the translation from byte code to Boogie works in B2BPL.

3.2.2 State Model

The state model used in the Boogie representation of the library is based on the logical foundations introduced by Poetzsch-Heffter and Müller [21]. The main type that represents a storage is Store. The heap of a running program is modeled as a variable of type Store. A Value is one of two things: An object or a primitive value, such as an integer or a boolean. These values are stored in a Store at a certain Location. A Location represents the location of a field inside an object or the location of an array element inside the array. Such a location can be computed given an object reference and a field-id or the reference point of an array and the index. An Allocation represents the nature of the object being allocated, either a class or array type. Location and Allocation together allow uniform access to the heap.

The most important functions to manipulate the heap are the following:

```
1 function update(Store, Location, Value) returns (Store);
2 function get(Store, Location) returns (Value);
3 function new(Store, Allocation) returns (Value);
4 function add(Store, Allocation) returns (Store);
```

The functions update and get are used to retrieve and manipulate the value of existing objects. The functions new and add are used to create new objects or arrays. add returns the heap after a new objects or array has been allocated, depending on the type of the given Allocation. new returns a newly created object or array of the given type.

An explicit model of a stack frame is not used by B2BPL, since each Java method is translated into its own Boogie procedure, so there is only a single stack frame to consider for each generated Boogie procedure. The stack frame and the local variables of a method are translated to local variables of the Boogie procedure. The Java virtual machine has a stack based architecture, the state of the computation is represented in form of a single stack. Byte code operations can add elements at the top of the stack or perform computations with the top-most elements of the stack. For each position of the stack and each type out of **int** and ref variables of the form stack<number>_<i or r> are added to the local variables. The bottom element of the stack of type **int** is stack0_i. The Java type **boolean** is also translated to the Boogie type **int**, since the Java byte code operations for both types are equal as well. Local variables follow the same pattern with prefix reg. The first local variable of the Java method of type object is reg0_r. Since all parameters of the Java method can be accessed as local variables, as defined in the Java Language Specification [5], the parameters of the Boogie procedure, called param0_r to param<n>_r get assigned to the local variables reg0_r to reg<n>_r. The same holds for the integer variant respectively.

3.2.3 Axiomatization of the Type System

Java types are represented simply by the type name. To distinguish between the different kinds of types like class-types, value-types/primitive-types and array-types there are different functions:

```
function isClassType(name) returns (bool);
function isValueType(name) returns (bool);
function isArrayType(name) returns (bool);
```

The primitive-types are modeled as constants of type name. An axiom defines those types as being the only value types.

```
const unique $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);
```

The range of the values of the primitive types is given by another function called isInRange.

```
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);
```

Classes encountered while analyzing the library are added as constants of type name and an axiom is added expressing that this new constant represents a class-type. The Boogie provided order <: is used to represent the type hierarchy.

The class java.lang.Object is a special case, it is the top of the subtype hierarchy. This is stated by the axiom

```
axiom (forall t: name :: $java.lang.Object <: t ==> t ==
    $java.lang.Object);
```

For all classes, their supertypes are added to the subtype relation using axioms. For class C with superclass A and implementing interfaces I_1 to I_k, the following axioms are added:

```
axiom $C <: $A;
axiom $C <: I_1;
axiom $C <: I_2;
...
axiom $C <: I_k;
axiom (forall t: name :: $C <: t ==> t == $C || t == $A || t == I_1
    || ... || t == I_k);
```

These axioms describe the relations between the class and its supertypes. A special focus lies on the fact that all supertypes are know at translation time, so the supertype relation is closed.

For types that are final an axiom is added that defines that this class type has no other subtypes.

```
(forall t: name :: t <: $C ==> t == $C)
```

In this example the class named C is defined to be final. The axiom tells Boogie that if a class is subtype of C than this type is C itself.

Arrays are created using the same mechanisms as objects. Their types are created using the following function:

```
function arrayType(name) returns (name)
axiom (forall t: name :: isArrayType(arrayType(t)));
```

The axiom tells Boogie that a type created using function arrayType is actually an array type.

3.2.4 Control Flow

As basis of the transformation process the control flow graph of the byte code instructions is built. This graph allows to find back-edges which indicate loops and the next basic block for the currently translated basic block. When a loop is detected during translation, assertions and assumptions of the loop invariant, which is defined in the BML specification, is added in the right places to handle loop verification correctly. The loop invariant is asserted right in front of the loop and at the end of the loop body, and assumed right inside the loop after **havoc**ing the variables used inside the loop.

```
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;
```

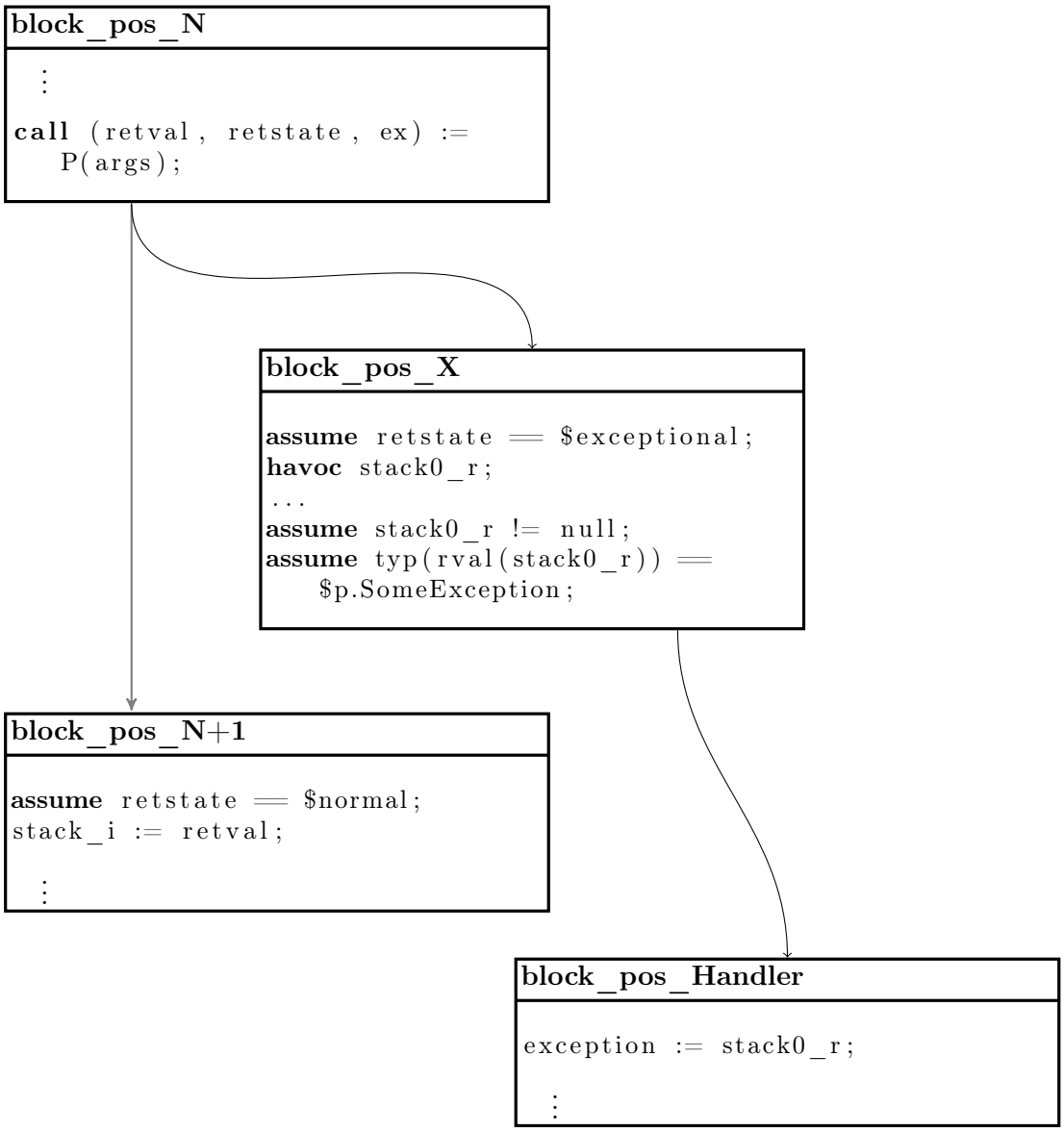
```
entry :
  x := 1;
  assert x <= 100;
  goto loop;
loop:
  havoc x;
  assume x <= 100;
  goto body, exit;
body:
  assume x < 100;
  x:=x+1;
  assert x <= 100;
  assume false;
  return;
exit :
  assume !(x < 100);
  assert x == 100;
  return;
```

The example is taken from a master project report by Mallo [20]. The left column shows the originally translated Boogie code of a typical loop. The label loop: marks the loop header which was recognized using the flow graph. The loop iterates a variable x, the loop condition is x<100, so the original loop might be a **while** loop such as the following: **while**(x<100){x++;}. The loop invariant is x<=100. The first step is to move the loop invariant up to all branches that directly lead to the loop header and add an equivalent **assume** statement to the loop header. The result of this step is shown in the middle column. In the final step a **havoc** is added to the loop header to reset the information about the variable x. That way the prover only has the information given in the loop invariant and shows for an arbitrary loop iteration that the loop invariant is preserved. The final result of the loop invariant translation is shown in the right column. The heap state before entering the loop is saved to a new variable and can be accessed by a BML specification. This way a loop variant can also be formulated to define what the effect of a loop iteration is.

If-branches are translated into two basic blocks, one for each branch, with appropriate assumes added, just as the Boogie compiler would desugar an **if** construct, described in Section 3.1. The two bodies of the if- and the else-branch are translated into additional basic blocks. The first assume of the first basic block of these bodies is the boolean condition of the **if** and the negated boolean condition, respectively. The checking process traverses both branches non-deterministically, so both possible executions are considered.

B2BPL has a model of exceptions and explicit handling of runtime exceptions. Standard behavior is to try to avoid runtime exceptions by adding additional checks to the program that rule out exceptional cases. When the alternative runtime exception model is activated, the following translation allows explicit handling of occuring exceptions. Method calls are translated into Boogie **call** statements. Each Java method is translated into a procedure enriched with a return

state and an exception as result parameters. At the call-site the in-parameters to the method are taken from the stack and passed to the procedure, the results of the procedure, a return value, a return state and an exception the method has thrown if the return state is $exceptional, are captured. Each method in Java that handles exceptions using a try-catch has an exception table added to the byte code when translated. For each possible exception the method might throw, branches to appropriate exception handlers are added, guarded by a check to the result state and thrown exception of the procedure.



In this example, the method P called in the basic block block_pos_N may throw an exception p.SomeException. Beside the statements following the method call, which are translated to the basic blocks block_pos_N+1 and subsequent blocks, the exception handler that catches the method thrown by P is translated into a block called block_pos_Handler. The basic block block_pos_N includes two jumps for the two situations that could occur. Either method P returns normally, then

the execution is continued with block block_pos_N+1 and the return state retstate is $normal. The second possible situation is that the method throws an exception. In this case the return state retstate is $exceptional and the jump to the exception handler is executed. The block block_pos_X states the situation where an exception is thrown by method P. The thrown exception is defined to be the top most element of the stack and has dynamic type p.SomeException.

After the call to the procedure, branches are added for the normal run as well as all exceptional runs through the exceptional handlers mentioned in the exception table. The return state of a procedure can either be $exceptional or $normal. The blocks leading to the exception handler is annotated with the concrete type of the exception the handler is supposed to handle. If more than on exception might be thrown, assumes are added to rule out the other exceptions that are handled by other handlers. This simulates the behavior of the Java exception handling of the try-catch construct.

3.2.5 Translation of Basic Constructs

The translation of the basic constructs of Java byte code is straight forward. Here are some examples.

Access to fields of an object, the Java byte code operation getfield , are translated to accesses to the heap. The location of the field with respect to the object reference gets computed, this location is afterwards used to access the heap. If stack0_r references an object of type Account, an access to the field Account.balance looks as in the following example.

```
1 stack1_i := toint(get(heap, fieldLoc(stack0_r, Account.balance)));
```

The function fieldLoc computes the location of the field balance with respect to the object reference stack0_r. The function get, as we have seen in Section 3.2.2, is used to access the heap and retrieve the value. The function toint is used to convert this generic value to a Boogie integer.

Setting the value of the field balance of the object pointed to be stack0_r to the integer value of stack1_i is translated to the following.

```
1 heap :=
    update(heap,fieldLoc(stack0_r,Account.balance),ival(stack1_i));
```

The function update updates the value of the field balance in the heap, as described in Section 3.2.2. The function ival is the reverse operation of the function toint. It converts a Boogie integer value to the internal representation used in the Store abstraction such that the following holds.

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

The translation of the mathematical operations is strait forward. Stack operations such as dup and swap are translated to assignments of the stack variables. Let stack<n>_r be the top most element of the stack. Then the operation dup is translated to stack<n+1>_r := stack<n>_r and the operation swap is translated to stack<n−1>_r, stack<n>_r := stack<n>_r, stack<n−1>_r. The assignment in the last case is simultaneous, so the values of the two variables are swapped.

Chapter 4

Boogie Model

The goal of this thesis is to develop a Backward Compatibility Verifier based on the formal model we introduced in Chapter 2. Since this verifier should work on class-libraries, we decided to focus on Java [5] as a modern object-oriented class-based programming language.

In this section we will show the Boogie model into which the Java code is translated, which is an implementation of the formal model we use. We give an impression on how two library implementations can be related using Boogie and how this relation can be used to prove backward compatibility. We show how to distinguish between boundary method calls as interactions between library and context, and internal calls, which are not directly observable by the context. We also show how dynamic and static properties of objects can be used to guide the proving process and how these properties can be used to reduce the number of cases the SMT solver has to consider.

4.1 Supported Java Subset

The Java subset supported by BCVerifier differs from the Java subset supported by LPJava, the programming language used as a basis of the theoretical model, in the following ways. (1) The primitive integer types **int** and **long** are supported to be able to compute numbers, a basic operation for more complex examples. The Java type **boolean** is supported as well, since the byte code operations for integers and booleans are identical, 0 represents the value **false**, all other integers represent the value **true**. (2) In contrast to LPJava, methods may not only be **public**, they may also be declared to be **private**. This allows to express that calls to a specific method may not be interactions with the context. An example that uses this construct is a recursive method that implements an internal computation. Public methods can be distinguished from private methods, the function isCallable(...) yields **true** for public methods and **false** for private methods. (3) Constructors can have an implementation and are handled the same way as normal methods. Additionally there may be more than one constructor per class, as Java supports

overloading not only for methods but also for constructors. (4) The parameters of methods are handled as if they were final. Changing the value of a parameter inside a method is not supported. This restriction has the effect that the parameters are related or equal in every program point of a method. (5) Runtime exceptions cannot be completely avoided in the chosen subset of Java. Object references may be **null**, because of this method calls and field access on references may fail, and integer divisions may fail because of division by zero. The translation process may be configured to generate proof goals that avoid these kind of exceptions, which leads to a failed proof if the specification is not strong enough, or to assume that these kind of exceptions do not occur and to restrict the values of variables accordingly.

4.2 State Model

The state of the two library implementations is captured using an abstraction of the heap and an abstraction of the stack. Both use polymorphic maps, a new feature of Boogie 2 described in Section 3.1. Since the implementation uses B2BPL as basis, we will highlight the difference between the B2BPL model and our model where appropriate.

Heap

The heap is modeled as a mapping from an object reference and a field to the value of the field.

```
1 type Heap = <alpha>[Ref, Field alpha]alpha;
```

The fields of classes are translated to unique constants of type Field **int** or Field Ref for fields with primitive type or fields of a class or interface type. To access the heap one needs an object reference as well as a field constant. The result is the value of the field with type **int** or Ref, depending on the type parameter of the field constant. For a heap heap, an object o and a field f, the expression heap[o, f] gives the value of f for object o.

The operations on the heap are also translated differently. An update of a value on the heap as well as adding a new value to the heap is translated into a map assignment instead of the functions update or add. An access to the value of a field or array element is translated as a map select instead of the function get, as seen above. This also means that the creation of a new objects has to be translated differently as well. The heap consists of all possible objects, allocated and non-allocated. We pick a non-allocated object from the heap, allocate this new object, and assume the properties this object should have.

```
1  var tmp: Ref;
2  ...
3    havoc tmp;
4    assume !heap[tmp, alloc];
5    heap[tmp, alloc] := true;
6    assume P(tmp);
7    ...
```

In lines 3 and 4, a non-allocated object is chosen from the heap. Line 5 allocates this object. The predicate P(Ref) stands for the properties the new object should have such as the type of the new instance and that this new object should not be null.

A valid heap has to fulfill some properties that are listed in the function WellformedHeap.

```
1  axiom (forall heap: Heap :: WellformedHeap(heap)
2    <==>
3  heap[null, alloc] &&
4  (forall r: Ref, f: Field Ref ::
5    heap[r, alloc] ==> heap[heap[r, f], alloc]) &&
6  (forall r: Ref, f: Field Ref ::
7    !heap[r, alloc] ==> heap[r, f] == null) &&
8  (forall r: Ref, f: Field int ::
9    !heap[r, alloc] ==> heap[r, f] == 0) &&
10 (forall r: Ref, f: Field Ref ::
11   isOfType(heap[r, f], heap, fieldType(libImpl(heap), f))) &&
12 (forall r: Ref, f: Field int ::
13   isInRange(heap[r, f], fieldType(libImpl(heap), f))));
```

The value null is always allocated since it is a build-in value of Java. For all objects that are allocated also the fields of those objects are allocated. This also implies that those fields are either allocated objects or null. All objects that are not yet allocated are uninitialized, meaning the fields are null for object references and 0 or **false** respectively for primitive types. For all fields of objects on the heaps holds that the values those fields point to are of the field type or the range of the value does not exceed the range of the field type. The field type is the static type of the field as declared in the class definition. The definition of the field type is parameterized on the library implementation the class belongs to, in which the field was declared, as described in Section 4.5.

Stack

The runtime stack is also represented using a polymorphic map. A StackFrame is defined as a mapping from variables to their value.

```
1  type StackFrame = <alpha>[Var alpha]alpha;
```

Stack frames are clustered into interaction frames. A stack frame in an interaction frame is addressed using a stack pointer. A StackPointer is simply an alias to **int**. An

interaction frame can belong to the library or the context. The interaction frame 0 belongs to the context, since the main method belongs to the context and this is where the execution starts. A stack is represented by an alternating sequence of interaction frames that belong to the library and interaction frames that belong to the context starting at 0. The type Stack is defined as a mapping from an integer index to an interaction frame [**int**] InteractionFrame and an interaction frame is represented by a map from stack pointer to stack frame [StackPointer] StackFrame. Since every interaction frame may have a different number of stack frames, we need a mapping from interaction frame to stack pointer. This mapping is represented by a stack pointer map spmap, which is a mapping from the index of the interaction frames to the stack pointer of the top most stack frame of this interaction frame [**int**] StackPointer. The stack frames of interaction frame i can be addressed by stack[i][0] for the first stack frame of i to stack[i][spmap[i]] for the top most stack frame of i. Variables on the stack are represented as unique constants of type Var **int** or Var Ref for primitive values or object references.

For stacks as for heaps there is a definition of a wellformed stack. The properties that make a stack wellformed are listed in the function WellformedStack.

```
1  axiom (forall stack: Stack, i: int, spmap: [int]StackPtr, heap: Heap
       :: WellformedStack(stack, i, spmap, heap)
2    <==>
3  i >= 0 &&
4  (forall j: int :: 0 <= j && j <= i ==>
5    0 <= spmap[j]) &&
6  (forall j: int, p: StackPtr, v: Var Ref ::
7    j <= i && 0 <= p && p <= spmap[j] ==>
8      heap[stack[j][p][v], alloc]) &&
9  (forall j: int, p: StackPtr, v: Var Ref ::
10   j > i || (j <= i && (p < 0 || p > spmap[j])) ==>
11     stack[j][p][v] == null) &&
12 (forall j: int, p: StackPtr, v: Var int ::
13   j > i || (j <= i && (p < 0 || p > spmap[j])) ==>
14     stack[j][p][v] == 0) &&
15 (forall sp: StackPtr, j: int ::
16   j < i && sp <= spmap[j] ==>
17     stack[j][sp][result_r] == null && stack[j][sp][result_i] == 0) &&
18 (forall sp: StackPtr ::
19   sp < spmap[i] ==>
20     stack[i][sp][result_r] == null && stack[i][sp][result_i] == 0) &&
21 (forall j: int, p: StackPtr ::
22   j <= i && j % 2 == 0 && 0 <= p && p <= spmap[j] ==>
23     heap[stack[j][p][param0_r], createdByCtxt]));
```

The number of interaction frames in the stack is always greater or equal to zero, so the index of an interaction frame i is also greater or equal to zero. The same holds for the stack pointers of valid interaction frames. Each variable of each stack frame references allocated objects, null is also defined to be allocated. We are mostly concerned with valid stack and interaction frames. But since we sometimes need to create new interaction and stack frames, we define the frames above and below the valid frames to be empty. Such a frame is empty if all object variables point to

null and all primitive values are 0, which also represents **false** for boolean variables. The next two properties come from the fact that we have an execution stack here. Since the execution of the methods of the stack frames below the current one is not yet finished, the result of those stack frames is set to default, null for object references and 0 for integer results. The last property is special to our specific model of context and library frames. Since we know, that every second interaction frame solely consists of stack frames of the context and therefore executes methods that are declared in a class of the context, the receiver, or the object referenced by this in Java terms, has to be created by the context. A more detailed explanation is given in Sections 2.3 and 2.4.

4.3 Relating two Library Implementations

We try to reason about the possible executions of two independent libraries at once. Therefore it is necessary to talk about the parallel execution of both libraries and the corresponding states each library is in during the execution. Boogie does not directly support parallel execution of two programs. The solution is to introduce two heaps and two stacks and relate them.

The first library is executed from an interaction between the context and the library up to the next context switch, a program point that shows observable behavior. Afterwards we execute the second library up to the point where the context switch occurs. If the second library shows compatible behavior, the operation of this library that leads to the context switch will be equal to the operation of the first library that lead to the context switch. The interactions correspond to the theoretical model of interactions introduced in Section 2.4.

Correspondence Relation

The correspondence relation is represented by the variable related, which is a bijection between objects of the library implementations. Figure 4.1 shows a sketch of the parallel execution of an old library implementation and its backward compatible modified implementation. The context of both library implementations is equal. We know that all the steps in the context up to and including the method call are equal and the objects in the context that participate in interactions are corresponding. We know nothing up front about the equality of the library steps. If we can show that the return of the library method or the next boundary call is equal, too, we can deduce that the following steps in the context are equal again.

For the correspondence relation we have to assume some properties.

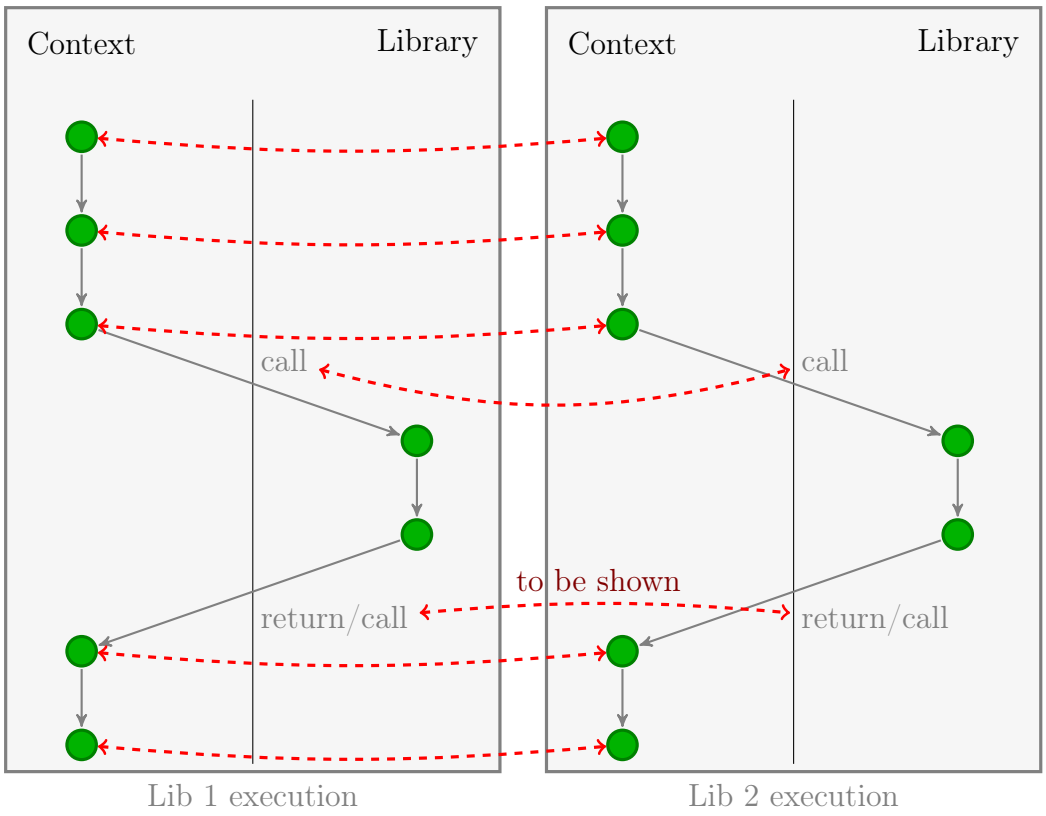


Figure 4.1: Parallel execution of both library implementations

```
1  axiom (forall heap1: Heap, heap2: Heap, related: Bij ::
       WellformedCoupling(heap1, heap2, related)
2    <==>
3  Bijective(related) &&
4  ObjectCoupling(heap1, heap2, related) &&
5  (forall r1: Ref, r2: Ref :: related[r1, r2] ==>
6      heap1[r1, exposed] && heap2[r2, exposed]) &&
7  (forall r1: Ref :: r1 != null && heap1[r1, alloc] && heap1[r1,
       exposed] ==>
8      (exists r2: Ref :: related[r1, r2])) &&
9  (forall r2: Ref :: r2 != null && heap2[r2, alloc] && heap2[r2,
       exposed] ==>
10     (exists r1: Ref :: related[r1, r2])) &&
11 (forall r1: Ref, r2: Ref :: related[r1, r2] ==>
12     (forall t: TName :: isPublic(lib1, t) && libType(lib1, t) ==>
13       isOfType(r1, heap1, t) <==> isOfType(r2, heap2, t))));
```

Since the correspondence relation holds between exactly two objects of an equal interaction, it has to be bijective. The property ObjectCoupling(heap1, heap2, related) tells us that exactly objects are related, related[r1, r2] ==> r1 != null && heap1[r1, alloc] && r2 != null && heap2[r2, alloc]. Additionally, since those objects participate in interactions and objects that participate in interactions are exposed, we say that related holds exactly between objects that are exposed. An important point is also, that those objects that are related appear in exactly the same position in the

interaction, the method call or return. Those objects should not be distinguishable from the point of view of the context, that means that all public types that are supertypes of the object in the old implementation need to be supertypes of the object in the new implementation. Otherwise those objects could be distinguished by looking at the possible casts they can be involved in.

Equivalence of Interactions

Our goal when trying to show backward compatibility of two library implementations is to show, that the interactions of both library implementations in response to each interaction of the context are equal. This is an implementation of the approach of the theoretical model introduced in Section 2.4. For this, we have to consider the boundary calls made by the implementation as well as the result of this library method in both implementations or the corresponding parameters of a boundary call performed by the library method. The behavior of these objects have to be related as well. We see that the problem of checking backward compatibility is recursive in nature. To show the equivalence of two interactions of the library implementations, we have to prove that the interactions performed by the parameters or result are equal. We consider generic invocations of all possible library methods. The coupling invariant defines how such a generic situation looks like, it describes the relation between possible states of corresponding objects. The goal is to show that this generic situation holds for every context switch.

Figure 4.2 shows a schema of an interaction frame. The Boogie model implements the concept of interaction frames as described in Section 2.4. The bottom-most interaction frame belongs to the context since the main method as the starting point of the execution is in the context. The interaction frames alternately belong to the context and the library. Most interesting for the proof is the frame created by the boundary call of the library method under examination.

In the theoretical model, for a boundary method call, the receiver of the method, referenced by **this** in Java, as well as the parameters of the library method are non-null objects in correspondence relation. Because we know the context of the libraries to execute the same steps, we know the interactions between context and library are equal and the signatures of the methods are the same. For method returns, the method from which we return are equal because of the same fact. In addition to objects, the Boogie model also supports null as well as **int**, long and boolean as primitive types. The receiver of a method call has to be a non-null object. Values of the other parameters and result values that are in correspondence relation in the theoretical model are either both null or related for reference types, class or interface types, or are equal for primitive types. For the property that two references are both null or in correspondence relation the function RelNull(Ref, Ref, Bij) is introduced. The equality for null and primitive types is a natural extension of the theoretical model. When talking about related parameters and

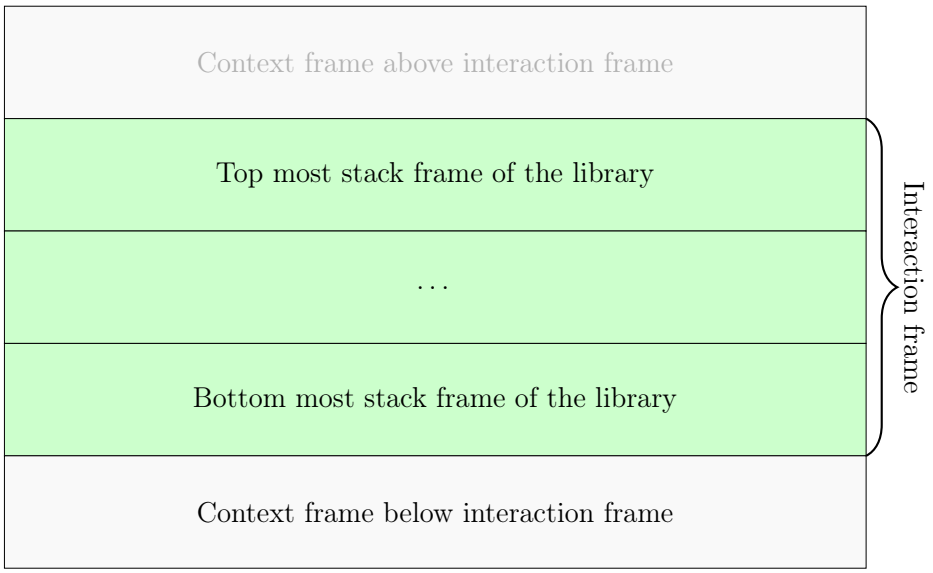


Figure 4.2: Schema of an interaction frame

return values we mean equality for parameters of primitive type, object references are either both null or refer to corresponding objects that are related.

```
1 axiom (forall r1: Ref, r2: Ref, related: Bij :: RelNull(r1, r2,
    related) <==> (r1 == null && r2 == null) || related[r1, r2]);
```

The simulation of a library implementation always reaches a context switch at some point. Either this context switch is a boundary call, a call of a method of the context, or a boundary return, the return of the method of the bottom most stack frame of the interaction frame. In both cases the control switches to the context and the behavior is visible. The objects that participate in this interaction have to be related, the behavior of these objects also has to be compatible. The check that the behavior of the objects appearing in the interaction label is compatible is performed in parallel, since all possible passes are considered using a non-deterministic choice. The non-deterministic choice between method signatures and types is described in following sections.

If the invariant is preserved by each and every execution of the library implementations between two interactions, the behavior is guaranteed to be backward compatible, as proven in the paper of the formal model [26]. Additionally, we check that the initial configuration is related for the given invariant and that steps executed in the context do not destroy the coupling invariant. The initial configuration consists of an empty heap and stack. This reduces the possibility of specification errors, inconsistent invariant definitions.

Since the observable behavior of the libraries is defined in terms of the methods they invoke on the context, these method calls have to be equal. The next sections describe how to check that two method calls or method returns are equal.

4.4 Method Calls

To prove backward compatibility of the new implementation of a library, we have to show that the visible behavior in all interaction that the first implementation could take part in is the same for the original and the new implementation. The visible behavior is highly dependent on method calls. We have to find a way to simulate all possible method calls of the context on the library and the library on the context as close to the real execution as possible. We will introduce how our Boogie model represents the methods of the library implementations, how to cope with dynamic dispatch of method calls and how to distinguish calls that cross the boundary from calls that are handled internally and are not visible for the context.

4.4.1 Representation of Method Calls

Using our approach it is impossible to translate the Java methods to Boogie procedures, as it is usually done. First of all, we have no specification of the behavior of a single method. The only thing we have, is a possible relation between two library implementations. Since Boogie only uses the specification of a procedure to prove the correctness of a caller, it is not possible to use the Boogie **call** statement, either. What we need is a way to branch from the code of one method into the code of a different method. In Boogie this is only possible using one procedure for all methods one might possibly branch into. For our approach, these are all methods of a library. Since we want to execute both library calls before checking the invariant relating the library states, we even have to include all methods of both libraries into a single big Boogie procedure.

Instance Methods

It is important for our implementation to be able to distinguish between the methods of a class. For this to be possible all method signatures are added to the Boogie model in form of unique constants of type Method. These constants include the name of the method, the types of the parameters and the type of the result. The class in which the method is declared is not part of the signature. This definition simplifies the handling of dynamic method dispatch because we can talk about all methods that have a specific signature. Together with a definition of a subtype of the class this makes for a good definition of all methods a call can dispatch to.

```
1  ...
2  var stack: Stack;
3  var spmap: [int] StackPointer;
4  var ip: int;
5  const unique meth: Method;
6  const unique $m#X$Y: Method;
7  const unique $X: TName;
8  const unique $Y: TName;
9  const unique $A: TName;
10 const unique $B: TName;
11 axiom B <: A;
12 ...
13 block_A_m:
14   assume stack[ip][spmap[ip]][meth] == $m#X$Y;
15 ...
16 block_B_m:
17   assume stack[ip][spmap[ip]][meth] == $m#X$Y;
18 ...
```

The example shows a simplified method dispatch between the method m of class A and the method m of class B. The types X, Y, A and B each are translated into unique constants of type TName. The signature of the method is translated into a unique constant of type Method, in this example $m#X$Y read method with name m which returns a value of type X and takes a parameter of type Y. Given B is a subtype of A and both have an implementation of method m we get two different blocks that represent the body of method m. The commonality of both blocks is the declaration at the start stating that the method the current stack frame belongs to is the method with signature $m#X$Y. To address both implementations we simply omit the information about the type of the receiver of the method.

Static Methods

For static methods we have a very similar situation, only we do not get the receiver of the method from the context. In place of the usual receiver we have a special object representing the class the static method is defined in. This class representative can be obtained using the function ClassRepr(TName): Ref. This function takes a type name and returns a reference to the representative of this class. For a static method call, this object replaces the usual receiver object in param0_r, the first implicit parameter of the invocation. The inverse of the class representative ClassReprInv(Ref) gives the class that is represented by the object. Applying the reverse function to the receiver gives the class the static method is defined in.

Predefined Places

There are times in the checking process when we need to talk about a specific point in the execution, for example when talking about a point at which a method is called. To support explicit points of the execution we have added a type called

Address which represents a dynamic point in the execution of a method. These dynamic points correspond to a static point in the code of a method. For each method call a new execution point is added as a unique constant. These are the predefined places. User defined places can also be added to the program which are called local places. These local places not only depend on a code point of the Java program but also on a condition with respect to the program state at this program point. More about local places in Fig. 4.11. The variable place of type Var Address represents the place the simulation of the program execution has last passed.

4.4.2 Handling of Dynamic Method Dispatch

The parameters of a method are accessible using the stack frame of the method. For example stack1[ip1][spmap[ip1]][param0_r] is the first parameter of the method, the receiver for non-static methods and the class representative for static methods. The other method parameters are represented the same way, param<i>_r is the i-th parameter of an object type and param<i>_i is the i-th parameter of a primitive type. We also assume the properties that hold for the parameters such as they are related and the receiver or representative is unequal to null.

The execution of both library implementations are simulated. Boogie needs to check all combinations of possible method dispatches to prove the behavior is equal in all possible interactions. The question is, how to translate the Java methods to Boogie for this combining to be possible.

Targets of Method Calls

We have already seen that it is possible to abstract from all possible targets of a method call using our method signature definitions. We use the internal variable meth to distinguish between implementations of different methods. An additional assume concerning the type of the receiver makes sure we only branch into the implementation if we assume a receiver with the right type. An artificial block for each library implementation, called the dispatch block, holds references to every method implementation encountered in the library. A simplified example can be seen in Fig. 4.3.

As we see in Fig. 4.3, for every method in classes A and B a starting block of the implementation of the corresponding method is added to the Boogie program. The body of the methods are omitted in this example. The information about the type of the receiver, param0_r on the stack, in addition to the method that is owner of the stack frame, stack1[ip1][spmap1[ip1]][meth], guides the proving process. Since the preconditions state that the method of stack frame 0 has to be equal for both library implementations, only valid combinations can be chosen by the prover. The branch of the proving process that leads through such a block can only be taken if the type of the receiver can be assumed to be a subtype of A

```
public class A{
        public X1 m1(Y1){...}
        public X2 m2(Y2){...}
                ...
        public Xk m<k>(Yk){...}
}
public class B{
        public X<k+1> m<k+1>(Y<k+1>){...}
                ...
        public Xn m<n>(Yn){...}
}
```

```
...
dispatch1:
  goto lib1_A.m1#X1$Y1, lib1_A.m2#X2$Y2, ... , lib1_A.m<n>#Xn$Yn;
lib1_A.m1#X1$Y1:
  assume isOfType(stack1[ip1][spmap1[ip1]][param0_r], heap1, $A);
  assume stack1[ip1][spmap1[ip1]][meth] == $m1#X1$Y1;
  ...
lib1_A.m2#X2$Y2:
  assume isOfType(stack1[ip1][spmap1[ip1]][param0_r], heap1, $A);
  assume stack1[ip1][spmap1[ip1]][meth] == $m2#X2$Y2;
  ...

...

lib1_A.m<k>#Xk$Yk:
  assume isOfType(stack1[ip1][spmap1[ip1]][param0_r], heap1, $A);
  assume stack1[ip1][spmap1[ip1]][meth] == $m<k>#Xk$Yk;

...

lib1_B.m<k+1>#X<k+1>$Y<k+1>:
  assume isOfType(stack1[ip1][spmap1[ip1]][param0_r], heap1, $B);
  assume stack1[ip1][spmap1[ip1]][meth] == $m<k+1>#X<k+1>$Y<k+1>;

...

lib1_B.m<n>#Xn$Yn:
  assume isOfType(stack1[ip1][spmap1[ip1]][param0_r], heap1, $B);
  assume stack1[ip1][spmap1[ip1]][meth] == $m<n>#Xn$Yn;
  ...
```

Figure 4.3: Simplified example of the dispatch blocks of a library implementation

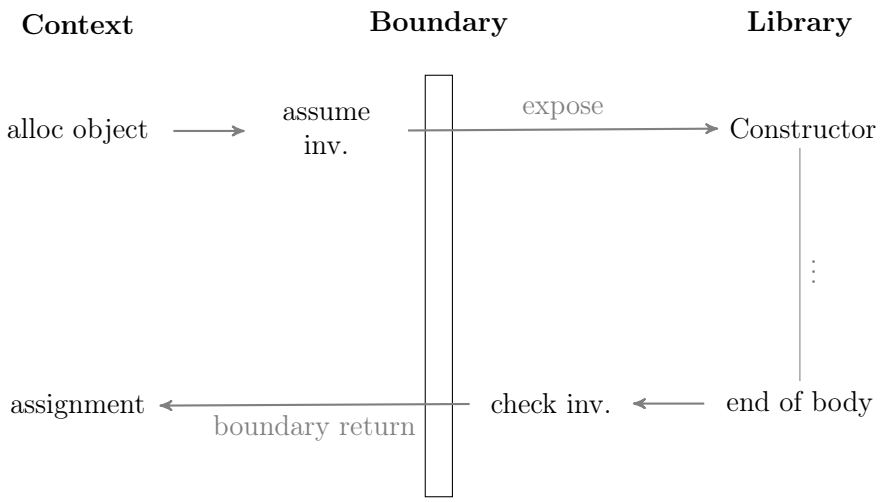


Figure 4.4: Boudary calls to constructors

or subtype of B respectively and the method currently executed can be assumed to be the method annotated, such as for example $m1#X1$Y1. Simulations that would lead to a contradiction at this point are abondoned.

Constructor Calls

Boundary calls to constructors are handled much like normal method calls. Since the objects we expect in the simulation of both library implementations are constructed by the same new statement, we know the dynamic type of the object to be identical typ(stack[ip1][spmap1[ip1]][param0_r])==typ(stack2[ip2][spmap2[ip2]][param0_r]). The big difference between normal method calls and constructor calls is that we expect the receiver object, the object to be constructed, to be not yet exposed and created by context, but the coupling invariant is expected to hold only for exposed objects. The situation is illustrated in Fig. 4.4. The idea behind this construction is that the constructor should initially establish the coupling invariants of objects being created. It is legal to assume the parameters given to the constructor to be completely constructed objects. We expect the class *java.lang.Object* to be part of the library and every object in Java has this class as top of the inheritance hierarchy, as does the theoretic model. Based on this fact we can assume every object created by the context to be exposed, since the constructor of the class *java.lang.Object* has to be called at some point of the creation process. The objects created by the library can also be assumed to be exposed when given as a parameter to the constructor called by the context, since these objects will have to be exposed before. Summarizing all these facts, the parameters of the constructor are completely created and exposed objects and the object to be constructed is created by the context and not yet exposed when passed to the constructor implementation.

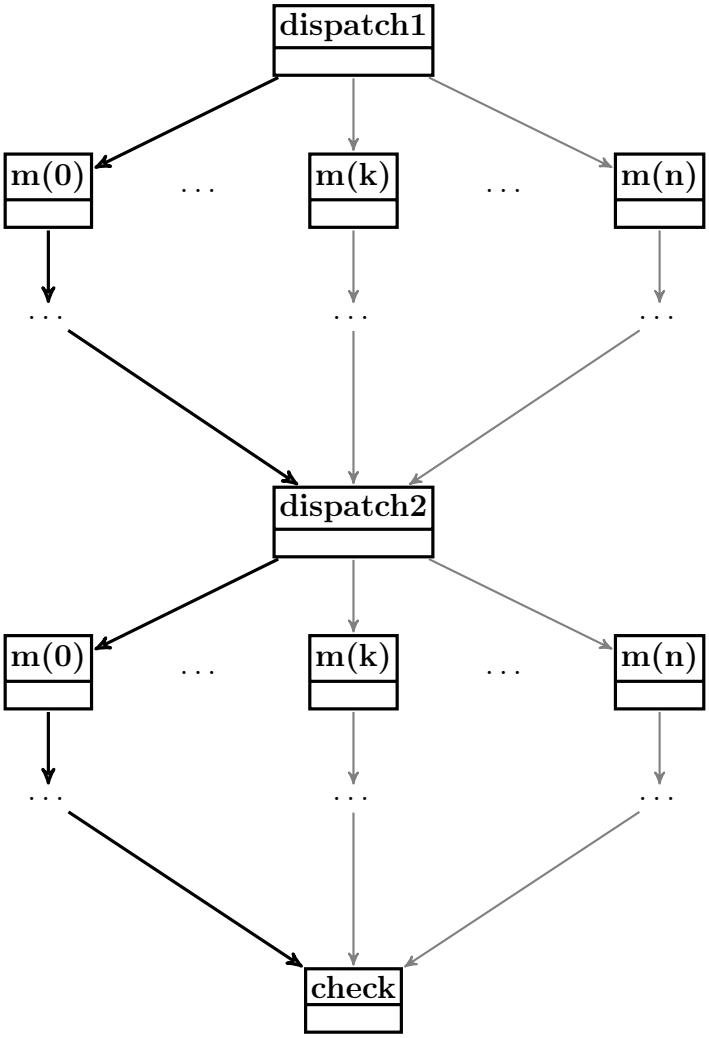


Figure 4.5: Overview of the handling of method dispatch

This construction allows us to check the coupling invariant for all but the non-constructed objects and still this unfinished object can be exposed immediately after checking the invariant, which is important for the rest of the simulation and proving process. Compared to a methodology using a flag to signal an object is not yet initialized, our approach allows us to handle object construction the same way as normal method calls. This is especially true for boundary method calls in constructors. Since these calls are also interactions between the library and the context, the coupling invariant has to hold before the control is passed to the context again. An object can only assume to be fully initialized after the constructor has finished. Using a flag to signal unfinished object initialization, we would have to specially handle this case. In our approach, this object under consideration is assumed to be exposed, so the coupling invariant is checked as in all other cases of boundary method calls. This also means that the creation of the object will have to establish the coupling invariant before calling a method on the context, whereas it is possible to call internal methods without first establishing

this invariant. We see, the rest of the proving process for constructors does not differ from those of the remaining method invocations.

Method Dispatch

Figure 4.5 shows an overview of the method dispatch. The block dispatch1 branches to all possible method blocks non-deterministically. The path taken is selected using the information given by the meth variable together with the information about the parameters. Since this information is given for the parameters of both implementations, all parameters are related, the path taken by the prover in both disptach1 and dispatch2 is the same.

Guiding the proving process works by under-specifying the type of the receiver. By using an assume-statement of the form isOfType(stack1[ip1][spmap[ip1]][param0_r], heap1, $A) we specify that the type of the receiver is a subclass of A. So all methods declared in subclasses of type A are addressed this way. What we did not consider yet is that methods may be overridden. When a class B extends A and overrides method m, then whenever m is called on B the implementation of m in B is used. Using only the isOfType definition, we cannot rule out the case that the method defined in class A is used instead. We need a representation of whether a method is member of a specific class and in which class the corresponding implementation can be found.

```
1  axiom (forall l: Library, m: Method, c1: TName, c2: TName ::
       memberOf(l, m, c1, c2)
2    <==>
3  (c1 == c2 && definesMethod(l, c2, m)) ||
4  (!definesMethod(l, c2, m) && (forall c3: TName ::
5      classExtends(l, c2, c3) ==>
6        memberOf(l, m, c1, c3))));
7
8  axiom (forall l: Library, m: Method, c1: TName, c2: TName ::
9      memberOf(l, m, c1, c2) ==>
10       definesMethod(l, c1, m));
11
12 // memberOf is a relation (could also be defined as function
       definedInClass(l, m, c2) == c1)
13 axiom (forall l: Library, m: Method, c1: TName, c2: TName ::
14     memberOf(l, m, c1, c2) ==>
15       (forall c3: TName :: c3 != c1 ==> !memberOf(l, m, c3, c2)));
16
17 // memberOf only talks about class types
18 axiom (forall l: Library, m: Method, c1: TName, c2: TName ::
19     memberOf(l, m, c1, c2) ==>
20       isClassType(l, c1) && isClassType(c2));
```

The property memberOf(l, m, c1, c2) expresses that in library implementation l method m is member of class c2 and the implementation of the method can be found in class c1. The definition is analogous to the member-of property of the theoretical model described in Section 2.3. Method m is member of class c2 if it is defined in it. A method is defined in a class if the method is implemented in the class. The

second possibility is that the method is not implemented in c2 but it is member of the superclass of c2, called c3 and there is no class A in between c2 and c1 in the inheritance hierarchy such that A has a different implementation of m. Class c1 is the class in which the implementation of m can be found. The property could also be expressed as a function definedInClass(l, m, c2) == c1 which gives the class in which the method is defined, but Boogie has problems using this function to determine the implementation of a method. So we decided to use the function memberOf instead and added a axiom stating that memberOf(...) is injective with respect to the third argument. Since we are only interested in the method body we want to simulate, we are not concerned with interfaces. A class implementing the interface has to implement the corresponding methods of the interface.

4.4.3 Boundary and Internal Calls

In Section 4.3, we have seen that we can use the coupling invariant to check that the observable behavior of both library implementations is equal. For this the coupling invariant is checked at each interaction. For this approach to work, the interactions, so the boundary method calls, must be equal. In this section we describe how this check is implemented in Boogie.

When looking at a method invocation we can distinguish two different cases. The method called might have an implementation in the library. In this case we know the code that will be executed and the check can continue. There is no context switch involved and no interaction happens, the invariant does not have to be checked. The second case is a boundary call, the implementation of the method invoked is in the context. In this case we do not know the implementation and this invocation is an interaction between the library and the context, with all its consequences.

Java supports subtyping with method overriding and dynamic method dispatch. The implementation that is executed by a method invocation can only be determined at runtime. We cannot statically tell if a method invocation leads to an internal call or a boundary call based on the information gathered from the Java byte code. Generally we have to consider both possibilities.

As for method returns the situation is quite similar. The point a method might return to, is not statically clear since there is generally more than one point in the program the method is called. The dynamic method dispatch plays a role here as well. Since there are multiple possible methods for a method call, there are also multiple possible methods that can return to that specific program point.

Targets of Method Returns

When the execution reaches a point where a method will return, we have to figure out if this method return is a boundary return or an internal return, similar to

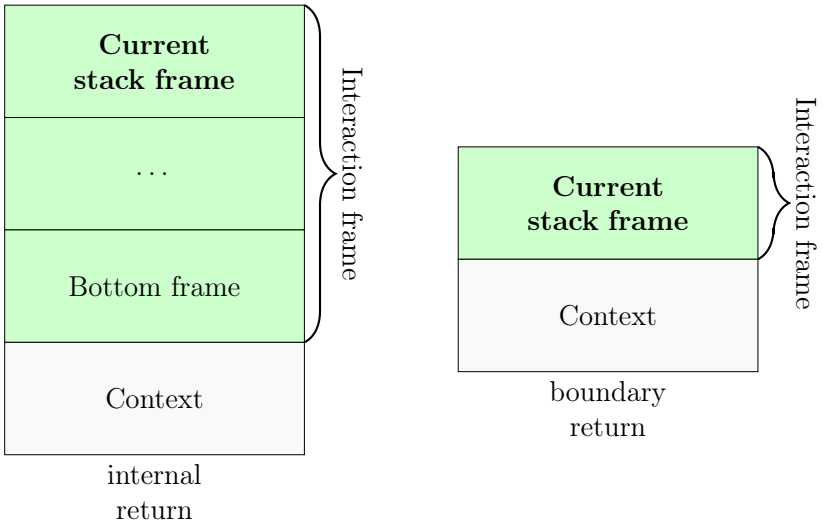


Figure 4.6: Possible situations for method return

method calls. For returns, the interaction frames come in handy. If the interaction frame belongs to the library and only consists of one single stack frame when a method returns, we know that the frame beneath this stack frame is a stack frame that belongs to the context. This method return is a boundary return. In this case it holds that ip1 % 2 == 1 && spmap1[ip1] == 0 && ip2 % 2 == 1 && spmap2[ip2] == 0. ip1 represents the index of the current interaction frame of the first implementation, ip2 the index of the interaction frame of the second implementation. The interaction frames belong to the library, ip1 % 2 == 1 and ip2 % 2 == 1. spmap1[ip1] == 0 and spmap2[ip2] == 0 expresses that the top stack frame of the current interaction frame has index 0, there is only a single stack frame in the interaction frame. On the other hand, if there is more than one stack frame in the interaction stack, we know that there is still at least one stack frame beneath the current one that belongs to the library. In this case it holds that ip1 % 2 == 1 && spmap1[ip1] > 0 && ip2 % 2 == 1 && spmap2[ip2] > 0. We can never have the situation that the stack frame belongs to the context when returning from a library method, since this would mean that a method of the context would return. Figure 4.6 illustrates these situations.

For finding the right point to return to, we have to consider the method currently returning, the callee of this method and we have to figure out if the corresponding call could have been made. For internal calls, the problem can be solved using our places concept. When a method call happens, the place variable of the current stack frame is set to a unique constant that stands for this particular point of the program. At the point the called method returns, the place variable is assumed to be equal with this constant. This means that a called method returns always to the point it was called at.

For boundary returns the situation is more complex. If the call of a method in the context and the corresponding return are split into two independent verification passes, all information about the stack and the situation on the stack is lost.

Generally we can only assume the facts that always have to hold for the situation of a context method returning into the library, such as for the current interaction frame pointer ip we know that ip % 2 == 0, since we return from the context and that there is at least one interaction frame of the library beneath the current one, since we return into the library. The additional information needed to reason about where the method could return to other than the name of the method and that the method has to be member of the class of the receiver, has to be given as part of the coupling invariant. The place variable can be used to specify facts that hold at specific points of the execution, in this case at the return point of the corresponding method call. Patterns used to specify such properties are presented in Section 4.7. In subsection Retaining Internal State, we will present a technique that preserves the information about the stack even between the boundary call and the boundary return.

Choice Between Internal and Boundary Calls

What we still have to focus on is how to find out if a method call is an internal call or a boundary call. Since the interactions made by both library implementations need to be equal, the decision of whether a method invocation leads to a boundary call or not needs to be based on the information we have about the state of the program. If the choice would not be deterministic, one possible simulation would be that one library implementation makes a boundary call while the other makes an internal call. This would most probably lead to incompatible interaction traces and therefore to a failed verification attempt. On the other hand, it is possible to allow both internal and boundary calls, if we have a boundary call in the old library implementation whenever we have such a call in the new implementation, and vice versa.

To choose between a boundary and an internal call, we have to find out if the callee could have a type that has an implementation in the library and that the method can be called on. If this is the case, we could have an internal call, if this is not the case than it is impossible to have an internal call. For the method call to be a boundary call, we need a type that is not a library type, meaning whose implementation is not in the library, and on which the method can be called. If it is possible to call a method m on a type t then m is a member of t.

In the example shown in Fig. 4.7 we assume a method m should be called. Because we have to consider an internal as well as a boundary call, we non-deterministically branch to both call blocks block_intern and block_boundary. An internal call leads to a new stack frame in the current interaction frame, a boundary call leads to a new interaction frame with a single stack frame, as defined in the theoretical model. The parameters of the method call are written to the new stack frame accordingly. So the callee of the method is param0_r on the current stack frame. The next statement of these blocks checks whether a possible library type or a possible non-library type exists that implements m. This check uses the fact

```
1   ...
2   //prepare method call
3   stack[ip][spmap[ip]][meth] := $m;
4   ...
5   goto block_intern, block_boundary;
6 block_intern:
7   <create new stack frame in current interaction frame>
8   <write parameters into new stack frame>
9   assume (exists t: TName :: memberOf(lib, $m, t,
        typ(stack[ip][spmap[ip]][param0_r], heap)) && libType(lib, t));
10  goto calltable;
11 block_boundary:
12  <create new interaction frame and first stack frame>
13  <write parameters into new stack frame>
14  assume (exists t: TName :: memberOf(lib, $m, t,
        typ(stack[ip][spmap[ip]][param0_r], heap)) && !libType(lib, t));
15  goto check;
16 ...
```

Figure 4.7: Blocks for internal and boundary calls

the m is member of the receiver object and an implementation in a supertype of the type of this receiver object. For each type Boogie knows if it is a library type or not. The internal call branches to the part of the dispatch table, that includes only the starting blocks of the method implementations of the library methods. The boundary call block branches to the coupling invariant check. The right path to all possible method implementations is chosen as described earlier in Section 4.3.

Retaining Internal State

As seen earlier, normally the checking process of a boundary call into the context and its corresponding boundary return are handled independent from each other in two different passes. All information about the stack, such as type information, aliasing and so on, is lost because of this. In the Boogie implementation this leads to the problem, that all the lost information needed for the further proof needs to somehow be re-added. The concept of places makes it possible to tell Boogie these facts by adding them to the coupling invariant. But doing this results in a coupling invariant that consists of a large amount of facts about the stack that could have been deduced from the program during simulation and is lost afterwards. We came up with a way to simulate boundary calls without loosing the information about the stack, but at the same time we are still sound in our verification method.

We handle boundary calls much like internal calls. Because we do not have the implementation of the boundary method for our simulation, we over-approximate the effect the context method could have had on the program state. We know, that the method cannot have changed any stack frames beneath its own. The values of local variables remains the same for integer and boolean variables. Variables of a

class- or interface type retain the reference they are pointing to. All other changes such as changes to the fields of an object referenced to by a local variable are changes of the heap. The interaction frame remains untouched. The remaining program state, the heaps of the library implementations, is **havoc**ed, all information is lost. Even though we do not know the exact effect of the context method on the heaps, we know that some of the information valid in the heap before the boundary call is still valid after the method execution has finished.

When an object is created, it starts as an object which is not exposed. During the execution, this object can be exposed for example because it was a parameter of a boundary call. Once an object is exposed it stays exposed. Let oldHeap be a reference to the heap before the boundary call and newHeap be a reference to the heap after the boundary call. We know that oldHeap[o, exposed] implies newHeap[o, exposed]. The context can create new objects and allocate them on the heap. Already allocated objects are assumed to never be deallocated. Additionally we know that the flag createdByCtxt is set only when the object is allocated. We can deduce that oldHeap[o, alloc] implies newHeap[o, alloc] and oldHeap[o, alloc] implies (oldHeap[o, createdByCtxt] == newHeap[o, createdByCtxt]). An object is created of a specific dynamic type. This dynamic type is also set when creating the object. This is why oldHeap[o, dynType] == newHeap[o, dynType]. These facts are the basis for the property ValidHeapSucc describing a valid heap successor.

```
1  axiom (forall oldHeap: Heap, newHeap: Heap, stack: Stack ::
2    ValidHeapSucc(oldHeap, newHeap, stack)
3      <==>
4    (forall sp: StackPtr, o: Ref ::
5      (oldHeap[o, exposed] ==> newHeap[o, exposed]) &&
6      (oldHeap[o, alloc] ==>
7         oldHeap[o, createdByCtxt] == newHeap[o, createdByCtxt]) &&
8      (oldHeap[o, alloc] ==> newHeap[o, alloc]) &&
9       oldHeap[o, dynType] == newHeap[o, dynType]
10   ));
```

After **havoc**ing the heap in preparation of a boundary call, the ValidHeapSucc property is assumed to express that the heap retains all information that is still guaranteed to be valid after the context method has returned. Afterwards we assume the coupling invariant to hold and jump to the return table where all possible return points of method calls are listed. Finding the right path to the return point is based on the place variable, as described earlier.

Aliasing Between Interaction Frames

When the execution of both library implementations reaches a boundary call, the coupling invariant is checked, the program state is reset and the invariant is assumed again before continuing the execution after the boundary call returns. The method executed by the boundary call may change the program state, because of aliasing even local objects may be affected. For this approach to be sound, the

```
public class C {
    public int m(A a) {
        a.f = 0;
        a.p();
        return a.f;
    }

    public void n(A a) {
            a.f = 1;
    }
}
```

```
global invariant:
  when at place C.m_p_call ==> a.f == 0
```

Figure 4.8: Example of an indirect destruction of a global invariant

properties of the invariant have to be formulated in such a way that they are independent of the current stack frame.

Figure 4.8 shows an implementation where the independence of the current stack frame is important. In the body of method m the field f of the local variable a of type A is set the value 0. Directly after the assignment a boundary call to method p happens. The check of the invariant that when the execution reaches the call to p the field f of a is 0 will always succeed. In addition to m, the method n also changes the field f of an object of type A to the value 1. The context method p may call n, which indirectly affects the state of the object referenced by the local variable a in method m. After the boundary return of p, the invariant is assumed, especially the property that the field f of a is equal to 0. This invariant could indirectly be destroyed by the boundary call.

An incorrectly formulated invariant could lead to an inconsistency in this situation. The property that a.f == 0 has to be assumed for each top-most stack frame of each interaction frame that is at the boundary call to p. To consider the top-most stack frame is sufficient, because a boundary call will always lead to a new interaction frame. Using this pattern and considering multiple interaction frames in the check, the destruction of the invariant will be recognized in the boundary return check of method n, the method that destroys the property. The complete pattern that is used to formulate the invariant is shown in Section 4.7.

Static Methods

For static methods and constructors, we have a different situation. Since those methods are statically bound, we do not need to distinguish boundary and internal calls in this case. Having a static method call in a method of the library means we know the concrete implementation. Such an invocation will always be an internal

call, because the only implementation we are aware of is that of the library and we are simulating a method of the library implementation. There is no context switch involved in a static method call. The simulation of the invocation can directly jump to the right implementation blocks, bypassing the dispatch table. A return to the right program point has to be handled by the return table, nevertheless. The remaining verification process is handled as presented, especially for method calls inside constructors and static methods.

We have seen that the checking of method calls highly depends on properties of the type system of Java. To distinguish between internal and boundary calls, we need to know about possible subtypes and which methods can be overridden by the context. The next section presents the axiomatization we chose of the Java type system.

4.5 Axiomatization of the Type System

The situation we encounter when comparing two library implementations is not natively supported by Boogie. We need to handle the types of both implementations. For public types, we know from the fact that we have source compatibility that each public type of the old implementation has to be contained in the new implementation. Additionally, for related objects each public supertype in the old implementation has to be public supertype in the new implementation. For non-visible types, such as types defined to be package visible or private, we can have very different subtype relations. We need to somehow distinguish between the types of the first and the second implementation of the library.

Separation of Type Hierarchies

One possibility would be to use two distinct Boogie types for the names of the Java types in the implementations, TName1 and TName2. The order $<:$ supported by Boogie is defined for each Boogie type separately, so the subtype relation would be separated as well. This would also mean that we had to duplicate all functions that remotely have to do with types, such as WellformedHeap(...), isOfType(...), and many more. Additionally the type names would have to be prefixed to make the constants we introduce for each Java type in the generated output unambiguous.

Another possibility would be to simply prefix the type names and stick with a single Boogie type TName representing Java types. Using this approach we could also use single instances of the functions dealing with types and we could also use the Boogie built-in order $<:$. On the other hand, we would not have a strict separation of the subtype relations of both library implementations. We could simply use the order $<:$ to relate a type of one implementation with a type of the other implementation, but a type of one implementation should never be subtype of a type of the other implementation. Because we do not want to specify, which

types are not related with each other for each type in each implementation, which would lead to quadratically many negative relation axioms, this could lead to wrong reasoning about the subtypes.

The solution we chose is to introduce a ternary subtype relation, to parameterize the subtype relation with the implementation this relation is valid in. We added constants lib1 and lib2 of type Libary which represent the first and second library implementation. For classes A and B, instead of using B <: A to defined that B is a subtype of A we use subtype(lib1, B, A) to define that B is subtype of A in the first library implementation. Using this approach we avoid duplication of the function as described above. Instead we get a new parameter to these functions, the library implementation. Since heap1 and heap2 are defined to represent the heap of the first and the second library implementation, if one of these heaps is already a parameter of a specific function then we can deduce the library implementation from the heap using the function libImpl(Heap)**returns** (Library).

Library Types

Java class, interface, and primitive types are represented as unique Boogie constants of type TName. For all classes we encounter during the translation process we add axioms stating that those classes are libTypes. Because we use the closed form

```
1 axiom (forall t: TName :: libType(lib1, t) <==> t == $C1 || t == $C2
    || ... || t == $Cn);
```

for C1 ... Cn are all class types encountered in implementation one, we also state that those are the only types in the library implementation. This information will be used by Boogie to decide about boundary or internal method calls as described in Section 4.4.

For all class and interface types c that are defined to be public we generate an axiom stating isPublic(l, c) for library implementation l and public class c. For all other types we generate an axiom !isPublic(l, c). We need this information to talk about visibility of types and methods.

We do not only what to consider specific types. Instead we have to consider also the subtype relation between those types defined in the library.

4.5.1 Subtype Relation

To reason about subtypes we first need knowledge about which types may actually have subtypes. For value types, the Java primitive types, we know that there are no possible subtypes and that those types have no possible supertypes. They stand alone in the type hierarchy with no relation to other types. This is exactly what the following axiom tells us.

```
1 axiom (forall t: TName :: isValueType(t) ==>
2   (forall l: Library, u: TName :: subtype(l, t, u) ==> t == u) &&
3   (forall l: Library, u: TName :: subtype(l, u, t) ==> t == u));
```

The top of the type hierarchy is the type java.lang.Object. We cannot add any type to a Java program such that there is a type which is a supertype of java.lang.Object. We express this fact by the following axioms.

```
1 axiom (forall t: TName :: subtype(lib1, $java.lang.Object, t) ==>
2   t == $java.lang.Object);
3 axiom (forall t: TName :: subtype(lib2, $java.lang.Object, t) ==>
4   t == $java.lang.Object);
```

The axiom tells us that every type t that is above java.lang.Object in the type hierarchy is again java.lang.Object. Since the subtype relation is a partial order and as such reflexive this property expresses that java.lang.Object can have no supertypes.

Direct Superclass

We sometimes have to talk about the direct supertype of a given type. This is not easily possible using the subtype relation. We have added a function that relates two class types if one is the supertype of the other.

```
1 function classExtends(Library, TName, TName) returns (bool);
2
3 // classExtends is a partial function
4 axiom (forall l: Library, c1: TName, c2: TName ::
5   classExtends(l, c1, c2) ==> (forall c3: TName :: c2 != c3 ==>
      !classExtends(l, c1, c3)));
6
7 axiom (forall l: Library, t: TName ::
8   !classExtends(l, $java.lang.Object, t));
```

classExtends(l, C1, C2) means that the class type C2 is the supertype of C1, in Java we write **class** C1 **extends** C2{...}. The axiom in line 4 states that the direct superclass relation is unique, so there is no type that has two or more direct superclasses. Again we have the exception that java.lang.Object does not have any supertype at all, as stated in line 6.

The axioms for the subtype relation and the classExtends relation are generated from the class file of the Java class. The connection between those two relations is made by the following axiom.

```
1 axiom (forall l: Library, c1: TName, c2: TName ::
2   classExtends(l, c1, c2) ==> subtype(l, c1, c2));
```

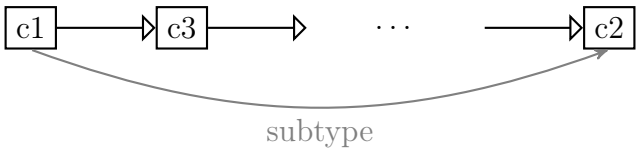


Figure 4.9: Single inheritance of classes in Java

The axiom states that if c1 extends c2 then c1 is a subtype of c2. The other direction is not correct since subtype is transitive but classExtends is not.

The relation classExtends can be used to describe the single inheritance hierarchy of classes. To express that we only want to talk about classes in this regard we use the function isClassType to distinguish classes from interfaces. Using this property we can express the afore mentioned fact.

```
1 axiom (forall l: Library, c1: TName, c2: TName :: isClassType(l, c1)
    && classExtends(l, c1, c2) ==> isClassType(l, c2));
```

The axiom states that if c1 is a class and c1 extends c2 then c2 is again a class. In the next step we want to describe that Java uses single inheritance. We can use the classExtends relation to describe the possible cases for two classes to be in subtype relation.

```
1 axiom (forall l: Library, c1: TName, c2: TName ::
2   subtype(l, c1, c2) &&
3   isClassType(l, c1) &&
4   isClassType(l, c2) ==>
5     c1 == c2 ||
6     classExtends(l, c1, c2) ||
7     (exists c3: TName :: isClassType(l, c3) && classExtends(l, c1,
        c3) && subtype(l, c3, c2)));
```

Let c1 and c2 be class types and c1 is a subtype of c2. There are three possible cases: (1) c1 and c2 can be the same type since the subtype relation is reflexive. (2) The class c1 can directly extend the class c2. (3) If neither (1) nor (2) is the case then we know that there is a class c3 in between the classes c1 and c2 in the inheritance hierarchy, such that c1 directly extends c3 and c3 is subtype of c2. The situation of (3) is illustrated in Fig. 4.9.

Final Classes

The last property we want to express about the subtype hierarchy is that a class can be final. Final classes can have no subtypes, these classes make for a lower end of the inheritance relation. The following axiom expresses this fact for a final class A.

```
1 axiom (forall l: Library, t: TName ::
2   subtype(l, t, A) ==> t == A);
```

4.5.2 Relation Between Types and Heaps

In our representation of the program state we have a representation of the heap of a program. On the heap we do not only have fields that are defined in classes, but we also have special internal fields that do not appear in any definition of a Java class. One of these fields is the field dynType. For every object on the heap this field indicates the dynamic type of the object which is set during the allocation of an object. We introduced a special function typ(o, heap) to retrieve the dynamic type of the object o from the heap heap.

To express that some object is of a specific type we introduced the function isOfType(o, heap, t). An object o is of type t if it is null or the dynamic type of o is a subtype of t.

```
axiom (forall o: Ref, heap: Heap, t: TName :: isOfType(o, heap, t)
  <==>
o == null ||
subtype(libImpl(heap), typ(o, heap), t));
```

For primitive types we can only give a range of values that is valid for the type.

```
axiom (forall i: int :: isInRange(i, $byte)
  <==> -128 <= i && i <= 127);
axiom (forall i: int :: isInRange(i, $short)
  <==> -32768 <= i && i <= 32767);
axiom (forall i: int :: isInRange(i, $char)
  <==> 0 <= i && i <= 65535);
...
```

In Boogie, an integer is not bounded. The function isInRange assigns a range of valid values to Boogie integer which represent Java primitive types.

Non-Instantiatable Types

In Java, we are talking about two kinds of type definitions. Interfaces include the signature of methods to be implemented by all classes that implement the interface. Those interfaces do not include an implementation of their method signatures. Such an interface type cannot be the dynamic type of any object on the heap. The same is true for abstract classes, which can also not be instantiated. The property WellformedHeap is extended as follows:

```
axiom (forall heap: Heap :: WellformedHeap(heap)
  <==>
...
... &&
(forall o: Ref :: !IsMemberlessType(libImpl(heap), typ(o, heap)));
```

The first part of the property WellformedHeap is the same as before. The property about memberless types is added in addition. Every interface and abstract class that is encountered during translation of the Java byte code is defined to be a memberless type. As such, a memberless type may not appear as dynamic type of an object on the heap, which is what the given axiom tells us.

4.6 Verification of Loops

Programs without loops and recursion are very limited with respect to what they can do. Interesting calculations often include repeated computations that are reduced into one single result. To support such calculations we have to consider supporting loops. But supporting repeated computation with unknown maximum iteration count is rather complex. For infinite loops, the behavior of the infinitely looping implementation is not defined. If the first implementation loops forever, the behavior of the second implementation is not relevant, since the behavior could not be proven to be not backward compatible with the behavior of implementation one. If the behavior of the first implementation is defined, the behavior of the second implementation must be equal. The solution in a specification based setting is to use a loop invariant to constraint the effect of a loop. We want to adapt this idea to fit into our verification technique of backward compatibility.

Loops in programs we consider can be distinguished into two kinds. A loop can include a boundary call in its body therefore performing interactions. On the other hand we do not only want to consider loops with interactions, but also loops that only calculate some value or change some internal state of the library. These loops have to be treated in a different way, since we are not yet concerned with non-visible computations.

The idea of classical loop verification is to have a single invariant that holds for each and every loop iteration and as such can be used to approximate the computation of the loop. When looking at simulation, this can be seen as simulating a generic iteration of the loop. What is to be considered as being a generic state of the program when entering and leaving a loop, has to be given in the form of the loop invariant. When comparing the specification based approach we have already seen in Section 3.2.4 with what we have up to now, we see many commonalities.

Loops with Boundary Calls

Figure 4.10 outlines a comparison between a loop invariant based verification and our coupling variant based verification of loops. The classical method considers all possible executions of the loop. Before entering the loop, the loop invariant is checked. After throwing away all information about the state of the program, a generic execution of the loop and the execution after the loop is simulated and all asserts on these ways are checked accordingly. When not using our optimization of boundary calls, splitting the verification into multiple passes, we have a very

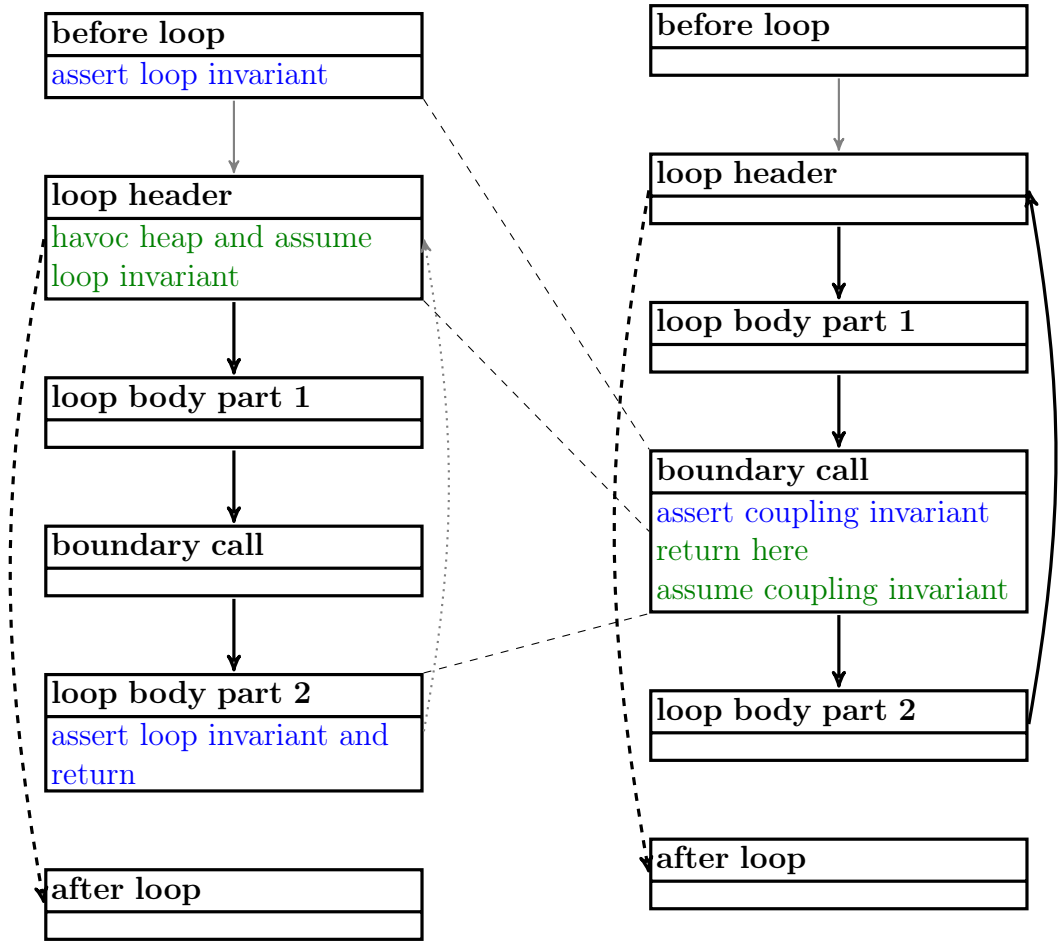


Figure 4.10: Comparison of loop verifications: Classical vs. boundary call

similar situation. When the simulation encounters a loop, it takes both branches, one into the loop body and one to the end of the loop. So the execution without entering the loop is considered in this way. After entering the loop, the simulation stops at some point when encountering the boundary call and checks the coupling invariant. The loop invariant can be seen as part of this coupling invariant. Another simulation starts after the boundary call returned, so in the middle of the loop body, and simulates a generic loop execution up to the boundary call of the next iteration, where the coupling invariant and thus the loop invariant is checked again. Additionally the execution branches to after the loop and resumes normal program execution. We see, the execution after the loop is checked twice. Other than this we have a very similar behavior compared to the classical approach.

For boundary calls in loop bodies, we know that the calls need to be related. This is what is checked by the coupling invariant check as described in Section 4.4. Since the interactions of both implementation executions need to be equal, the same method has to be called by the second implementation whenever it is called in the first implementation and vice versa. It is reasonable to always relate those places the method is called. The proving process fails if the method is called on

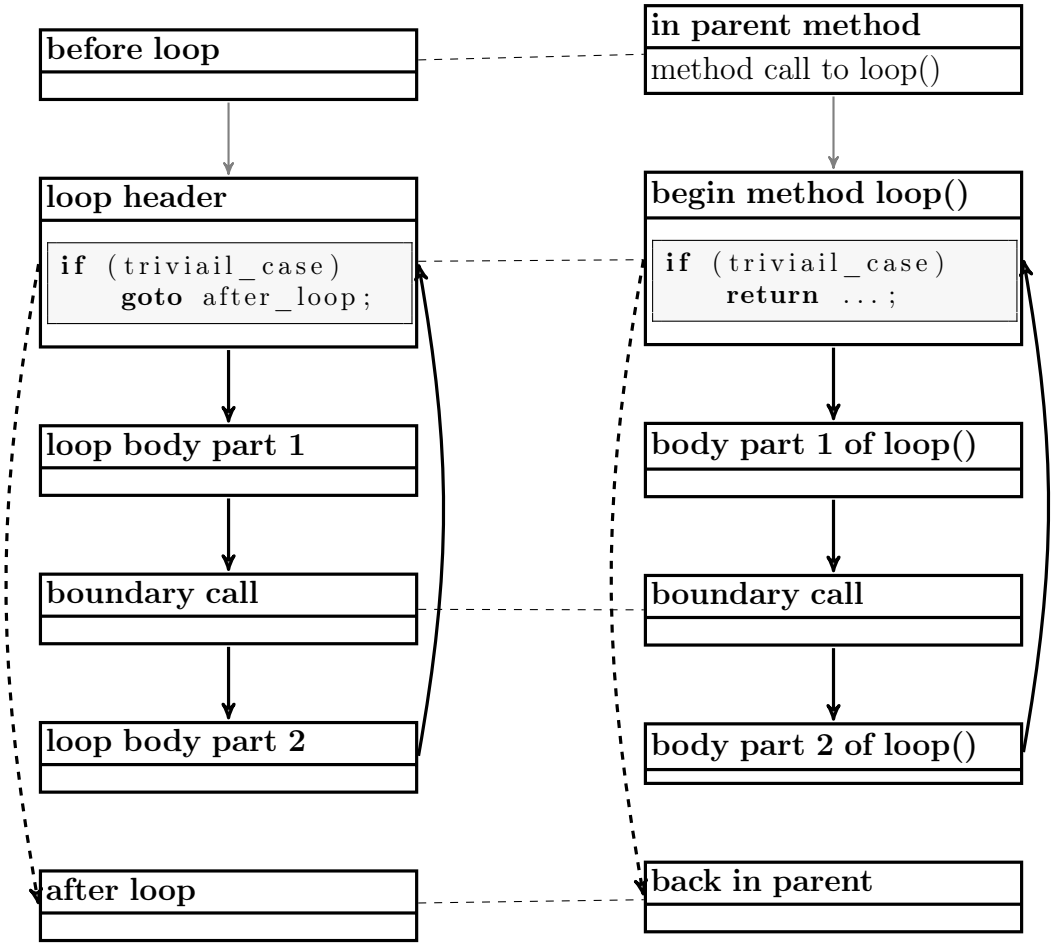


Figure 4.11: Comparison of a loop implementation and a recursive function

one side but not on the other, since this is part of the checking procedure.

Recursive Methods

The more complex case is the relation between a loop and an internal recursive method that result in the same computation. The place of the method call is in the body of a loop in one library and in the body of a recursive method in the other library. The idea is the same in this case as for the two loops example. One pass simulates the execution into the method. Since a recursive method normally has a trivial case that marks the termination of the recursion, this case is also simulated by the proving process as well as the more complex case of entering the computation body of the method up to the boundary call. The generic execution of the loop body can be compared to the generic execution of the recursive method, the recursive call of the method and the afterwards following boundary call. In this case we also have the termination of the recursion, which can be compared to a branch to the end of the loop.

Figure 4.11 shows a comparison of an execution of a loop and an execution of a comparable recursive implementation. Preconditions for this comparison is that the execution the loop performs can be compared to the execution the recursive method loop() performs. On the left hand side, the simulated method enters into a loop which has a boundary call as part of the loop body. On the right hand side, the implementation uses the recursive method loop() to perform the same computation. The method is called in the body of the parent method. The loop header and the beginning of the recursive method can be compared since each has a very similar condition which decides, if the recurring execution should be restarted or the abort case has been reached. The abort case leads back to comparable points in the execution by breaking out of the loop and returning form the recursive method, respectively.

The big difference relating a loop implementation and a recursive implementation compared to relating boundary calls in sequential programs or relating two loop implementations is that the recursive methods have built up a list of stack frames, which is removed from the stack after termination of the recursion. So for n iterations of the loop, we have n stack frames that have to be returned from. We can look at this situation as if the execution of the recursive implementation continues while the execution of the loop stalls at the end of the loop. Asynchronous execution is introduced further down.

Loops Without Interactions

We do not always have the situation that we have a boundary call inside every loop. We still want to verify the compatibility of the behavior of two loop implementations. If the behavior of the loop depends on a statically known number of iterations of the loop body, it would be possible to simulate the exact execution of the program. Using Boogie we would need to replicate the complete Boogie program or at least the loop as often as we have iterations of the loop. This would lead to too complex programs for the prover to finish in acceptable time and memory consumption boundaries. In case the number of loop iterations is not statically know for example because it depends on a parameter of the method in which the loop under consideration is located, a simulation of the exact behavior of the program is not possible. What we need in this case is the possibility to define custom places the simulation treats similar to a boundary call.

Local Places

Section 4.4.1 introduced the notion of a place, a point in the execution of a library implementation. At each point in the execution in which a boundary call may happen, at interaction points, we have predefined places. When the execution of both library implementations reaches a predefined place, the coupling invariant is checked. To handle recursive computations, we introduce the notion of a *local place*, which is a custom place to check a special invariant in the execution of

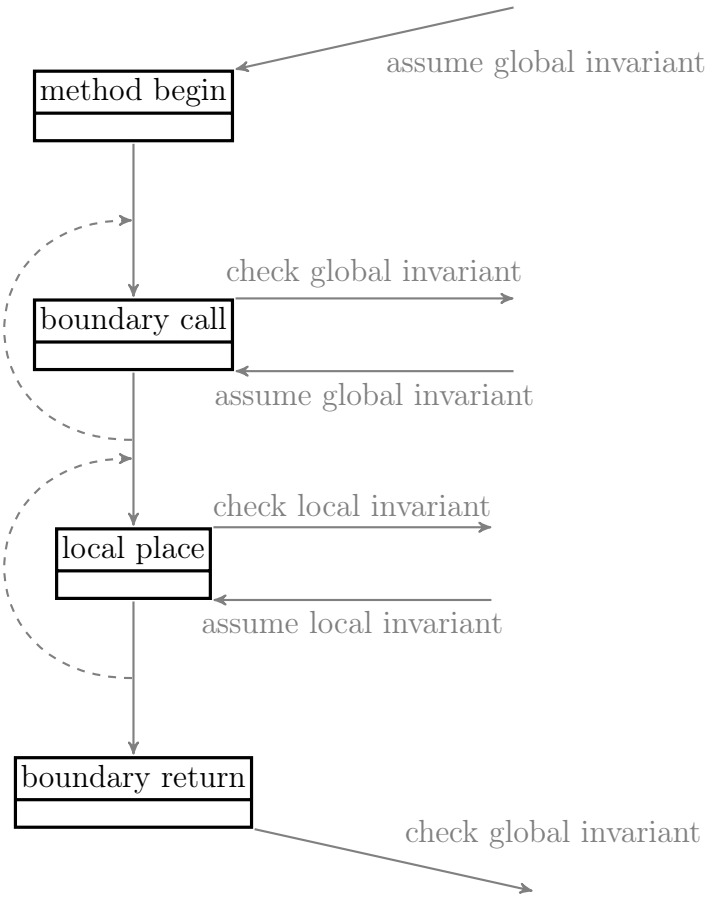


Figure 4.12: Global and local coupling invariant

the implementations. Since at local places no interaction between the context and the library happens, it is not correct to check the global coupling invariant. Instead, a local coupling invariant that is specified separately makes it possible to relate the internal state of the library implementations. Figure 4.12 illustrates when the global invariant is checked and assumed, and when the local invariant is checked and assumed. The difference between local places and predefined places is that between the check of the local invariant and the assumption that the local invariant holds, neither of the implementation executes a step. The assumption that after entering the execution of both library at a local place the local invariant holds cannot be destroyed by any execution steps. The prover can only loose information, because the steps executed before reaching the local places is not reproduced. The only information about the program state is the information given in the local invariant, which is checked before exiting the proving pass at the local places.

When looking at the boundary call example above, we can replace the boundary call with local places, one inside the loop body of each implementation. We have a comparable situation then. We relate one local place in the first implementation with exactly one local place in the second implementation. The simulation of

```
1  public class C {
2      public int sum(int n) {
3          int x = 0;
4          for(int i = 0; i<n; i++){
5              oldInLoop:
6              x += 1;
7          }
8          oldBeforeReturn:
9          return x;
10     }
11 }
```

```
1  public class C {
2      public int sum(int n) {
3          int x;
4          if(n>=0){
5              x = n;
6          } else {
7              x = 0;
8          }
9          newBeforeReturn:
10         return x;
11     }
12 }
```

Figure 4.13: Example of asynchronous use of local places

the first implementation stops at the local place, the second implementation is simulated up to the local place and the local invariant is checked. In another pass a generic execution of the loop body is simulated as well as the execution from the local place to the end of the method. The difference in this simple example compared to using boundary calls is that the global invariant does not have to hold at local places, the local invariant is checked instead.

Asynchronous Execution

In contrast to examples including boundary calls, implementations without interactions except for the boundary return at the end of the method can differ much more. For boundary calls we know that these calls have to be equal, so for example the number of calls has to be the same in the executions of both implementations. Using local places, a place in one implementation may correspond to more than one place in the other implementation. This is the case for example when relating an execution which uses a closed formula to compute a value with an iterative computation of the same value.

Figure 4.13 shows an example where a loop is compared to a computation of a closed formula. The local places oldInLoop, oldBeforeReturn and newBeforeReturn are

```
1 public class C {
2     int x;
3     public int m() {
4         x = 5;
5         return x;
6     }
7 }
```

```
1 public class C {
2     int x;
3
4     public int m() {
5         while(true) {
6             x = 5;
7         }
8     }
9 }
```

Figure 4.14: Example of a possibly infinite stall

marked by Java labels. In place oldBeforeReturn in implementation one and place newBeforeReturn in implementation two the value of local variable x is equal. To prove this fact we have to show that the loop in the first implementation computes the same value as the formula in the second implementation does. When reaching the place newBeforeReturn in the second implementation, the simulation has to stall this execution and solely simulate the first implementation until the simulation of implementation one reaches the place oldBeforeReturn. At the same time the loop invariant needs to be strong enough to show the equivalence of the values of the variables x after the loop iteration ends.

Termination

The backward compatibility of two library implementations has to be checked for each terminating execution. Possibly non-terminating executions have to be identified and handled in a special way to be sound. The reason for non-terminating computations is always recursion, either in form of a loop or recursive method calls. Each backward-jump to an already executed part of the program and each method call and return increases a loop-unroll-counter in the Boogie model. The user has to give a maximum loop-unroll-count, a upper bound for the loop-unroll-counter that may not be exceeded during execution of the program. Each implementation has its separate loop-unroll-counter. Each non-terminating computation that is not interrupted by a boundary call or local place exceeds the loop-unroll-count and leads to a failed verification attempt.

Recursive computations can be verified by disabling the optimization of boundary calls or by adding local places inside the body of the loop or recursive method. The

verification process is interrupted at those places, the loop-unroll-counter is reset. Boundary calls in recursive computations have to be equal. If one implementation loops forever then the other implementation also has to loop forever, which is checked by the usual boundary call check. The same holds for synchronous execution including local places. For two corresponding local places p1 and p2 used to synchronize the executions of the library implementations, whenever the first implementation is at place p1 then the second implementation has to reach place p2. If one of the implementations loops forever the second implementation also has to loop forever. This check is made by the usual local-place check. For terminating executions the backward compatibility is checked as defined in the theoretical model.

An additional proof goal for asynchronous execution is that the simulation is not inconsistent with a real execution of the program. The problem comes up because we may stall the execution of one implementation for an infinite timespan when the other execution is in an endless loop. While if the first implementation hangs in an endless loop this means that the behavior of the first implementation is not defined and the behavior of the second implementation is not relevant anymore. If the first implementation terminates after specific interactions, the termination of the second implementation has to be proven in addition to the equivalence check for the behavior to be comparable. The behavior of the second implementation is checked because at some point the execution is continued since the first implementation terminates.

Figure 4.14 shows an example of an infinite loop. The execution of the first implementation must be stalled while the second implementation is in the loop. In this case the first implementation would end while the other implementation would loop forever. But since we stall the execution in the simulation, Boogie would not recognize the problem on its own. We have two additional proof obligations when stalling the first execution. First we need to show that one of the executions is not stalled, since otherwise we would not make any progress at all. The second proof obligation is to show that the second execution that is not stalled ends at some point, so we do not stall the execution of the first implementation forever. In a specification based technique, a loop variant would be used to show progress and termination of a loop. In the simulation scenario with stalling, a measure for the termination of the second implementation while the first is stalled is needed. A *measure* in this case is a function from program state to an integer that has to decrease in each iteration and is always greater than zero. With these requirements, we know that the measure reaches a minimum at some point of the simulation, which also marks the end of this particular iteration. From this fact we can deduce that the stalled execution will continue at some point.

The proof obligation for two implementations I_1 and I_2 is that for two local places P_1 and P_2 if the execution of I_1 is stalled in place P_1 then I_2 makes progress during execution. I_2 making progress is translated into a decrease of the measure. The execution of I_1 can only be stalled in a local place. We can conclude that P_2 is also a local place. Since the execution of I_1 is stalled in place P_1, I_1 is in place P_1 at the

next check. Checks are only performed when reaching interactions or local places. One can conclude that I_2 is also in a local place P_3 at the next check. A local place definition can include a measure. If either P_2 or P_3 does not include a measure, the check that the measure decreases will fail because either of the measures is not defined. If both P_2 and P_3 include a measure expression then a decreasing measure shows progress of I_2. The other way round, the measure has to be chosen in a way that progress of I_2 leads to a decreasing measure. Implementation I_2 makes progress iff the measure of the new program state is less than the measure of the old program state.

When stalling the new implementation, the additional proof obligations for termination are not needed. If the first implementation does continue at some point, the behavior of the second implementation is check as usual. If the first implementation does not terminate, the behavior up to the point of the infinite loop in the first implementation and the infinite stall in the new implementation is check normally. The rest of the behavior of the second implementation is not checked. The check is sound nevertheless since the reference behavior of the first implementation after the infinite loop is not defined and as such the behavior of the second implementation is not relevant.

This approach does not show termination of an implementation. One place in the first implementation is related with a non-empty set of places in the second implementation. If the backward compatibility check succeeds this proves that the second implementation terminates if the first implementation terminates. The other direction is not correct since the non-termination of the second behavior is not guaranteed if the first implementation does not terminate.

Conditions in Local Places

We have seen a general model of asynchronous execution of both library implementations. We expect the implementations of loops to often deviate only in the boundaries of the loop counter. One example of this pattern is to optimize a loop by removing the loop iteration for the counter value 0, if this loop iteration does not contribute to the computed value. BCVerifier supports the definition of conditional places to handle this scenario without coping with measures and stalling of the execution.

Figure 4.15 shows an outline of such a situation. We define a place P1 to be at program point L with condition x>0. The code of a loop, which is outlined on the left, includes the definition of the program point L to be in the body of the loop. During execution, which is outlined on the right, the loop body is traversed multiple times. The first time the execution reaches L, the condition of P1 does not hold. All the other times the program point L is reached, the condition is true. Whenever the program point L is reached and additionally the condition x>0 holds we say, that the execution reaches place P1. We see, this way it is possible to rule out some of the traversals of a program point during execution.

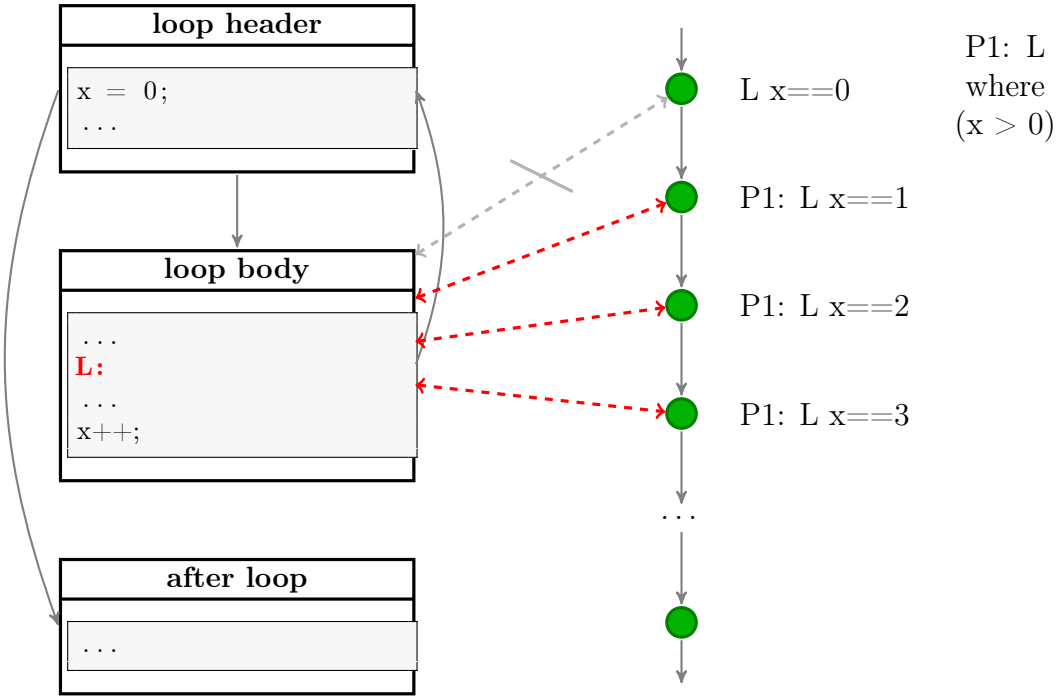


Figure 4.15: Program points and places

Up to now we talked about local and global invariants. We used them to relate places, predefined or local, to abstract the behavior of a loop or relate state in two similar implementations. In the next section we will see how to formulate such an invariant, which constructs we can use when defining invariants, and some common patterns that will be part of almost every invariant relating the program state of the library implementations.

4.7 Formulating an Invariant

To show backward compatibility of two implementations of a library, we use a coupling invariant to relate the internal state of both implementations. What we describe in the following section are common patterns that appear in invariants and what needs to be taken care of for the invariant to be safe.

There are three kind of definitions one usually uses to define a coupling between the inner state of two library implementations, two for the global invariant and one for the local invariant specification.

Relation Between Fields

The global state of both library implementations, that can be accessed by all methods of the respective implementation, is implemented in fields of the class. This means there is a relation between the fields of both implementations because

the state of the old implementation has to be represented in the new implementation as well for the behavior to be backward compatible. This relation should be expressed in the coupling invariant.

Since we want to talk about objects that are of a specific type, we have to specify the type of the object in the invariant. We can use the definition RefOfType(o, heap, t) to do that.

```
1 axiom (forall o: Ref, heap: Heap, t: TName :: RefOfType(o, heap, t)
2   <==>
3 Obj(heap, o) ==> subtype(libImpl(heap), heap[o, dynType], t));
```

o is the object reference under consideration, heap is the heap of the respective library implementation, either heap1 or heap2, and t is the type we want to enforce on the reference. o is of type t if it is an object. Additionally we want to define that the object under consideration is really an object, since otherwise it is not possible to talk about the fields of the object. For this to be possible the objects needs to be allocated and completely initialized. Objects are initialized if they are allocated and unequal to null. Additionally, the construction of the object has to be finished. As we have already discussed in Section 4.4.1, objects are completely initialized if they are created by a library implementation or exposed during the call to the constructor of a superclass belonging to the library. We have added a definition to the prelude enforcing these properties.

```
1 axiom (forall heap: Heap, r: Ref :: Obj(heap, r)
2   <==>
3     r != null &&
4     heap[r, alloc] &&
5     (heap[r, exposed] || !heap[r, createdByCtxt]));
```

The expression Obj(heap, r) evaluates to true if r is an object that is completely initialized in the heap, either heap1 or heap2.

The last property we have to consider is that both objects should be related since otherwise the program state would not be comparable in any way. We already know how to enforce that two objects are related, we use related[r1, r2] to specify the object references r1 and r2 point to objects that are related.

Putting all pieces together we have the following pattern:

```
1 (forall o1, o2: Ref :: Obj(heap1, o1) && RefOfType(o1, heap1, $C) &&
    Obj(heap2, o2) && RefOfType(o2, heap2, $C) && related[o1, o2] ==>
    P)
```

This pattern defines an invariant P that has to hold for two related objects o1 and o2 that appear in the equal interactions at the same position. The pattern can be used in invariant definitions by replacing the class $C and the condition P with custom expressions.

Relating Local State at Boundary Calls

The second pattern we present here is a pattern relating the state of two program executions before a boundary call from the library to the context and after the corresponding boundary return into the library. As described in Section 4.4.3 it is important to check multiple interaction frames for adherence to the coupling invariant. We want to specify conditions that hold in every stack frame that is at a specific place, a boundary call in this case. We first have to clarify, how such a situation can look like.

Since boundary calls create new interaction frames, we only have to look at all top most stack frames of all interaction frames instead of every stack frame. What we have to quantify over are all interaction frames of the library, for the stack pointer we can use the top most stack pointer of these interaction frames identified by spmap[i] for the i-th interaction frame. An interaction frame belongs to the library if the index of this frame is odd, `i % 2 == 1` for interaction frame index `i`, since the first interaction frame belongs to the context containing the main method and interaction frames are alternating. The place of the boundary call has to be selected using the stack variable place. Putting all pieces together we have the following pattern:

```
1 (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
    && (stack1[iframe][spmap1[iframe]][place] == q) ==> P)
```

This pattern defines an invariant P, which holds at all place q of the boundary call from the library to the context. The variable ip1 holds the index of the current interaction frame. In P we can use stack1[iframe][spmap1[iframe]][<varname>] to refer to local variables of the first implementation and stack2[iframe][spmap2[iframe]][<varname>] to refer to local variables of the second implementation. Accessing the heaps of both implementations is possible as well using the usual syntax heap1[<ref>, <fieldname>] and heap2[<ref>, <fieldname>], respectively.

Local Invariant

The patterns we presented above are patterns for global invariants, so invariants that have to hold whenever the control flow is outside the library implementation. These global invariants are not checked at local places, since these invariants do not have to hold when the control flow is still inside the library implementation. The invariants for local places, also called local invariant, have to be specified separately.

Local invariants are only checked and assumed for the current stack frame of the current interaction frame. In contrast to the global invariant, the execution is not continued between checking the local invariant and assuming the local invariant. When the execution is at a local place, there will never be another stack frame above this current frame. The construction that checks a condition for all interac-

tion frames of the library is not needed in this case. We specify the place as seen above using the variable place of the current stack frame.

```
1 (stack1[ip1][spmap1[ip1]][place] == q) ==> P
```

This pattern defines a local invariant P, which holds at all local places q. In P we can access the stack and heap as usual.

Safe Expressions

For the conditions P in the examples above we have to make sure that these properties are defined for all possible values. Partial expressions, expressions that are not defined for some variable allocations, leave Boogie with freedom to choose arbitrarily the meaning of these expressions. A specification which includes partial expressions most likely does not have the intended meaning. A prominent example is an object on which a field is accessed which is not checked to be unequal to null. An access to a field on null is not defined. The same holds for division or modulo operation, where we have to take care of division by zero. These safeguards have to be added to the invariant by hand in the current implementation. Failing to do so might result in unexpected results of the proving process. Most of the defects in the specification are recognized by the check that the coupling invariant has to hold in the initial state, so with an empty stack and heap. Some, such as the properties of objects that might as well be null, may not be caught be this check.

```
1 class A {
2         B f1;
3         B f2;
4 }
5 class B {
6         int i;
7 }
```

As an example where the error in the specification is not easily discovered, a specification might try to relate the fields f1 and f2 of class A. These fields are of type B which has a field i of type integer. A specification only stating that the field i of f1 and f2 are unequal f1.i != f2.i without checking f1 and f2 to be unequal to null will also state the null.i != null.i, which is clearly an error. Since in the initial state there are no objects that could be corresponding, the specification will not be found to be invalid by the initial configuration check. In all other cases, the coupling invariant is assumed to be true, initially. The inconsistency will most likely not be revealed in the proving process. For the user of the verifier it is not clearly visible that the specification is erroneous.

The properties that ensure the safety of the expressions used in the coupling invariant can be generated from the information obtainable from the implementations of the library. A specialized specification language can be developed that extracts

information about fields and methods of classes and enriches the expressions given by the user with the needed checks. This specification language is not part of this theses and is deferred to future work.

4.8 Divergence in Simulated Behavior

The simulation of the byte code we use is rather close to how the code would be executed by the Java runtime. But in few places we had to diverge from the actual behavior. We will describe this divergence here.

For mathematical operations that lead to an overflow, such as adding a positive integer to Integer.MAX_VALUE, Java has the semantic that the overflow is a valid operation and the value will be "wrapped around". For our Boogie implementation of the simulation of bytecode, this is not easily to reproduce. The possibilities for us were to simply fail if this kind of overflow was possible inside a program or to ignore these cases and to choose the possible values of integer variables accordingly. Since even in simple examples the first alternative would introduce a not neglectable overhead in the invariant specifications we chose to ignore overflows in mathematical operations altogether. We assume the value to be in the correct range of the corresponding number type after performing a mathematical operation.

A different problem exists for access to fields or methods of an object. Normally, when accessing a field or method on null, Java throws a NullPointerException and terminates the program if the exception is not handled inside to program. Since we have no model of exceptions, we had to either add a proving obligation to the Boogie program to show that all references that access a field or method are unequal to null or to assume that those references are unequal to null. Neither of these alternatives is correct in every situation. One can always imagine a situation that favors one of the alternatives over the other. Thus we have introduced a runtime parameter to the generator to switch between both alternative generations.

Chapter 5

Implementation of the Tool

This section is dedicated to our implementation of a Backward Compatibility Verifier, short BCVerifier, which is based on the Boogie model introduced in Chapter 4.

One decision we had to make was, which source language to use for the translation of the verification conditions. One possibility was to directly use Java [5] as the source language. That way the Boogie representation of the library code would have been very close to the original Java source code. Java as well as Boogie both support complex nested expression as well as loops and if-branches. The second possibility was to use *Java Bytecode*, the stack machine code the Java Virtual Machine uses to execute Java programs at runtime. Since Java byte code is a rather low level language it abstracts from some complex features of Java such as different loop constructs, if-constructs with- and without else-branch and generic types. Hence we would have to support only a rather small set of possible operations in our translation.

There already was a translation framework that targets translation from Java byte code to the Boogie Programming Language, called *B2BPL*, with a different focus then we had. We already introduced B2BPL in Section 3.2. We decided to reuse some of the translation routines and analysis techniques used to generate Boogie code, that is rather close to the byte code representation of the libraries.

In this section, we will give an overview of the lifecycle of BCVerifier, what we took from B2BPL and what we adapted.

5.1 Overview

Figure 5.1 shows the translation process of BCVerifier. Our tool works on Java byte code. The library is first translated into bytecode using the standard Java compiler, the results are class-files of both library implementations. Implementation one is assumed to be the old version, implementation two is assumed to be the new one which should be behavioral compatible to the old version. The

class-files of those two library implementations are passed to BCVerifier together with the coupling invariant, which at the moment has to be formulated as Boogie expressions. The translation process described below transforms the libraries together with the coupling invariant to a Boogie program which, if verified using a combination of Boogie and an SMT solver such as Z3, results in either a success or a failure. Success means in that regard that the second library is backward compatible with respect to the observable behavior. A failure could be the result of a non-compatible second library, or a coupling invariant which is not strong enough to prove the correct relation.

5.2 Lifecycle

In this section we describe the lifecycle of BCVerifier, which consists of four phases:

Compilation The backward compatibility checker takes Java source files as input. For the underlying instruction translator to work, we need to have access to the byte code files which result from translating the source code using a Java compiler. For all our translations to Boogie to work, we need additional information such as line number encoding into the byte code and the name of local variables. By default, we leave the compilation to the user. Nevertheless, we support the flag −c or −−compile as a runtime flag to the tool, which enables a preconfigured compilation to be performed before the further steps of the translation are started.

Source Compatibility Check As we know from the theoretical model described in Chapter 2, source compatibility is a prerequisite of the backward compatibility check. By default, the source compatibility check is also left to the user. We offer an optional checking phase, which can be enabled by the runtime flag −k or −−sourcecompatibility. If enabled, the phase checks the conditions that have to hold for the library implementations to be source compatible, such as every public type of the first implementation is also a public type in the second implementation and so forth.

Translation The translation phase is the most important phase of the backward compatibility check. It is therefore not possible to disable this phase. The byte code of all types is parsed and translated into the Boogie model described in Chapter 4. The Java methods are translated with the help of the MethodTranslator. The result is a list of Boogie procedures with the stack and heap implementations adapted to the correct library implementation. These Boogie procedures are taken apart, the bodies are combined with the precondition and check blocks into a new procedure and the needed local variables are taken care of. Last this new procedure

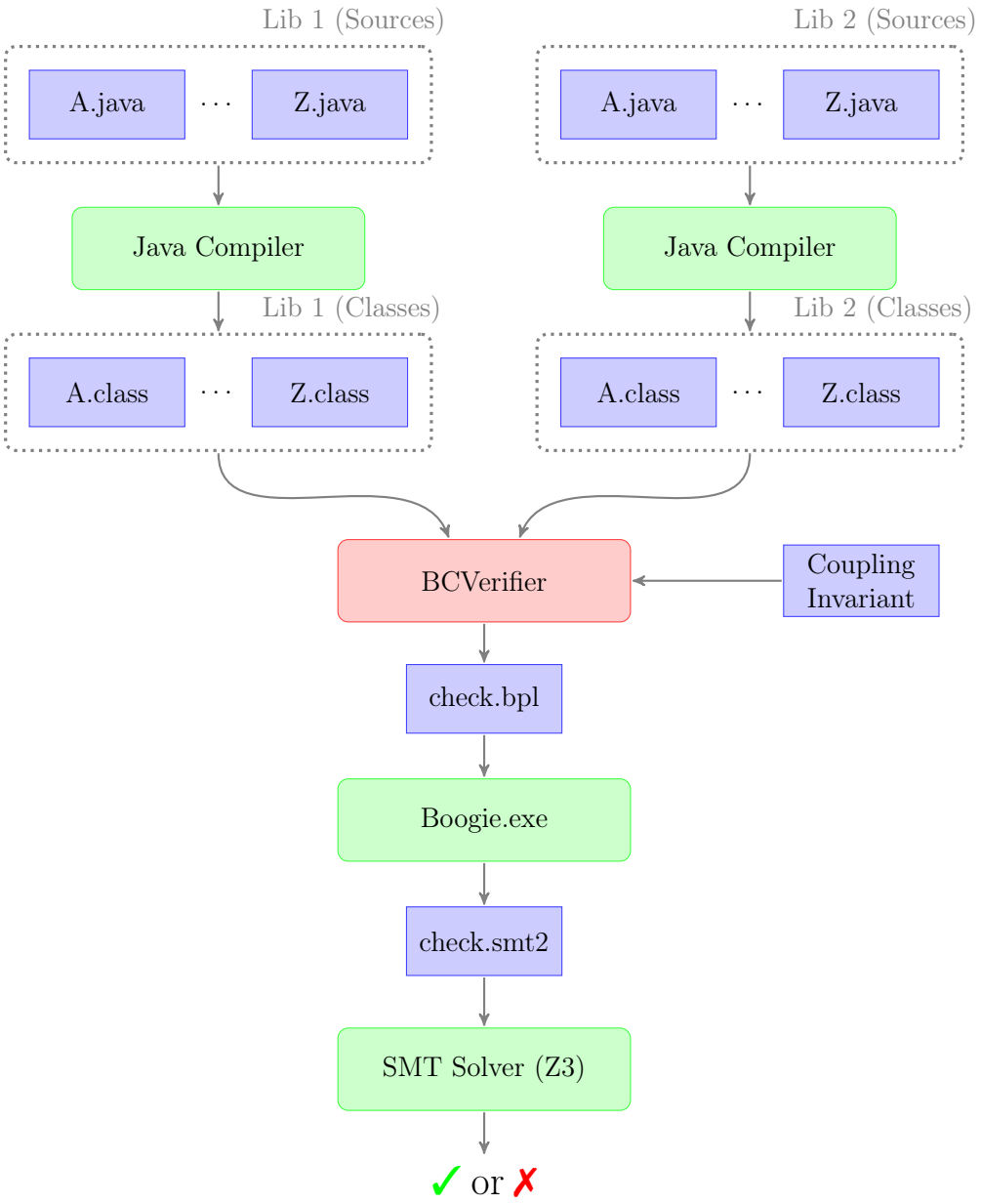


Figure 5.1: Overview of the translation process of BCVerifier

called checkLibraries is combined with the prelude generated for our Boogie model to a complete Boogie model, which is written to a file.

Behavior Check The behavior check phase checks backward compatibility of both library implementations. It runs Boogie over the file created by the translation phase. We support three possible actions this phase can perform. The action NONE skips the whole behavior check phase. The boogie file is still retained and can be run through Boogie manually. The action TYPECHECK performs a type check on the Boogie file. The specification is not checked using an SMT solver, the result only tells if the Boogie program is type correct. The last possible action is VERIFY, which is also the default action. This actions takes the Boogie model, runs the Boogie checker over the file and performs an SMT check on the resulting verification conditions. The result of this action tells, if both library implementations are backward compatible or if backward compatibility could not be verified.

5.3 Architecture

The architecture of the tool is split into the following parts:

- Library abstraction, life cycle, and generation of connecting parts
- LibraryCompiler, which compiles the Java source code of the library implementations
- MethodTranslator, which translates Java methods to Boogie code
- InstructionTranslator, which translates single byte code instructions to Boogie instructions
- BoogieRunner, which runs Boogie with the needed parameters on the generated file

Some of these parts are taken from B2BPL and modified to support our Boogie model. Some parts are written from scratch to use the underlying translations and assemble the complete Boogie file.

The Library abstraction represents the library and its two implementations, the original and the new one. The lifecycle of the translation is encoded into this abstraction and can be run using method runLifecycle(). The lifecycle generates the different parts of the Boogie representation, the prelude and the Boogie code representing the Java methods, and connects them by suitable Boogie code. This connecting code consists of the blocks encoding the preconditions of the proving process and the dispatch blocks used to resolve dynamic method dispatch as seen in Section 4.4.2. Additionally the blocks to check the invariant is preserved are generated and connected to the method implementations.

The LibraryCompiler uses a standard Java compiler implementation, currently the Eclipse JDT BatchCompiler, to compile the Java source files into byte code class files. The compiler is configured to include debug information we need to find out about line numbers in the byte code and to add debug information into the generated Boogie code about names of local variables. The LibraryCompiler is also used to parse an AST of the Java source files for source compatibility checking.

The parts of BCVerifier that have been presented up to now are all written from scratch. The MethodTranslator is mostly taken from B2BPL. The internal translations needed to be adapted to support our custom model of program state. The byte code classes are parsed into ASTs of the classes of the library. These ASTs are passed to the MethodTranslator which translates all methods of all classes and passed them back to the Library abstraction for further processing.

The InstructionTranslator is used by the MethodTranslator to translate single byte code instructions to Boogie instructions that simulate the behavior. This translator addresses the translation from a stack machine model as used by the Java runtime to our assignment model used as representation of the program state. The InstructionTranslator is also taken from B2BPL and modified to support our abstraction of stack and heap.

The BoogieRunner is used after the translation process is finished to check the specification. Boogie is configured using parameters to perform loop unrolling, which we need because of our simulation approach. The result, which Boogie outputs to the console, is parsed and analyzed to give feedback about the number of errors that occurred during the proving process, which is part of the success criterium.

5.4 Reuse of B2BPL

B2BPL uses a classical method of behavior checking. It uses a specification of the program and checks the behavior shown by the byte code is compatible with this behavior. We are reusing parts of B2BPL to translate the Java byte code into Boogie procedures. Since our method is to compare the behavior of two library implementations, we had to add a TranslationController, which replaces the statements involved in modifying the program state. Access to the heap in the translation of the first library implementation has to use the variable heap1, whereas in the second implementation the variable heap2 is to be used. A similar distinction has to be made for the stack. This library implementation specific information can be acquired from the TranslationController, which keeps track of the implementation currently processed.

Concerning the MethodTranslator, we mostly reused the translation process from Java procedures to Boogie. We removed or deactivated the translation of BML, the byte code specification language, since it is not usable for our approach. The initialization blocks of procedures now include assumptions of the types of the

parameters and assignments from the parameters to the corresponding local variables. This is implemented in the method translateInit (). After the body of the method, we added blocks to distinguish boundary returns from internal returns in the method translatePost (). To form the basic blocks of the Boogie procedure, the method translator uses a control flow graph of the byte code instructions. For each block that marks a back-link, indicating a loop in the program execution, a check of the loop unrolling is added. This is implemented in the method translateCFGBlockStart().

The InstructionTranslator is operationally equal to the implementation used in B2BPL. The stack and heap accesses had to be adapted to support our special model of program state. This also resulted in a different implementation of object allocation, which also had to be adapted. Additionally, we had to change the translation of method invocations, since B2BPL uses Boogie procedures to represent Java methods whereas we use blocks in the procedure checkLibraries. A new method translateLocalPlaces(List<Place> localPlaces) takes care of the translation of local places for a given program point. Since we mostly ignore exceptional behavior at the moment, we also had to replace the exception handling code with respective assumes and asserts to cancel the proving process at occurrence of an exception or add prove obligations to avoid exceptional program states.

The Translator, which is the main entry point of the translation process of B2BPL, was modified with respect to what is generated. Originally, the complete Boogie program including the prelude, all procedures and all needed helper functions was generated and returned using the method translate (JClassType... types). We added a method translateMethods(JClassType... types) to separate the methods, which we use to build our own checkLibraries procedure, from the rest of the program, which we mostly can not reuse. The prelude of B2BPL was completely replaced with a modified version of the prelude used by Spec# [6], since they use a similar heap model.

The representation of a Boogie abstract syntax tree together with all tools needed such as pretty printers was taken from B2BPL. A modification was only needed to support including Boogie commands, which are not parsed, as a raw string into the AST. This support is used to avoid parsing the invariant declarations, which are instead directly passed into the Boogie program.

Chapter 6

Case Study

This chapter shows some examples that can be verified with the current implementation of BCVerifier. These examples include structurally different implementations of the same functionality. For most examples knowledge about the dynamic dispatch of methods for class-based programming languages is needed to reason about the backward compatibility of the library implementations.

6.1 Internal Representation

One detail in which two library implementation can differ is the internal representation of the object state. The OBool example covers difference in the internal representation and an abstract data type which is used internally and can also be used by the context.

The OBool example consists of two implementations of an object that serves as a container for a boolean value.

Figure 6.1 shows the old implementation of the library. The class Bool is a simple container for a primitive boolean value. The methods set() and get() allow access to the internal state of the abstract data type. The class OBool is a wrapper around a Bool object. Each instantiation of OBool creates a new instance of the class Bool representing the internal state of the OBool object. The constructor initializes the value of the Bool object to **false**. The methods setg() and getg() give access to internal state using primitive boolean values. The Bool object itself is not exposed to the outside.

Figure 6.2 shows the new implementation of the library. The class Bool is identical to the old implementation. The internal state of the OBool object is negated with respect to the old implementation. The internal state is initialized to **true**, whereas the methods setg() and getg() negate the value before processing the internal state. These facts are also expressed in the coupling invariant.

```
1  public class Bool {
2    private boolean f;
3    public void set(boolean b) {
4            f = b;
5    }
6    public boolean get() {
7            return f;
8    }
9  }
10
11 public class OBool {
12   private Bool g = new Bool();
13   public OBool() {
14         g.set(false);
15   }
16   public void setg(boolean b) {
17         g.set(b);
18   }
19   public boolean getg() {
20         return g.get();
21   }
22 }
```

Figure 6.1: Old implementation of OBool

```
1  public class Bool {
2    private boolean f;
3    public void set(boolean b) {
4            f = b;
5    }
6    public boolean get() {
7            return f;
8    }
9  }
10
11 public class OBool {
12   private Bool g = new Bool();
13   public OBool() {
14         g.set(true);
15   }
16   public void setg(boolean b) {
17         g.set(!b);
18   }
19   public boolean getg() {
20         return !g.get();
21   }
22 }
```

Figure 6.2: New implementation of OBool

```
1  ( forall o1, o2: Ref ::
2    Obj(heap1, o1) && Obj(heap2, o2) &&
3    RefOfType(o1, heap1, $obool.OBool) && RefOfType(o2, heap2,
         $obool.OBool) &&
4    related[o1,o2]
5  ==>
6   (heap1[heap1[o1,$obool.OBool.g],$obool.Bool.f] !=
       heap2[heap2[o2,$obool.OBool.g],$obool.Bool.f]) )
7
8  ( forall o1, o2: Ref ::
9    Obj(heap1, o1) && Obj(heap2, o2) &&
10   RefOfType(o1, heap1, $obool.Bool) && RefOfType(o2, heap2,
         $obool.Bool) &&
11   related[o1, o2]
12 ==>
13   heap1[o1, $obool.Bool.f] == heap2[o2, $obool.Bool.f] )
```

For objects (line 2) of type OBool (line 3) that are related (line 4) the fields f of the object referenced by field OBool.g are unequal. Since the value of this field f is a boolean, this means the value in the new implementation is the negation of the value in the old implementation. For objects (line 9) of type Bool (line 10) that are related (line 11) the value of field Bool.f is equal, since these objects have been used in the same interactions between the context and the library. The objects of type Bool that are used in the implementation of class OBool are not exposed and therefore not in correspondence relation. This is expressed by the following addition to the coupling invariant.

```
1 Internal($obool.OBool,$obool.OBool.g,heap1) &&
      Internal($obool.OBool,$obool.OBool.g,heap2)
2 NonNull($obool.OBool,$obool.OBool.g,heap1) &&
      NonNull($obool.OBool,$obool.OBool.g,heap2)
3 Unique($obool.OBool,$obool.OBool.g,heap1) &&
      Unique($obool.OBool,$obool.OBool.g,heap2)
```

The objects assigned to fields that are defined to be internal are not created by the context and not exposed. These objects are internal to the library.

```
1 axiom (forall c: TName, f: Field Ref, heap: Heap ::
2   Internal(c, f, heap)
3   <==>
4   (forall r: Ref :: Obj(heap, r) && RefOfType(r, heap, c) ==>
5     !heap[heap[r, f], createdByCtxt] && !heap[heap[r, f], exposed]));
```

In the invariant of the OBool example, the field OBool.g is declared to be internal. Additionally, this field is also declared to be non-null, the field is initialized during initialization of the instance of type OBool and is never assigned the value null.

```
1 axiom (forall c: TName, f: Field Ref, heap: Heap ::
2   NonNull(c, f, heap)
3   <==>
4   (forall r: Ref :: Obj(heap, r) && RefOfType(r, heap, c) ==>
      heap[r, f] != null));
```

The most important definition of the function NonNull() is heap[r, f] != null, fields that are defined to be NonNull never have the value null when the control flow is in the context.

Aliasing between the fields OBool.g could influence the invariant. Aliasing is ruled out be the definition of Unique(TName, Field Ref, Heap).

```
1 axiom (forall c: TName, f: Field Ref, heap: Heap ::
2   Unique(c, f, heap)
3   <==>
4   (forall r1: Ref, r2: Ref ::
5     Obj(heap, r1) && Obj(heap, r2) &&
6     RefOfType(r1, heap, c) && RefOfType(r2, heap, c) &&
7     r1 != r2 ==>
8       heap[r1, f] != heap[r2, f]));
```

Fields are unique in the sense that if two objects that declare the field are unequal then the objects assigned to these fields are unequal as well. Aliasing between two fields that are declared unique is impossible.

The complete coupling invariant of the OBool example is shown in Fig. 6.3. The parameters used are --iframes 1 to reduce the number of interaction frames used in the checking process to one and --loopUnroll 5. The difficulty of this example is to show that the calls of methods g.get() and g.set() in lines 17 and 20 in the old and the new implementation are internal calls instead of boundary calls and as such no callback to the context can happen at these points. The fact that the result returned by the method OBool.getg() is equal for both library implementations and that the internal state is equal after equivalent calls to the method OBool.setg() can be proven using the given invariant.

In the installation folder (install/BCVerifier), the example can be started using the command

```
1 ./bin/bcv --compile --specification examples/obool/bpl/spec4.bsl
    --libs examples/obool/old examples/obool/new --loopUnroll 5
    --iframes 1
```

This execution of BCVerifier generates the Boogie model of the library OBool in the file examples/obool/bpl/output.bpl and checks the model. The result of the execution should be a success of the verification process. The console output should be Boogie program verifier finished with 1 verified, 0 errors.

```
1  >>>invariant
2  ( forall o1, o2: Ref ::
3    Obj(heap1, o1) && Obj(heap2, o2) &&
4    RefOfType(o1, heap1, $obool.OBool) && RefOfType(o2, heap2,
         $obool.OBool) &&
5    related[o1,o2]
6  ==>
7   (heap1[heap1[o1,$obool.OBool.g],$obool.Bool.f] !=
       heap2[heap2[o2,$obool.OBool.g],$obool.Bool.f]) )
8
9  ( forall o1, o2: Ref ::
10   Obj(heap1, o1) && Obj(heap2, o2) &&
11   RefOfType(o1, heap1, $obool.Bool) && RefOfType(o2, heap2,
         $obool.Bool) &&
12   related[o1, o2]
13 ==>
14   heap1[o1, $obool.Bool.f] == heap2[o2, $obool.Bool.f] )
15
16 Internal($obool.OBool,$obool.OBool.g,heap1) &&
       Internal($obool.OBool,$obool.OBool.g,heap2)
17 NonNull($obool.OBool,$obool.OBool.g,heap1) &&
       NonNull($obool.OBool,$obool.OBool.g,heap2)
18 Unique($obool.OBool,$obool.OBool.g,heap1) &&
       Unique($obool.OBool,$obool.OBool.g,heap2)
19 <<<
```

Figure 6.3: Coupling invariant of the OBool example

```
1  public interface MyList {
2          public void set(int i, Object o);
3
4          public Object get(int i);
5  }
6
7  public class C{
8          private MyList list;
9
10         public void setList(MyList list){
11                 this.list = list;
12         }
13
14         public void m(){
15                 for(int i=0; i<5; i++){
16                         list.set(i, new Object());
17                 }
18         }
19 }
```

Figure 6.4: Old implementation of simpleLooop

6.2 Callback

The OBool example includes only method calls that are known to be internal calls. The method invocations target fields whose objects are created by the constructor and are never reassigned. Boundary calls made by the library that target methods in the context are another possible behavior of a library implementation. These callbacks to the context have to be equal in both implementations.

The simpleLoop example relates a for-loop in the old implementation with a while-loop in the new implementation, which is not very hard to do considering the fact that the byte code of both implementations is almost equal. The more interesting problem is to show that both implementations lead to the same callbacks to the context.

Figure 6.4 shows the code of the old implementation of simpleLoop. The interface MyList defines a standard interface for list implementations. The example library does not include an implementation of the interface MyList. Class C includes a method setList() to pass the list to fill. The method m() fills the list with five new objects using a for-loop. This use-case does not consider that the field C.list has to be non-null for the method call to succeed and that the list needs to offer space for the five objects created.

Figure 6.5 shows the code of the new implementation of simpleLoop. The interface MyList is identical to the interface in the old implementation. The new implementation of class C also includes the same method setList(). The method m() is implemented using a while-loop instead of a for-loop. The rest of the computation

```
1  public interface MyList {
2          public void set(int i, Object o);
3
4          public Object get(int i);
5  }
6
7  public class C{
8          private MyList list;
9
10         public void setList(MyList list) {
11                 this.list = list;
12         }
13
14         public void m(){
15                 int i = 0;
16                 while(i<5){
17                         list.set(i, new Object());
18                         i++;
19                 }
20         }
21 }
```

Figure 6.5: New implementation of simpleLooop

is equal to the old implementation. To show this fact, the callbacks made by the call to list.set () need to be proven to be equal in both implementations.

To handle the loops included in both implementations of the example library, it would be sufficient to unroll the five statically known loop iterations and prove the resulting sequential program. We want to consider the general case where the number of iterations of the loops is not statically know. The calls to MyList.set() in line 16 in the old and line 17 in the new implementation are known to be boundary calls, because the example library does not include an implementation of the interface MyList. The loops are split into single iterations by deactivating the optimization we implemented for boundary calls. This is done by adding the following lines into the specification in the section "preconditions".

```
1 useHavoc[lib1_C.m_set$int$java.lang.Object_0] := false;
2 useHavoc[lib2_C.m_set$int$java.lang.Object_0] := false;
```

lib1_C.m_setintjava.lang.Object_0 is the place of the call to MyList.set() in the old implementation and lib2_C.m_setintjava.lang.Object_0 is the place of the call to said method in the new implementation. If the property useHavoc is set to **false** for two places, the optimization **havoc**ing the heap and continuing the proving process is not performed when reaching these two places. Since both loops run synchronously, when the execution of the old implementation reaches the boundary call then the second implementation also reaches the boundary call to MyList.set(). This property is expressed in the invariant using the following expression.

```
1 (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
    ==>
2   (stack1[iframe][spmap1[iframe]][place] ==
      lib1_C.m_set$int$java.lang.Object_0) ==>
3     (stack2[iframe][spmap2[iframe]][place] ==
        lib2_C.m_set$int$java.lang.Object_0))
```

We use the pattern described in Section 4.7 for the local state of a class at a boundary call. This part of the invariant expresses that whenever the top-most stack frame of the first implementation is at the place lib1_C.m_setintjava.lang.Object_0 then the top-most stack frame of the second implementation is at the place lib2_C.m_setintjava.lang.Object_0 in the same interaction frame.

The proving process is interrupted when reaching the boundary calls and is continued afterwards in a new pass. During this process, the information about the local variables of the method is lost. Part of this information is required to prove the loop iterations perform the same interactions with the context. The following expressions synchronize the execution of both library implementations.

```
1 (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
2 ==>
3   (stack1[iframe][spmap1[iframe]][place] ==
      lib1_C.m_set$int$java.lang.Object_0)
4   ==>
5     (stack1[iframe][spmap1[iframe]][reg1_i] ==
        stack2[iframe][spmap2[iframe]][reg1_i]))
6
7 (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
8 ==>
9   (stack1[iframe][spmap1[iframe]][place] ==
      lib1_C.m_set$int$java.lang.Object_0)
10  ==>
11    (spmap1[iframe] == 0 && spmap2[iframe] == 0))
```

The first expression synchronizes the loop iterations. The value of the loop counter, local variable i, is equal in both implementations. Variable i is the first local variable declared in the method and the method has no parameters. The receiver object is represented by reg0_r followed by the local variable i represented by reg1_i. The second expression synchronizes the stack frames of both implementations. Since neither of the methods m() makes any internal calls, the interaction frame consists of exactly one stack frame executing method m().

The complete coupling invariant specification is shown in Fig. 6.6. The Boogie model can be generated and verified using the following command in the installation folder.

```
1 ./bin/bcv --compile -N --specification
    examples/simpleLoop/bpl/spec.bsl --libs examples/simpleLoop/old
    examples/simpleLoop/new --loopUnroll 3 --iframes 1
```

```
1  >>>invariant
2  (forall o1,o2: Ref :: ObjOfType(o1, $C, heap1) && ObjOfType(o2, $C,
       heap2) && related[o1, o2] ==> RelNull(heap1[o1, $C.list],
       heap2[o2, $C.list], related))
3
4  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==>
       (stack2[iframe][spmap2[iframe]][place] ==
       lib2_C.m_set$int$java.lang.Object_0))
5  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==>
       (stack1[iframe][spmap1[iframe]][reg1_i] ==
       stack2[iframe][spmap2[iframe]][reg1_i]))
6  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==> (spmap1[iframe] == 0 &&
       spmap2[iframe] == 0))
7  <<<
8  >>>preconditions
9  useHavoc[lib1_C.m_set$int$java.lang.Object_0] := false;
10 useHavoc[lib2_C.m_set$int$java.lang.Object_0] := false;
11 <<<
```

Figure 6.6: Complete specification of the simpleLoop example

The command line parameter –N disables null-checks and assumes all object references to be non-null objects. The prover will not complain about null-references for method calls, which are out of scope for this example. The additional precondition for the context is that the list is set to a non-null object before calling method m().

6.3 Conditional Place

The simpleLoop example shown in Section 6.2 showed two loops that performed the same number of iterations. This can not always be assumed. When optimizing loops for example a specific number of iterations may be omitted by one implementation. The following example shows such a situation.

The oneOffLoop example calculates the sum of the first n integers. The standard way of computing the sum is to start at zero and sum up all integers up to the given n. The first iteration summing up zero has no influence on the computed result.

Figure 6.7 shows the code of the old implementation of the oneOffLoop library. The loop starting in line 4 initializes the loop counter with 0 and increases the

```
1 public class C {
2         public int m(int n){
3                 int x = 0;
4                 for(int i=0; i<n; i++){
5                         x += i;
6                 }
7                 return x;
8         }
9 }
```

Figure 6.7: Old implementation of oneOffLoop

```
1 public class C {
2         public int m(int n){
3                 int x = 0;
4                 for(int i=1; i<n; i++){
5                         x += i;
6                 }
7                 return x;
8         }
9 }
```

Figure 6.8: New implementation of oneOffLoop

counter by one in each iteration. The variable x is used to compute the result by adding the counter in each iteration of the loop up to n, thereby iteratively computing the sum of the first n integers. The new implementation of the library shown in Fig. 6.8 initializes the loop counter with 1, omitting the first iteration of the loop since adding zero to x makes no contribution to the computation of the result. The rest of the code is identical to the old implementation.

The loop in the old implementation performs one additional iteration compared to the loop in the new implementation. This suggests that synchronizing the iterations of both loops can not be easily achieved. A possible solution is to stall the execution of the new implementation before entering the loop and meanwhile execute the first iteration of the loop in the old implementation. The measure, to verify that the execution of the new implementation is continued, would be based on the loop counter. A simpler solution is to define a local place with a condition that rules out the first iteration of the loop in the old implementation.

```
1 inLoop1 = old 5 (stack1[ip1][spmap1[ip1]][reg3_i] > 0)
2 inLoop2 = new 5 (true)
```

The variable reg3_i represents the loop counter i. The place inLoop1 is defined in the old implementation before the expression in line 5 whenever the value of the loop counter is greater than zero. This omits the first iteration of the loop. The place inLoop2 is defined in the new implementation before the expression in line 5 and is reached during each iteration of the loop.

```
1  >>>local_invariant
2  stack1[ip1][spmap1[ip1]][place] == inLoop1 <==>
       stack2[ip2][spmap2[ip2]][place] == inLoop2
3  (stack1[ip1][spmap1[ip1]][place] == inLoop1) ==>
       (stack1[ip1][spmap1[ip1]][param1_i] ==
       stack2[ip2][spmap2[ip2]][param1_i]) //the value of the parameter
       is the same
4  (stack1[ip1][spmap1[ip1]][place] == inLoop1) ==>
       (stack1[ip1][spmap1[ip1]][reg2_i] ==
       stack2[ip2][spmap2[ip2]][reg2_i]) //the value of x is the same
5  (stack1[ip1][spmap1[ip1]][place] == inLoop1) ==>
       (stack1[ip1][spmap1[ip1]][reg3_i] ==
       stack2[ip2][spmap2[ip2]][reg3_i]) //the value of i is the same
6  <<<
7  >>>places
8  inLoop1 = old 5 (stack1[ip1][spmap1[ip1]][reg3_i] > 0)
9  inLoop2 = new 5 (true)
10 <<<
```

Figure 6.9: Complete specification of the oneOffLoop example

```
1  stack1[ip1][spmap1[ip1]][place] == inLoop1 <==>
       stack2[ip2][spmap2[ip2]][place] == inLoop2
2  (stack1[ip1][spmap1[ip1]][place] == inLoop1) ==>
3      (stack1[ip1][spmap1[ip1]][param1_i] ==
           stack2[ip2][spmap2[ip2]][param1_i])
4  (stack1[ip1][spmap1[ip1]][place] == inLoop1) ==>
5      (stack1[ip1][spmap1[ip1]][reg2_i] ==
           stack2[ip2][spmap2[ip2]][reg2_i])
6  (stack1[ip1][spmap1[ip1]][place] == inLoop1) ==>
7      (stack1[ip1][spmap1[ip1]][reg3_i] ==
           stack2[ip2][spmap2[ip2]][reg3_i])
```

Whenever the old implementation reaches the local place inLoop1, the execution of the new implementation reaches the local place inLoop2 (line 1). We consider the invocation of method m for equal parameters n (lines 2 and 3). The computed value is the same in both executions of the method, the local variable x has equal value (lines 4 and 5). The loop iterations that reach the places inLoop1 and inLoop2 respectively are synchronized over the loop counter i in both implementations (lines 6 and 7).

```
public class C {
        public int sum(int n) {
                int x = 0;
                for(int i = 0; i<n; i++){
                        x += 1;
                }
                return x;
        }
}
```

Figure 6.10: Old implementation of loopFormula

The complete specification of the oneOffLoop example is shown in Fig. 6.9. Because the first loop iteration in the old implementation is executed including the back branch to the loop header, the loop unroll count has to be increased. The Boogie model can be generated and verified using the following command in the installation folder.

```
./bin/bcv --compile --specification examples/oneOffLoop/bpl/spec.bsl
   --libs examples/oneOffLoop/old examples/oneOffLoop/new
   --loopUnroll 4 --iframes 1
```

The example shows that for specific asynchronous executions the conditional local places can be used to simplify the coupling invariant specification. Examples for such cases are loop optimizations where a small, statically known number of iterations of a loop is omitted in one of the implementations.

6.4 Stalled Execution

If the executions of both library implementation only differ in a statically known number of iterations of a loop it is simpler to use conditional local places to synchronize the loops. If the number of iterations that are different is not statically known or the situation is more complex, conditional places are not enough to synchronize the executions of the library implementations.

6.4.1 Verification of a Closed Formula

The loopFormula example compares the computation using a closed formula with an iterative computation of the result. The formula used in this example is very simple to avoid the complexity of giving Boogie enough facts to prove the formula. The result that is computed by the loopFormula example is the passed integer parameter.

```
1  public class C {
2          public int sum(int n) {
3                  int x;
4                  if(n>=0){
5                          x = n;
6                  } else {
7                          x = 0;
8                  }
9                  return x;
10         }
11 }
```

Figure 6.11: New implementation of loopFormula

Figure 6.10 shows the code of the old implementation of the loopFormula example. In this implementation, the result for parameter n is computed by adding up ones in n iterations of a loop. The new implementation, shown in Fig. 6.11, directly assigns the value of n. The result of both computations is the same, the value of the parameter n. The problem that has to be solved in this example is that the number of iterations of the loop in the old implementation depends on the parameter of the method and as such is not statically known. The new implementation computes the same result without iterations. The execution of the new implementation has to be stalled while the loop in the old implementation is executed.

```
1  newBeforeReturn = new 9 (true)
2  oldInLoop = old 5 (true)
3  oldBeforeReturn = old 7 (true)
4
5  stall2[oldInLoop, newBeforeReturn] := true;
```

In the old implementation two local places are defined. The place oldInLoop is defined to be before adding up the ones inside the loop. The place oldBeforeReturn is before the return statement of the method sum(). In the new implementation only a single local place is necessary, which is defined to be before the return statement of the method sum(). The old implementation is executed up to the place oldInLoop while the new implementation is executed up to the place newBeforeReturn. The execution of the new implementation is stalled while the old implementation is at place oldInLoop and the new implementation is in place newBeforeReturn (line 5). The execution of the loop in the old implementation continues until reaching the point oldBeforeReturn after the loop. When the execution of the old implementation reaches oldBeforeReturn and the execution of the new implementation is at newBeforeReturn the values of the local variable x on both sides are equal. Because the value of x is returned on both sides, the result of method m on both sides is equal.

```
1  (stack1[ip1][spmap1[ip1]][place] == oldBeforeReturn ||
2   stack1[ip1][spmap1[ip1]][place] == oldInLoop) <==>
3     (stack2[ip2][spmap2[ip2]][place] == newBeforeReturn)
4  stack1[ip1][spmap1[ip1]][place] == oldBeforeReturn ==>
5      stack1[ip1][spmap1[ip1]][reg2_i] ==
6      stack2[ip2][spmap2[ip2]][reg2_i]
7  stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
8     (stack1[ip1][spmap1[ip1]][param1_i] ==
9      stack2[ip2][spmap2[ip2]][param1_i])
10 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
11    (stack1[ip1][spmap1[ip1]][reg2_i] ==
12     stack1[ip1][spmap1[ip1]][reg3_i])
13 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
14    (stack1[ip1][spmap1[ip1]][reg3_i] <
15     stack1[ip1][spmap1[ip1]][param1_i])
16 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
17    (stack1[ip1][spmap1[ip1]][reg3_i] >= 0)
18 stack2[ip2][spmap2[ip2]][place] == newBeforeReturn ==>
19    (if(stack2[ip2][spmap2[ip2]][param1_i] >= 0)
20     then (stack2[ip2][spmap2[ip2]][reg2_i] ==
21           stack2[ip2][spmap2[ip2]][param1_i])
22     else (stack2[ip2][spmap2[ip2]][reg2_i] == 0))
```

The relation between the local places in both implementations is given in lines 1 to 3. The old implementation is in oldInLoop or in oldBeforeReturn while the new implementation is in place newBeforeReturn. The result of both local variables x is equal before returning (lines 4 to 6). We consider only executions where the parameters of both implementations are equal (lines 7 to 9). In this very simple example, the relation between the loop counter i and the result is clear, the value of both variables is equal (line 10 to 12). The loop counter runs from zero to n, which is stated in lines 13 to 17. The result of method m in the new implementation is also clear: It is the value of parameter n if n is greater or equal to zero and 0 otherwise (lines 18 to 22).

Figure 6.12 shows the complete specification of the loopFormula example. This example shows how to relate the computation of a value using a closed formula with an iterative computation of the same result. The closed formula used in the example is very simple. When trying to use more complicated formulas, Boogie will be unable to handle the task of verifying that the result computed by the loop is equal to the result of the formula. The complexity is equal to proving conformance of a loop computation with its functional specification, which in general is not possible without assistance for automatic verifiers such as Boogie. To cope with these kinds of challenges a more complex reasoning system for loops with access to the state before executing the loop iteration would be required to specify the effect of a single iteration. The Boogie model of the loopFormula example can be generated and verified using the following command in the installation folder.

```
1 ./bin/bcv --compile --specification
     examples/loopFormula/bpl/spec.bsl --libs examples/loopFormula/old
     examples/loopFormula/new --loopUnroll 3 --iframes 1
```

```
1  >>>places
2  newBeforeReturn = new 9 (true)
3  oldInLoop = old 5 (true)
4  oldBeforeReturn = old 7 (true)
5  <<<
6
7  >>>preconditions
8  stall2[oldInLoop, newBeforeReturn] := true;
9  <<<
10
11 //param1_i = n
12 //reg2_i = x
13 //reg3_i = i
14 >>>local_invariant
15 (stack1[ip1][spmap1[ip1]][place] == oldBeforeReturn ||
       stack1[ip1][spmap1[ip1]][place] == oldInLoop) <==>
       (stack2[ip2][spmap2[ip2]][place] == newBeforeReturn)
16 stack1[ip1][spmap1[ip1]][place] == oldBeforeReturn ==>
       stack1[ip1][spmap1[ip1]][reg2_i] ==
       stack2[ip2][spmap2[ip2]][reg2_i] //value of both x variables is
       equal before return
17 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
       (stack1[ip1][spmap1[ip1]][param1_i] ==
       stack2[ip2][spmap2[ip2]][param1_i])      //n on both sides is the
       same
18 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
       (stack1[ip1][spmap1[ip1]][reg2_i] ==
       stack1[ip1][spmap1[ip1]][reg3_i])      //the value of x is the
       value of i (before adding)
19 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
       (stack1[ip1][spmap1[ip1]][reg3_i] <
       stack1[ip1][spmap1[ip1]][param1_i])       //i runs until n
20 stack1[ip1][spmap1[ip1]][place] == oldInLoop ==>
       (stack1[ip1][spmap1[ip1]][reg3_i] >= 0)                          //i
       starts at 0
21 stack2[ip2][spmap2[ip2]][place] == newBeforeReturn ==>
       (if(stack2[ip2][spmap2[ip2]][param1_i] >= 0) then
       (stack2[ip2][spmap2[ip2]][reg2_i] ==
       stack2[ip2][spmap2[ip2]][param1_i]) else
       (stack2[ip2][spmap2[ip2]][reg2_i] == 0))
22 <<<
```

Figure 6.12: Complete specification of the loopFormula example

```
1  public interface MyList {
2          public void set(int i, Object o);
3
4          public Object get(int i);
5  }
6
7  public class C{
8          private MyList list;
9
10         public void setList(MyList list){
11                 this.list = list;
12         }
13
14         public void m(){
15                 for(int i=0; i<5; i++){
16                         list.set(i, new Object());
17                 }
18                 return;
19         }
20 }
```

Figure 6.13: Old implementation of recursiveLoop

6.4.2 Recursive Methods

For examples comparing an implementation using a loop with an example using a recursive method to compute the same value, the executions of both implementations can not directly be synchronized. The example recursiveLoop shows such a case.

The code of the old implementation of recursiveLoop is shown in Fig. 6.13. The code is identical with the simpleLoop example from Section 6.2. The loop adds five new objects to a list. The same computation is performed by the new implementation of recursiveLoop shown in Fig. 6.14, this time using a private recursive method instead of a loop. The iterations of the loop in the old implementation can be synchronized with the executions of the loop-method in the new implementation called in lines 15 and 23. The boundary calls to MyList.set() in line 16 in the old implementation and line 22 in the new implementation are equal. The situation after performing the last iteration of the loop in the old implementation and the last recursive call in the new implementation is as follows: The old implementation breaks out of the loop and the computation is complete. The new implementation has built up stack frames of the method loop(), one for each recursive call. These stack frames have to be removed from the stack by the method returns of method loop() to come to the same situation as in the old implementation. The execution of the old implementation is stalled during execution of the method returns of the new implementation.

```
1  public interface MyList {
2          public void set(int i, Object o);
3
4          public Object get(int i);
5  }
6
7  public class C{
8          private MyList list;
9
10         public void setList(MyList list){
11                 this.list = list;
12         }
13
14         public void m(){
15                 loop(0);
16                 return;
17         }
18
19         private void loop(int i){
20             if(i>=5)
21                 return;
22             list.set(i, new Object());
23             loop(i+1);
24             return;
25         }
26 }
```

Figure 6.14: New implementation of recursiveLoop

```
1  (forall o1,o2: Ref :: ObjOfType(o1, $C, heap1) &&
2                        ObjOfType(o2, $C, heap2) &&
3                        related[o1, o2] ==>
4     RelNull(heap1[o1, $C.list], heap2[o2, $C.list], related))
5
6  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==>
7     (stack1[iframe][spmap1[iframe]][place] ==
8      lib1_C.m_set$int$java.lang.Object_0) ==>
9        (stack2[iframe][spmap2[iframe]][place] ==
10        lib2_C.loop$int_set$int$java.lang.Object_0))
11 (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==>
12    (stack1[iframe][spmap1[iframe]][place] ==
13     lib1_C.m_set$int$java.lang.Object_0) ==>
14       (stack1[iframe][spmap1[iframe]][reg1_i] ==
15        stack2[iframe][spmap2[iframe]][param1_i]))
16 (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==>
17    (stack1[iframe][spmap1[iframe]][place] ==
18     lib1_C.m_set$int$java.lang.Object_0) ==>
19       (spmap1[iframe] == 0 && spmap2[iframe] ==
20       (spmap1[iframe] + 1 + stack1[iframe][spmap1[iframe]][reg1_i])))
```

The loop iterations in the old implementation are synchronized with the executions of the body of method loop() in the new implementation as already described for the simpleLoop example. The field C.list holds corresponding objects such that the boundary calls to MyList.set() are equal (lines 1 to 4). The boundary call is performed in the new implementation when it is performed in the old implementation (lines 6 to 10). This time, we do not have two loop counters with equal value. Instead, the value of the loop counter i in the old implementation and the parameter to the loop() method are equal (lines 11 to 15). The number of stack frames is not equal in this example. Instead, the number of stack frames built up by recursive calls to the method loop() is related to the loop counter i in the old implementation (lines 16 to 20). For each iteration of the loop in the old implementation a new stack frame is created in the new implementation.

```
1 afterLoop = old 12 (true)
2 endLoop = new 18 (true) (spmap2[ip2])
3 afterRec = new 10 (true) (spmap2[ip2])
```

Three local places are used to asynchronously execute the method returns of loop() after the computation is completed. The place afterLoop is located after the loop and before the return in method m() in the old implementation. In the new implementation, place endLoop is at the end of the recursive method loop() before the return statement and the place afterRec is after the first call to loop() in method m(), which corresponds to the place afterLoop in the old implementation. The old implementation is stalled in place afterLoop to execute the returns of the recursive method invocations. To verify the correct termination of this asynchronous execution of the new implementation a strictly decreasing measure is required for

the places in the new implementation. Each return of method loop() results in a decrease of the stack pointer of the new implementation, thus, the stack pointer spmap2[ip2] can be used as measure in this case (lines 2 and 3).

```
1  stack1[ip1][spmap1[ip1]][place] == afterLoop <==>
2     ((stack2[ip2][spmap2[ip2]][place] == endLoop) ||
3      (stack2[ip2][spmap2[ip2]][place] == afterRec))
4  stack2[ip2][spmap2[ip2]][place] == endLoop ==>
5      (spmap2[ip2] >= 1 && spmap2[ip2] <= 6)
6  stack2[ip2][spmap2[ip2]][place] == endLoop ==>
7      (stack2[ip2][0][meth] == $m)
8  (forall o1,o2: Ref :: ObjOfType(o1, $C, heap1) &&
9                        ObjOfType(o2, $C, heap2) &&
10                       related[o1, o2] ==>
11      RelNull(heap1[o1, $C.list], heap2[o2, $C.list], related))
```

The old implementation is at place afterLoop whenever the new implementation is either at place endLoop or at place afterRef (lines 1 to 3). The loop iterations are statically known such that the stack frame count can also be given in the coupling invariant. When the execution of the new implementation is at place endLoop in method loop() the stack pointer spmap2[ip2] is greater or equal to one (lines 4 and 5), since the method is called from method m() in the first stack frame 0 (lines 6 and 7), and is less or equal to six (line 5), since five calls to loop() are performed in total. The global invariant that the list implementations in field C.list are related to is not destroyed during the execution of method m() (lines 8 to 11). The precondition we already stated for the simpleLoop example is that the context has to set the list used by the implementations before calling method m(). Therefore, we omit the null checks as in the simpleLoop example.

```
1  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
      ==>
2     (stack1[iframe][spmap1[iframe]][place] ==
3      lib1_C.m_set$int$java.lang.Object_0) ==>
4        (stack1[iframe][spmap1[iframe]][reg1_i] <= 5))
5  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
      ==>
6     (stack1[iframe][spmap1[iframe]][place] ==
7      lib1_C.m_set$int$java.lang.Object_0) ==>
8        (stack1[iframe][spmap1[iframe]][param0_r] ==
9         stack1[iframe][0][param0_r] &&
10        stack2[iframe][spmap2[iframe]][param0_r] ==
11        stack2[iframe][0][param0_r]))
```

The global coupling invariant has to be extended. The loop counter in the old implementation is less or equal to five (lines 1 to 4). This information is needed to resynchronize the executions at the places afterLoop in the old implementation and endLoop and afterRef in the new implementation. Because the verification procedure is interrupted when reaching a local place, the information about aliasing is lost. The most important aliasing information is that the receiver of the method calls,

```
1  >>>invariant
2  (forall o1,o2: Ref :: ObjOfType(o1, $C, heap1) && ObjOfType(o2, $C,
       heap2) && related[o1, o2] ==> RelNull(heap1[o1, $C.list],
       heap2[o2, $C.list], related))
3
4  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==>
       (stack2[iframe][spmap2[iframe]][place] ==
       lib2_C.loop$int_set$int$java.lang.Object_0))
5  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==>
       (stack1[iframe][spmap1[iframe]][reg1_i] ==
       stack2[iframe][spmap2[iframe]][param1_i]))
6  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==> (spmap1[iframe] == 0 &&
       spmap2[iframe] == (spmap1[iframe] + 1 +
       stack1[iframe][spmap1[iframe]][reg1_i])))
7  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==>
       (stack1[iframe][spmap1[iframe]][reg1_i] <= 5))
8  (forall iframe: int :: 0<=iframe && iframe<=ip1 && iframe % 2 == 1
       ==> (stack1[iframe][spmap1[iframe]][place] ==
       lib1_C.m_set$int$java.lang.Object_0) ==>
       (stack1[iframe][spmap1[iframe]][param0_r] ==
       stack1[iframe][0][param0_r] &&
       stack2[iframe][spmap2[iframe]][param0_r] ==
       stack2[iframe][0][param0_r]))
9  <<<
10 >>>local_invariant
11 stack1[ip1][spmap1[ip1]][place] == afterLoop <==>
       ((stack2[ip2][spmap2[ip2]][place] == endLoop) ||
       (stack2[ip2][spmap2[ip2]][place] == afterRec))
12 stack2[ip2][spmap2[ip2]][place] == endLoop ==> (spmap2[ip2] >= 1 &&
       spmap2[ip2] <= 6)
13 stack2[ip2][spmap2[ip2]][place] == endLoop ==> (stack2[ip2][0][meth]
       == $m)
14 (forall o1,o2: Ref :: ObjOfType(o1, $C, heap1) && ObjOfType(o2, $C,
       heap2) && related[o1, o2] ==> RelNull(heap1[o1, $C.list],
       heap2[o2, $C.list], related))
15 <<<
16 >>>preconditions
17 useHavoc[lib1_C.m_set$int$java.lang.Object_0] := false;
18 useHavoc[lib2_C.loop$int_set$int$java.lang.Object_0] := false;
19 stall1[afterLoop, endLoop] := true;
20 <<<
21 >>>places
22 afterLoop = old 12 (true)
23 endLoop = new 18 (true) (spmap2[ip2])
24 afterRec = new 10 (true) (spmap2[ip2])
25 <<<
```

Figure 6.15: Complete specification of the recursiveLoop example

param0_r, is equal in all stack frames (lines 5 to 11). Only methods on the same object are called.

The complete specification of recursiveLoop is shown in Fig. 6.15. The example combines many of the problems already shown in other examples such as synchronization of iterations, boundary calls, and stalled execution. The Boogie model can be generated and verified using the following command in the installation folder.

```
1 ./bin/bcv --compile -N --specification
    examples/recursiveLoop/bpl/spec.bsl --libs
    examples/recursiveLoop/old examples/recursiveLoop/new
    --loopUnroll 4 --iframes 1
```

Chapter 7

Related Work

This chapter introduces some tools that also implement verification techniques based on behavioral equivalence or backward compatibility. Additionally, the ASTOOT approach, a testing framework for object-oriented programs, is described in short, a framework whose ideas could be used to implement testing of abstract data types written in an object-oriented language using BCVerifier.

7.1 Product Programs

Backward compatibility is a property between two programs that can be verified using different techniques. Barthe et al. [10] present in their paper an approach to verify properties about two programs or two runs of the same program that does not rely on side-by-side simulation of both programs. Instead, the code is merged into a single routine. For this to work, the programs must be separable, meaning the set of variables used in one program is distinct from the set of variables used in the other program. This can be achieved by substituting every variable v by its primed version v' in one program as long as primed variable names are not valid in the underlying programming language. The approach soundly transforms relational verification tasks into standard ones. The validity of the Hoare-triple $\vDash \{\bar{\varphi}\}c\{\bar{\psi}\}$ implies $\vDash \{\varphi\}c_1 \sim c_2\{\psi\}$. φ and ψ are relations on the states of programs c_1 and c_2. $\bar{\varphi}$ and $\bar{\psi}$ are predicates on the states of program c. $c_1 \sim c_2$ is the parallel execution of programs c_1 and c_2 and c is the product program that simulates the execution steps of its constituents. Since the variable sets of c_1 and c_2 are disjoint, the predicates describing the relations between c_1 and c_2 can be set equal to the predicates on the states of c, $\bar{\varphi} \equiv \varphi$ and $\bar{\psi} \equiv \psi$. The domain of the programs is restricted to non-stuck programs. The product program c is constructed by interweaving equal commands of both programs. The body of loops that have an equal guard on both sides are merged and the loop guards are synchronized using an assertion. This makes sure that the same number of iterations are performed by both loops. Loops that do not have equal guards on both programs to be composed can not be directly merged and have to be composed sequentially. Optimizations such as unrolling the loop can help to synchronize loops.

The goal of the approach of interweaving programs is to cope with the problem that arises by sequentially composing programs, $c_1; c_2$, know as self-composition [11]. To prove properties about this composed program $\vDash \{\varphi\}c_1; c_2\{\psi\}$ it is required to prove and verify an intermediate assertion that corresponds to the functional specification of the program c_1 [24], $\vDash \{\varphi\}c_1\{\phi\}$ and $\vDash \{\phi\}c_2\{\psi\}$. This proof is not feasible to do by automatic tools in general. The deviation of the states of two programs that compute the same result is assumed to be small after one or few steps in the execution of both programs. This is obvious for equal programs and programs that only differ in few execution steps but may grow if the programs differ more. For examples of composed product programs we refer to the paper of Barthe et al. [10].

The described approach is based on a simple imperative language. Methods and tools that automatically generate product programs for given input programs are not yet available. The authors are convinced to be able to generalize the approach to programs written in different languages and to accommodate to nondeterminism and dynamic allocation.

To generalize the approach to be usable to prove backward compatibility of class libraries, the state of the programs has to be adapted to represent a call stack of the programs. One major problem is to find corresponding methods to merge. Dynamic method dispatch would have to be resolved statically such that the composition of each target method on one side could be composed with each target method on the other side. A pre-selection can be made based on the method name. More complex information about subtyping and inheritance would complicate the generation process since complex reasoning would be needed to find valid pairs of methods. Using a simpler approach to choose method pairs would lead to quadratically many programs to be verified in the worst case. Additionally, the handling of method invocations in the body of methods suffers from the same problem. This approach is not as suitable as our approach to integrate the method resolution into the proving process itself. The idea of relating the state of both programs after a few steps instead of executing the complete methods one after the other can be used in our implementation as needed by defining local places and relating the state of both programs in the local invariant before continuing the execution.

7.2 Backward Compatibility of Microcode

Backward compatibility is a topic not only of class-libraries but also of microcode used in modern processors. Arons et al. [9] present an approach for verifying backward compatibility of microcode programs . Modern processor architectures appear to developers as CISC (Complex Instruction Set Computing) CPUs. Internally, the processors work more like RISC (Reduced Instruction Set Computing) CPUs in the sense that the complex instructions are translated into a rather small set of instructions the hardware understands. The layer between the archi-

tecture and the hardware layer is called microcode layer. A significant portion of the implementation of new technologies in modern processors is done in microcode. Although the new processor generation may feature new technologies which clear the way for optimization and performance gain, legacy software must still work without modification on the new CPU. Backward compatibility of microcode programs is defined as equivalence under restrictions that disable the new functionalities.

The microcode programs are translated into an operationally equivalent model in an intermediate representation language (IRL). The IRL model is translated by the tool called MircoFormula to verification conditions. These verification conditions are checked by an SMT solver, currently MathSAT [15]. The formal model used is the model of exit-differentiated transitions systems. The registers of the processor are modeled as variables. The value of those registers are bit-vectors. The model captures the different outcomes of a microcode operation such as exceptions and correct termination. For each of these outcomes, called doors, an observation function defines the observable state of the operation, the subset of the registers, such as the result of a mathematical operation or the cause of an exception. Symbolic simulation is used to compute the effect of the program on a symbolic initial state. Two microcode programs are backward compatible if starting in the same symbolic initial state the same doors are reached and the observations at these doors are the same.

The inputs of MicroFormula are (1) IRL models of all microcode instructions which are templates used to translate microcode instructions to IRL, (2) the source and target microcode programs, and (3) a set of restrictions under which to verify compatibility. The information about the restrictions used to guide the verification process and the inner working of MicroFormula are not publicly available. The lifecycle reminds of our tool BCVerifier. The model of the microcode instructions corresponds to the model of the byte code instructions. In contrast to microcode instructions, the translation of Java byte code is rather fixed in relation to the library implementations under consideration. The source and target microcode programs correspond to the two implementations of the library. The restrictions of the compatibility verification could be the relation between the internal state of the microcode translations, which would correspond to the coupling invariant we use in BCVerifier. The outcomes called doors in MicroFormula correspond to the method calls and the method return a method of the library implementation can perform and the observation functions define the internal state that is relevant for an outcome. The observation functions correspond to the parameters for method calls and the result value for a method return.

MicroFormula is develop by Intel in cooperation with the academic community since 2003 and is still under development as of 2010 [15]. A working version providing core functionality was implemented in 2005, the last published information about the project dates back to 2010 [15].

7.3 ASTOOT Testing Framework

The ASTOOT approach to testing object-oriented programs [14] is a testing framework focused on testing the implementation of abstract data types (ADT). ADTs consist of an internal state and operations that manipulate or expose specific portions of the state. Operations that expose portions of the state of an ADT are called observers. Observers have no side effects, neither on the state of the ADT nor on its parameters. The state manipulating operations have no side effect on there parameters, either. These conditions are given for the specification language used in the approach called LOBAS, but the restrictions make sense for ADTs in general. Two objects are observationally equivalent if it is impossible to distinguish them using the operations defined on the class of the object or one of its superclasses.

Test cases of the ASTOOT framework are given in form of sequences of messages, along with tags indicating whether these sequences should put objects of this class in an equivalent state. These messages trigger the creation of an object of the specified class or operations on the existing instance of the class. An example of such a sequence is create.add(5) .delete.add (3) for a list implementation. The resulting list should contain the integer 3 as only element. An equivalent state should be reached by the sequence create.add(3). The test cases can be written manually, the paper of Doong and Frankl [14] also includes a description of a method to generate the sequences of test cases from an algebraic specification of the abstract data type. Test drivers that can be used to execute the test cases can be generated from the interface of the classes. A user-supplied method checking the equivalence of two objects is used to reason about the observational equivalence of two instances of the ADT after execution of the respective sequences of operations.

The approach is based on the programming language Eiffel [4] but can be adopted to other object-oriented languages. The tests can be executed automatically, the expected result is included in the test case in concise format as a boolean value. The major problem of the approach is the equivalence checking operation that has to be supplied by the user. An error in this operations leads to invalid test runs and false results.

The BCVerifier tool could be adapted to work as a test driver as well as an equivalence checker. Instead of supplying an invariant that describes the relation between the state of two implementations after execution of equal steps, the operations defined in the test case are executed on two separate implementations of the class. Afterwards the observational equivalence of both states could be checked simulating the same steps on both implementations and comparing the outcome of the observers of the class in a symbolic simulation, as it is done in the backward compatibility check.

7.4 Mutual Summaries

Hawblitzel et al. [17] describe in their paper a contract mechanism for comparing two programs. The approach generalizes single program contracts such as preconditions and postconditions. The language used as a basis for the approach is an imperative language with integer variables, binary operations on integers, boolean relational operations and quantification, sequential composition, and nondeterministic choice. Variables can represent global as well as procedure-local state. Procedures can have classical preconditions and postconditions. A mutual summary describes the relation between the parameters, the global variables in the pre and post state of the execution, and the results of two procedures. For any pair of procedures $(p1, p2)$, the executions of $p1$ and $p2$ should satisfy the mutual summary $MS(p1, p2)$ of the procedures.

Boogie is used as intermediate verification language. The procedures are transformed into Boogie procedures. Uninterpreted function symbols R_p are added to the model for each procedure p to represent the execution this particular procedure including the parameters, global state in pre and post state, and the result of the procedure. A free postcondition is added to each procedure p that the execution symbol R_p is fulfilled after the execution of the procedure. Free postconditions are not checked in the verification process, they are assumed to hold after the execution of a procedure. Additionally a Boogie procedure $MUTUALCHECK_{p1,p2}$ is generated for each pair of procedures $(p1, p2)$ which checks the conformance to the mutual summary. The procedure consists of three parts. (1) The procedure $p1$ is inlined, the precondition, the body and the postcondition are added to the checking procedure. (2) The procedure $p2$ is also inlined as described before. (3) An assert is added to the procedure that checks that the mutual summary procedures $p1$ and $p2$ is fulfilled. The relation between the executions of two procedures and their mutual summary is added as an axiom

$$
\begin{aligned}
&R_{p1}(params1, old(globals1), result1, globals1) \wedge \\
&R_{p2}(params2, old(globals2), result2, globals2) \implies \\
&MS(p1, p2)(params1, params2, old(globals1), old(globals2), \\
&\quad result1, result2, globals1, globals2)
\end{aligned}
$$

The function symbol R_{p1} represents the terminating execution of $p1$, R_{p2} represents the terminating execution of $p2$. $MS(p1, p2)$ represents the mutual summary of procedures $p1$ and $p2$. For all pairs of procedures that have no explicitly given mutual summary, the mutual summary is equal to *true*. At each call site of a procedure, the free postcondition introduces the facts needed to derive from this axiom that the mutual summary between both called procedures needs to hold. This fact is used to derive the assertion that the mutual summary holds after the execution of both procedures $p1$ and $p2$ in the procedure $MUTUALCHECK_{p1,p2}$.

Using the technique described above, the order of procedure calls in two procedures that perform the same computation is not relevant. The fact that the mutual summary between both procedure calls holds is only dependent on the state of the

program before and after the procedure call. For two procedures P and Q, if the first implementation first calls P and the second implementation first calls Q, the mutual summary approach compares the program state. For a simulation based approach such as ours it is much more difficult to relate the procedure calls, since the simulation first reaches calls to P and Q, which are not related, and has to stall one of the executions. Afterwards the first implementation calls Q, but this call is related with the call to Q of the second implementation, which was skipped. The simulation would have to track back or try a second execution order.

Chapter 8

Conclusion

In this thesis we developed a backward compatibility checker that can verify backward compatibility of Java libraries. The theoretical model used by the checker is the trace-based semantics introduced by Welsch and Poetzsch-Heffter [26]. We support a larger subset of Java than the theoretical model including booleans, integers and mathematical operations. The implementation of the model is tailored to support a flexible representation of heap and stack as internal state of the library implementation and also supports loops and multiple stack frames to support method calls. Examples illustrate that relations between different kind of recursive computations can be expressed in the coupling invariant relating the inner state of two implementations of a library which can be checked by our tool.

The expressions used in the invariant specifications are complex and coupling invariants are hard to write in the current implementation of the checker. A specification language supporting the developer in writing these specifications based on the code of the library implementations would reduce the learning curve and allow for automatic generation of well-definedness properties for the invariant. Relating two implementations that perform the same subtasks in different order is not easily supported by the chosen approach since it is based on symbolic simulation of the resulting byte code. As of yet it is not clear how to handle these kind of relations in our implementation. Relating an implementation that uses a single loop and an implementation that performs the same computation using two independent loops is also not well supported, since the execution of the single loop would have to be split into two independent executions that are related to each of the two loops in the other implementation. This is not supported by the implementation as of yet.

Very well supported are implementations that are structurally equivalent with respect to recursive computation. Relating two loops performing the same computation can mostly be reduced to synchronizing the loop counters or the loop conditions. Different possible targets for internal method calls are automatically found and checked. The decision about possible boundary method calls are automatic and can be influenced using the coupling invariant. If computations are too complex for Boogie to prove them automatically it is possible to interrupt the prov-

ing process and insert facts about the relation of the internal state of methods to assist the automatic processing. We are confident that our implementation could be modified to infer most of the statements of the coupling invariant for structurally very similar implementations as they are encountered when modifying a library implementation. For identical classes it should be possible to completely generate the coupling invariant.

8.1 Future Work

Observable behavior is defined to consist only of method invocations and returns from context to library or vice versa. We do not consider exceptions that can be thrown by a method or direct access to public fields of objects.

Exceptional behavior is difficult to relate. For the question of terminating and non-terminating programs, we make the assumption that non-termination means that the behavior of this non-termination program part is not defined. This also means that we say nothing about the behavior of the new implementation if the old implementation did not terminate for the given parameters.

For exceptional behavior, the situation is quite different. If the first implementation did throw an exception but the second does not anymore, what can we say about the backward compatibility of the behavior of this library? One way to look at the situation would be to say, the behavior is not compatible. This statement would be correct for many cases. But sometimes, not throwing an exception in the new implementation would be considered to be backward compatible, especially since we want to be able to extend the behavior of the implementation. An operation that was not supported previously but is supported in the new implementation would sometimes be considered to be backward compatible.

One possible solution would be to extend the possible labels of program interactions for the trace-based semantics and add the possibility to define, which of the two possibilities should be considered backward compatible.

Considering write-access to public fields of an object, this operation is not possible and therefore not considered in LPJava, the language considered by the trace-based semantics. The special problem is to somehow capture this behavior in means of operations on the program configuration, since writing to a field of an object is clearly observable. One possible solution would be to compare the program state before the execution of an operation with the program sate after the operation has been performed. The behavior would be related if for all fields of related objects that have been changed by the operation the values of these fields are related afterwards. A very similar approach could be implemented for arrays, since an array can be seen as an object with only public fields addressed by an index.

Since our proving procedure is based on simulation and places, we cannot easily relate two implementations that are structurally very different but perform the

same operation. What could be done, is to include classical specification based reasoning for single parts of a program, that establish the conditions required to show the relation between the two library implementation on a trace basis. Two places would be defined in each program, the behavior of the operations between these places would be specified in the classical way using an abstract specification. The simulation of both programs would stop when reaching the first points in the programs. The abstract specification would be checked against the behavior of the operations for each implementation. Afterwards the simulation could be continued based on the program state the abstract specification on both sides describe.

Most non-academic examples are implemented using the Java class library. To verify the backward compatibility of such implementations using BCVerifier, the subset of Java we support would have to be extended. Other than access to fields and arrays, as described above, the semantics of a limited subset of the Java runtime should be directly supported by the tool. Some static methods of built-in types, such as String and Integer, have a quite complex implementation. On the other hand, some basic properties of these methods are often enough to prove backward compatibility. String.equals() and Integer.parseInt() for example could be approximated by functions StringEquals(s1: Ref, s2: Ref) **returns** (**bool**) and ParsedInt(s: Ref) **returns** (**int**). One relation between both methods is that if two strings are equal, the parsed integers are equal as well. This property is expressed by the following axiom:

```
1 axiom (forall s1: Ref, s2:Ref ::
2     isOfType(s1, heap1, $java.lang.String) &&
3     isOfType(s2, heap1, $java.lang.String) &&
4     StringEquals(s1, s2) ==>
5     ParsedInt(s1) == ParsedInt(s2))
```

Approximating the semantics of the most common methods of the Java class library would make it possible to verify backward compatibility of non-academic examples without adding the complete Java runtime as part of the example library.

Additionally, what is not yet supported by our implementation but could be added quite easily is support for **float** and **double**. The byte code instructions that are needed to handle these types are not supported by B2BPL. Support would have to be added from scratch.

For users unexperienced with Boogie to understand the errors returned by BCVerifier, the error reporting needs to be enhanced. At the moment, the error trace of Boogie is returned as an error message. Since the labels in the generated Boogie code are named after the methods, tracing such an error is mostly possible without looking at the code. But for details, which statement could not be checked, what this statement belongs too such as the global invariant, local invariant, or loop unrolling check, the user will still have to look up the exact line in the generated source. This could be improved by giving detailed error messages about

what went wrong. For this to be possible, one would have to trace back from lines of the generated Boogie source code to the statements of the invariants and checks.

Bibliography

[1] ASM. URL http://asm.ow2.org. [Online; accessed August 3th 2012].

[2] Chalice - ETH - Chair of Programming Methodology. URL http://www.pm.inf.ethz.ch/research/chalice. [Online; accessed August 3th 2012].

[3] Dafny: a language and program verifier for functional correctness. URL http://research.microsoft.com/en-us/projects/dafny/. [Online; accessed August 3th 2012].

[4] Eiffel. URL http://www.eiffel.com. [Online; accessed September 27th 2012].

[5] Java SE Specifications. URL http://docs.oracle.com/javase/specs/. [Online; accessed June 15th 2012].

[6] Spec#. URL http://research.microsoft.com/en-us/projects/specsharp/. [Online; accessed August 3th 2012].

[7] Z3: Theorem Prover. URL http://research.microsoft.com/en-us/um/redmond/projects/z3/. [Online; accessed August 3th 2012].

[8] *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, 2009. ACM. ISBN 978-1-60558-497-3.

[9] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 185–198. Springer, 2005. ISBN 3-540-27231-3. URL http://dx.doi.org/10.1007/11513988_20.

[10] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011. ISBN 978-3-642-21436-3. URL http://dx.doi.org/10.1007/978-3-642-21437-0.

[11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6): 1207–1252, 2011. URL http://dx.doi.org/10.1017/S0960129511000193.

[12] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for java bytecode. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2007. ISBN 978-3-540-71288-6. URL http://dx.doi.org/10.1007/978-3-540-71289-3_18.

[13] Danny Dig and Ralph E. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance*, 18(2):83–107, 2006. URL http://dx.doi.org/10.1002/smr.328.

[14] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol*, 3(2):101–130, 1994. URL http://doi.acm.org/10.1145/192218.192221.

[15] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying SMT in symbolic execution of microcode. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 121–128. IEEE, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5770940.

[16] Benny Godlin and Ofer Strichman. Regression verification. In *DAC* con [8], pages 466–471. ISBN 978-1-60558-497-3. URL http://dl.acm.org/citation.cfm?id=1629911.

[17] C. Hawblitzel, M. Kawaguchi, S.K. Lahiri, and H. Rebêlo. Mutual summaries: Unifying program comparison techniques. *This page intentionally left (not quite) empty.*, page 40, 2011.

[18] Hermann Lehner and Peter Müller. Formal translation of bytecode into boogiePL. *Electr. Notes Theor. Comput. Sci*, 190(1):35–50, 2007. URL http://dx.doi.org/10.1016/j.entcs.2007.02.059.

[19] K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 615–622. ACM, 2009. ISBN 978-1-60558-166-8. URL http://doi.acm.org/10.1145/1529282.1529411.

[20] Ovidio José Mallo. A translator from BML annotated Java Bytecode to BoogiePL. Master's thesis, ETH Zurich, March 2007.

[21] Arnd Poetzsch-Heffter and Peter Müller. Logical foundations for typed object-oriented languages. In *Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, PROCOMET '98, pages 404–423, London, UK, UK, 1998. Chapman & Hall, Ltd. ISBN 0-412-83760-9. URL http://dl.acm.org/citation.cfm?id=647321.721346.

[22] K. Rustan and M. Leino. *This is Boogie 2*, 2008.

[23] Alex Suzuki. Translating java bytecode to BoogiePL. Master's thesis, ETH Zurich, October 2006.

[24] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005. ISBN 3-540-28584-9. URL http://dx.doi.org/10.1007/11547662_24.

[25] Yannick Welsch and Arnd Poetzsch-Heffter. Full abstraction at package boundaries of object-oriented languages. In Adenilso da Silva Simão and Carroll Morgan, editors, *Formal Methods, Foundations and Applications - 14th Brazilian Symposium, SBMF 2011, São Paulo, Brazil, September 26-30, 2011, Revised Selected Papers*, volume 7021 of *Lecture Notes in Computer Science*, pages 28–43. Springer, 2011. ISBN 978-3-642-25031-6. URL http://dx.doi.org/10.1007/978-3-642-25032-3.

[26] Yannick Welsch and Arnd Poetzsch-Heffter. A fully abstract trace-based semantic for reasoning about backward compatibility of class libraries. submitted for journal publication, April 2012.

[27] Yannick Welsch and Arnd Poetzsch-Heffter. Verifying backwards compatibility of object-oriented libraries using Boogie. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 35–41, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1272-1. doi: 10.1145/2318202.2318209. URL http://doi.acm.org/10.1145/2318202.2318209.

[28] Samuel Willimann. An automated program verifier for Java bytecode. Master's thesis, ETH Zurich, September 2007.