Proc. 2nd International Workshop on State Of the Art in Java Program analysis (SOAP '13)—Seattle, WA

Joogie: From Java through Jimple to Boogie

Stephan Arlt
United Nations University, IIST
arlt@iist.unu.edu

Philipp Rümmer
Uppsala University
philipp.ruemmer@it.uu.se

Martin Schäf
United Nations University, IIST
schaef@iist.unu.edu

Abstract

Recently, software verification is being used to prove the presence of contradictions in source code, and thus detect potential weaknesses in the code or provide assistance to the compiler optimization. Compared to verification of correctness properties, the translation from source code to logic can be very simple and thus easy to solve by automated theorem provers. In this paper, we present a translation of Java into logic that is suitable for proving the presence of contradictions in code. We show that the translation, which is based on the Jimple language, can be used to analyze real-world programs, and discuss some issues that arise from differences between Java code and its bytecode.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Assertion checkers, assertion languages, performance

General Terms Algorithms, Performance, Theory, Verification

Keywords Inconsistency, infeasible code, intermediate verification languages

1. Introduction

During the software development process, engineers use a variety of tools to keep code quality high, and error rate low. In the early stages of coding, when the source code is still in the making, simple tools can be applied that detect common mistakes like unreachable code or contradictory control flow, without having full information about the program environment or the purpose of the code available. These tools detect what we refer to as *infeasible code*, by showing that a piece of code has no feasible execution. Infeasible code detection comes with the benefit that it can be done on code fragments where the full context of its execution is not yet known. That is, a piece of code without any feasible execution within a given context will still not have a feasible execution in a larger context, as adding additional statements around the fragment can only reduce the feasible executions of this fragment. Hence, it is possible to detect these contradictions on incomplete code fragments without running the risk of introducing false alarms. Popular instances of such tools are, for example, the static analysis in IDE's such as Eclipse, Visual Studio, or Xcode that warn the programmer about unreachable or contradicting control-flow by underlining the code in question with a squiggly line.

Recently, techniques have been introduced to detect infeasible code using static verification [1, 11, 20]. From a static verification perspective there are several interesting aspects to this problem. First, proving the presence of infeasible code on program fragments, such as procedures, without considering its context lifts (to some extent) the notorious scalability problem of static verification. Second, to detect contradicting control-flow, static verification has to prove the absence of (feasible) executions of the code fragment under consideration. This is, in some way, dual to verification and only a proof guarantees the presence of a contradiction, whereas a counter-example just witnesses a feasible execution of the code fragment. Hence, if a tool fails to prove a certain statement to be infeasible, it just remains silent and does not bother its user with false alarms (soundness). And third, abstractions which might be needed by static verification tools only have to preserve the feasible executions of the original program in order to detect contradictions. This is a significant difference to static verification that proves correctness, which has to preserve the infeasible executions. For example, for detecting contradictions, it is almost always sound to directly map integer types in Java to unbounded/mathematical integers in logic (as long as the programmer does not explicitly catch arithmetic exceptions), whereas this abstraction would be too coarse to prove correctness, and additional axioms would be required.

Preserving feasible executions is a much simpler task than preserving infeasible executions. For detecting infeasible code, a coarse but fast abstraction might cause some false negatives, whereas a coarse abstraction in static verification might make a proof of correctness impossible.

We show how we use Soot [22] to build a translation from Java into Boogie [2] that preserves the feasible executions of the Java program. We report on how this translation, which is integrated in Joogie [1], a tool to detect infeasible code, can be used to detect problems in real-world software. In Section 2, we formalize the idea of detecting infeasible code and state our soundness requirement for the translation. In Section 3, we give an overview of the Boogie language, and show how a simple Burstall-Bornat heap model [3] can be used as a sound memory model for Java. For the actual translation from Java to Boogie, we use the Jimple intermediate representation of Soot. The translation of each Jimple statement into Boogie is explained in Section 4. Exception handling is discussed in Section 5. Some problems that arise from using bytecode or Jimple instead of Java source code are discussed in Section 6.

We evaluate our translation in Section 7 with Joogie [1], a fully-automated tool to detect infeasible code in Boogie programs. Joogie takes a Boogie program as input, computes a loop-free abstraction and then repeatedly queries a SMT solver to identify all statements that cannot occur on feasible executions. We apply Joogie with our translation to five real-world programs. Since Joogie works on a per-method basis, it scales almost linearly in the size of programs; in our experiments, Joogie is able to process programs with more than 100 kLOC in comparatively short time (less than one hour), without requiring any code annotations.

SOAP'13 June 20, 2013, Seattle, Washington, USA
Copyright © 2013 ACM ACM ISBN 978-1-4503-2201-0/13/06. . . $15.00

2. Preliminaries

Throughout the paper we consider Jimple programs, which are a 3-address intermediate representation of Java bytecode. Jimple provides the usual statements one expects from Java bytecode, such as assignments, conditional jumps, switch cases, and so on. A control-flow path π of such a program is a sequence of statements $\pi = s_0 \dots s_{n-1}$. For the sake of simplicity, we assume that control altering statements such conditional jumps like `if (e) goto L;` are represented by assume statements `assume(e)` or `assume(!e)` on a control-flow paths. The execution of an assume statement `assume(e)` either does nothing if `e` evaluates to `true`, or blocks otherwise. For the execution of all other statements, we use the standard semantics of Java.

We say a path is feasible if it has at least one execution from start to end. The feasibility of a path can be checked using the weakest-liberal precondition (**wlp**): the wlp $wlp(P, Q)$ of a program P w.r.t. a post-condition Q is the set of all states, such that an execution of P started from such a state either terminates in a state satisfying Q or does not terminate at all. Hence, a given path π is infeasible if and only if the formula $wlp(\pi, \mathit{false})$ is valid (for the computation of wlp, we refer to [6]). In other words, π is infeasible if from any pre-state, the execution of π either does not terminate normally or ends in a post-state satisfying false. Since the latter case cannot occur, $\models wlp(\pi, \mathit{false})$ only holds if π has no (normal terminating) execution.

To show the presence of infeasible code within a given code fragment, we have to show for some statement s that any complete path in this fragment that contains s is infeasible. That is, a contradiction exists if, for a given statement s in a program P, and the program P' which consists of all (possibly infinitely many) paths that contain s in P, the formula $wlp(P', \mathit{false})$ is valid. Like other reachability problems, checking the validity of this formula is not decidable for the general case. Hence, abstraction is needed: we are looking for a program $P^{\#}$ for which we can check the validity of $wlp(P^{\#}, \mathit{false})$. Further we want that $wlp(P^{\#}, \mathit{false})$ is valid only if $wlp(P', \mathit{false})$ is valid as a soundness criterion. That is, if we can find a contradiction in $P^{\#}$, we want to be sure that there is a contradiction in P' as well, so we do not report a false alarm.

This abstraction consists of two steps. One step abstracts from a possibly infinite number of control-flow paths in P' to a finite number in $P^{\#}$ by abstracting looping control-flow. This abstraction is discussed elsewhere (e.g., [12, 20]). The other part of the abstraction is how to represent the type system of Java in a logic that can be understood by an automatic theorem prover.

To meet our soundness criterion, we have to be sure that our abstraction over-approximates the set of feasible executions of P'. A brute-force solution to this would be to approximate P' by the logic formula true that describes all possible executions. This is sound but would not reveal any contradiction. A precise approximation that can reveal all contradictions might result in a very complex formula which cannot be solved by automatic theorem provers.

In this paper we present a solution that is not sound in theory, but produces reasonably efficient formulas and, in our experiments, does not produce false positives. We first show the sound subset of our translation, and then discuss the issue of soundness.

3. The Boogie Language

Boogie [16] is an imperative intermediate verification language (IVL) that has been used to encode and analyse a range of high-level programming languages. This section gives a high-level overview of the Boogie language (version 2), concentrating on the features that will be used to encode Jimple. For a more detailed definition we refer the interested reader to [16, 18].

A Boogie program is a collection of declarations, each of which can introduce *types, functions, constants, axioms, variables, procedures*, or *procedure implementations.*

Boogie types cover basic built-in data types like `bool` and `int`, as well as (monomorphic and polymorphic) map types. The latter are mainly used to represent language features such as arrays or heap (see below for further details). User-defined type constructors are introduced with the help of the `type` keyword; in the following example, `ref` is declared to be a nullary type constructor, `Field` to be a type constructor with one type argument (later used to represent fields of different types), and `IntField` is defined as a type synonym for the compound type `Field int`:

```
type ref;
type Field t;
type IntField = Field int;
```

Functions in Boogie are pure (i.e., total and free of side-effects), and mainly used to define background theories and language properties by means of axioms. For instance, a function mapping pairs of integers to integers can be declared with:

```
function pow(x: int, y: int) returns (z: int);
```

Since the semantics of functions is not restricted a priori, functions are also called *uninterpreted.*

Constants are functions without any arguments.

Axioms are used to restrict the interpretation of functions and constants. For instance, the semantics of the function `pow` can be specified through the following axiom:

```
axiom (forall x: int, y: int ::
       pow(x, 0) == 1 &&
       pow(x, y+1) == x*pow(x, y));
```

The expression language for axioms is the same as used for procedure implementations (see below), and includes all operations common in programming languages (such as literals, arithmetic and Boolean operations, function and map applications, etc.), as well as first-order quantifiers.

Variables are mutable (can be modified by procedures) and are used to represent the state and data of encoded programs. Variables can be declared *globally* or *locally* within the scope of a procedure.

Procedures are used to encode the executable parts of a program, including functions, methods, constructors, etc. In contrast to Boogie functions, Boogie procedures can have side-effects and might be partial, for instance due to indefinite looping. Procedures can have both input and output parameters, and can also specify *pre-conditions* (states in which the procedure can be called), *post-conditions* (states in which the procedure can terminate), and *modified global variables:*

```
var x : int;
var H : Heap;
procedure f(this : ref, y : int)
  returns (z : int);
  modifies x, H;
  requires y >= x;
  ensures  z != 0;
```

Procedure implementations describe the behavior of a procedure in an imperative style. Boogie supports both structured and unstructured code in implementations; in the scope of this paper we concentrate on the latter. The main statements available for unstructured programs are `goto` (to one or multiple target labels, selected non-deterministically), assignments to variables, `assume` and `assert`, `return` from a procedure, as well as `call` to other procedures; the semantics of each of the statements is as usual. It is clear that arbitrary control-flow graphs can be encoded in this language.

3.1 Modelling Object-Oriented Heap

Most of the basic Java data types can be mapped straightforwardly to Boogie data types. Classes, objects, and arrays can be modelled in an elegant way with the help of *polymorphic maps,* which represent polymorphic functions that can be updated by assignments. The heap of a program is modelled as a map from heap locations to values. A heap location (i.e., instance field) is uniquely specified by a *reference* of a class instance and a *field* of the class. Parametric polymorphism (in the Boogie model) is used to express that the type of the value stored at a location depends on the type of the field. It is common to use the following pattern:

```
type ref;
type Field t;

const x, y : Field int;
axiom (x != y);
const p : Field ref;

type Heap = <t>[ref, Field t]t;
```

This code defines a nullary type constructor `ref`, used for references, and a unary type constructor `Field` for instance fields. With the latter type constructor, constants `x`, `y` can be declared to represent (distinct) fields of type `int`, and `p` a field of type `ref`.

The type `Heap` is defined as a map from reference-field pairs to values, and used to explicitly represent heap by means of the variable `H`. The heap map is polymorphic in the type parameter `t`, expressing that the value stored at a `Field t` location has type `t`. The value of field `x` in an object `obj` is accessed using the map expression `H[obj, x]`; since `x` has type `Field int`, the type of `H[obj, x]` is `int`. The value of fields is updated using map update expressions like `H[obj, x := 42]`, returning a new heap (of type `Heap`) that coincides with `H`, with the exception that `42` is stored at the location ⟨`obj`, `x`⟩. Boogie also provides syntactic sugar to simplify updates as assignments in procedures.

Arrays, which are in Java allocated on the heap, can be modelled as objects with an implicit field that stores the array contents:

```
const $intElements : Field ([int]int);
const $refElements : Field ([int]ref);
```

If `ar` is a reference to an integer array, then an array element can be accessed using an expression like `H[ar,$intElements][5]`, and updated using
`H[ar,$intElements := H[ar,$intElements][5 := 42]]`.

4. Jimple to Boogie

With the memory model from the previous section, we can now translate most of the expressions and statements in a Jimple program in a straight forward manner.

Expressions As Jimple expressions are already simplified, their translation to Boogie is fairly simple. We translate Integer-typed constants directly to Boogie integers. As mentioned earlier this is not sound if the programmer explicitly catches arithmetic exceptions. In our experiments, however, this has never been the case. Integer arithmetic operations are handled by inlined functions. For the translation of the division operator, we further add an assertion to the Boogie program that guarantees the denominator is different from zero.

The prover we currently use in Joogie cannot handle real-values constants very efficiently. Therefore, we keep a map $rmap : double \rightarrow var$ during the translation that replaces each float or double constant d by unique constant variable $rmap[d]$. For arithmetic operations on values and variable of these types, we use uninterpreted functions. This is a coarse but sound abstraction. Depending on the capabilities of the prover, more precision can be added.

For strings and characters we also keep a map $smap : string \rightarrow var$ from variables to constant values to replace their occurrence by constant values. In addition to that, we keep a map $slen : var \rightarrow int$ that maps each string variable to its length. That is, each time a variable is assigned to a string constant str, we add two statements to the Boogie program. The first statement assigns the variable to the respective constant variable $smap[str]$, and the second one assigns the length of str to the appropriate field $slen[smap[str]]$. Any operation on string variables is translated into a non-deterministic assignment to this variable.

This string handling allows us simple comparisons, if strings have been assigned to constant values within the same scope. In fact, this has proved to be sufficient to detect cases where programmers deliberately render code unreachable in stable releases of code. Of course, unreachable code is not a heavy target; however, other analysis methods were not able to detect such cases of unreachable code in our experiments.

Array and Field Access. Array and Field access is modeled as described in the previous section. For arrays, we store the size of the array as the -1st element of the array. Each call to the operator `length` is translated into an access of this field. For each array access, we add an assertion that the index is in the given array bounds. For field access, we assert that the used object is allocated. The `instanceof` operator is also modeled by an uninterpreted function and returns an unknown Boolean value.

4.1 Statements

Assignments and Invocations With the above memory model and translation of Jimple expressions, assignments in Jimple can be directly translated into Boogie assignments. For procedure calls, we have to add surrounding statements to handle possible exceptions thrown by the callee. This is discussed in detail in Section 5.

Enter and Exit Monitor Monitor statements are not handled right now. This causes unsoundness if we expect the translation to handle this. We could as well say that this is the job of the static analysis. In any case, the unsoundness only causes false positives, if code is only reachable due to interleaving. In our experiments this only occurred once, where a trivial non-terminating loop was waiting for a response from the GUI.

Conditional Choices Conditional jumps are translated in a straight forward manner. Switch statements are first translated into nested-conditional choices and then translated accordingly. In Section 6 we discuss the soundness problems that arise from conditional choices.

Return and Throw statements Boogie uses specific return variables for procedures. Hence, we translate return statements to assignments of this variable. To model exceptional returning of a procedure caused by a `throw` statement, we use additional return variables that are only set if the procedure returns exceptional. As a procedure may throw different exceptions, we add one return variable for each type of exception that occurs in an uncaught throw. Exception handling is described in Section 5 in more detail.

5. Exceptions

The computation of verification condition in the presence of exceptions has been extensively studied (e.g., [10]). For preserving

```
void foo(C x, C y) {
  if (x == null &&  y ==
      null) {
  ...
  } else if (x == null &&
        y != null) {
  ...
  }
}
```

```
 if x != null goto l0;
 if y != null goto l0;
 ...
 goto label1;
l0:
 if x != null goto l1;
 if y == null goto l1;
 ...
l1:
  return;
```

```
if (x==null) {
  if (y==null) {
    goto 1;
  }
}
if (x==null) {
  if (y!=null) {
  } else {/*unreachable
      */}
}
1: return
```

Figure 1. Code snippet with `else if` statement, its corresponding Jimple code, and a reconstructed version of it showing the unreachable branch. Through the none mutual exclusive conditions that are needed to enter each of the cases, unreachable blocks might be created.

the feasible executions of a program, a worst-case analyze is sufficient. We traverse the call graph from its leaves to its source and check for each procedure which exception can possibly be thrown. As our analysis in Joogie cannot detect all infeasible (and hence unreachable) code, the set of possibly thrown exceptions is an over-approximation as well. We translate exceptional returns like regular returns, only that they assign to different return variables. For each possible exception inside a procedure, we create one additional return variable of appropriate type to the procedure. To avoid creating redundant return variables we keep a simple type hierarchy.

Each time we encounter a throw statement, we use the type hierarchy to check if there exists a `catch` block that can catch an exception of this, or any of its super-types. If so, we create a local variable that we assign to the created exception and create a control-flow edge to the `catch` block. For an uncaught exception in a `throw` statement, we create a return variable, assign it, and return from the procedure. Note that this only works because we can be sure, that the caller will not use the normal return value if an exceptional return value is set.

For call statements, we make use of the multi-assignments in Boogie, and translate each procedure call into a Boogie statement of the form:

```
ret, ex1,..,exN = call foo(arg1, argM)
```

where each exI stands for a possible exceptional return. Then insert conditional choice for each exceptional return variable where rethrow the exception, if the variable is not null.

`Finally`-clauses are already eliminated by the bytecode translation. This, however, causes some problems with our translation which will be discussed in the following section.

We do not collect all possible exceptions. We only collect those that are created by a `throw` statement, and those that are in the `throws` clause of procedures which are not in the scope of our analysis (e.g., library procedures). NullPointerExceptions, or array bounds violations are modeled through assertions. We can model them as exceptions as well, but for now, we do not see any benefit in it, besides the neater theory, and it will result in larger Boogie programs. In theory, using assertions instead of exception is unsound, as code might be unreachable if the programmer catches, e.g., null pointer exceptions. However, this is bad style and did not occur in our experiments.

6. Translation issues

Infeasible code detected on the bytecode level does not always have corresponding infeasible code on the Java level. Sometimes, a line of code is cloned during the translation to bytecode. This may cause false warnings if one of the clones is infeasible but the other clone is not. That is, there may be an infeasible instruction in the bytecode, but the corresponding instruction in the java code also maps to other bytecode instructions which are feasible. Thus, the bytecode may be infeasible while the Java code still is feasible.

Conditional branching and conjunctions. Figure 1 shows an example of infeasible code in a Jimple program that does not have corresponding infeasible in its Java program. The problem is introduced by the `else if` branch of the Java program in the left column of Figure 1, which is translated to the Jimple program in the middle column. The right column shows a Java program that is equivalent to the Jimple program which illustrates the unreachable code that has been introduced during the translation. This problem only arises for conditional choices with `else if` branches and branch conditions with conjunctions which are not disjunct from each other. One could explain the infeasible block as follows: If the `then` branch is not taken but `x` is `null` the `else if` branch must be taken in any case, and thus the check for `y` becomes useless. However, reporting this in the Java code would be clearly considered a false warning. A solution would be to rewrite the code such that first, `x` is checked to be `null`, and only if this holds `y` is checked in a nested `if`. However, this has to be done on the Java side even before the translation to Jimple.

Finally Clauses. An unavoidable source of false warnings that arises from using bytecode instead of source code are `finally`-blocks, see Figure 2. Let us assume that the procedure `creator` can throw an exception of type `MyException`. What happens in the bytecode and also later on in the Boogie program is that there is a path from the `try` block into the `finally` block on which we can assume that the exception was not thrown. And there is a path from the `try` through `catch` into `finally` where the exception is thrown. The problem is that the bytecode clones the `finally` block, such that the block that is reached from `try` is different from the one reached from `catch`. Hence, in the `finally` block reached from `catch`, `obj` must be `null` because `creator` threw an exception. Thus, the line labeled with `A` is unreachable in this copy of the `finally` block, but, as it might be reachable in another copy of the `finally` block, reporting it may introduce false warnings.

The only way to avoid false warnings in the presence of `finally` blocks is to identify copied statements. Unfortunately, this information is not available in the bytecode. Currently, we use a heuristic that works well in practice but is no general solution. First, we use the option of Soot to preserve the original Java source code line number for each statement in the Jimple program. Note that one cannot look for cloned statements, because the bytecode or the Jimple code may introduce or rename local variables, so the statements in the different clones are not exactly the same.

In the Jimple program with original line numbers, we iterate over a procedure body and check if we find the same line number multiple, but non-consecutive times. It is important to consider

```
void foo() {
  C obj = null;
  try {
    obj = creator();
  } catch (MyException e) {
  } finally {
    if (obj!=null) {
A:      //unreachable
    }
  }
```

Figure 2. Code snippet with `finally` statement, which causes a false warning.

only non-consecutive repetitions, as consecutive repetitions can be introduced, e.g., by multiple initialization. This solution is not robust against hostile users. If, e.g., the user writes everything into one line, this approach obviously fails. However, we assume that Java programmers usually follow established code conventions at least to some extent.

Once we have identified all non-consecutive repetitions we have several options: the precise way would be to eliminate all but one clone, introduce a helper variable that is a assigned to a unique value at each location that can jump into the finally block, and add conditional choice over this variable with the corresponding back-jumps to the end of the finally block. Currently, we use a cheaper workaround: we keep record of all cloned statements, and only if all clones of one statement are infeasible, we report it to the user.

For a proper solution to this problem we would need to carry the information about which statements have been cloned from the Java code through the bytecode to the Jimple code.

Sources of Unsoundness Beyond the above mentioned soundness issues which arise from the difference between Java code and bytecode, our translation has other sources of unsoundness which we introduced on purpose for more efficient infeasible code detection. As mentioned before, we do not model runtime exceptions. That is, code that is only reachable if such an exception is thrown will become unreachable. This did not happen in our experiments. Further, we do not model aliasing. For a given code fragment (in our analysis we only consider procedure bodies), if a piece of code is only reachable if there is an alias between variables in the initial state, we will wrongly report it as unreachable. This did not happen either in the experiments.

7. Evaluation

We evaluate our translation and the issues of unsoundness discussed in the previous section on five benchmark programs: the CASE tool ArgoUML, the mind-mapping tool FreeMind, the time tracker Rachota, a tiny word processor called TerpWord, and our tool, Joogie, that does the translation. The benchmark programs are picked without any deep consideration of statistic validity, but due to their size, we believe that they give fairly realistic results. Joogie, and all AUTs are available online[1].

The goal of this evaluation is to measure how many instances of infeasible code can be found and how many false positives are being emitted. Note that all the AUTs are stable releases, so it is unlikely to find much infeasible code, as the applications hopefully have been tested before.

Table 1 shows that, with our translation, infeasible code detection takes in average less than a second per method, and we are able to find infeasible code in each of the analyzed programs. However, in all programs besides Rachota, we report false positives. Fortunately, all but one false positive arise from one of the two issues discussed above: code duplication of finally blocks or non-existing control-flow from conditional choices. Both cases can be handled by incorporating information about the source code into the Jimple translation. Only one false positive occurred for another reason in FreeMind, where a function calls `System.exit`, but our translation does not know that this call terminates the program.

In conclusion, we can say that our experiments show that a very simple translation is sufficient to detect infeasible code, and that most sources of unsoundness do not cause false warnings in practice. Further, the two main sources of false warnings can be eliminated with little engineering effort.

8. Related Work

Various approaches have been presented that use Jimple [21] as a basis for static analysis of Java (byte) code. For brevity, we refer to [14] for an overview. In this paper we discuss how we can use Jimple for *infeasible code* detection. Infeasible code detection algorithms have, for example, been presented in [1, 7, 8, 13, 20]. The approaches presented in [8] and [13] use a syntactic pattern matching directly on the source code of `C` resp. Java programs. In [7] and [20] algorithms to detect infeasible code in `C` programs are presented which are implemented in `clang` and `llvm`, however, these papers do not discuss the translation from source- to intermediate code in detail. Joogie [1] detects infeasible code in Jimple (or Java) code. This paper discusses the translation used in Joogie in detail.

There is a large body of research on **intermediate verification languages** (IVLs), and the encoding of real-world programming languages into IVLs. The two most widely used IVLs are Boogie [16] and Why [9]; for both languages mature implementations and frameworks are available to parse, type-check, and analyze programs. We concentrate on Boogie in this paper (an introduction is given in Section 3), but all results straightforwardly carry over to Why, or to other IVLs with comparable functionality.

Among others, Boogie has been used to represent programs in the following programming languages:

The design of Boogie was motivated by the requirements of the **Spec#** [2] language, which is a variant of C# (and a verification system) extended with verification-related features and code annotations. Our work is similar to Spec# in that we process programs on the level of bytecode (Jimple), not on the level of source code (Java). Spec# uses a highly sophisticated verification methodology tailored to functional verification, whereas our method targets detection of infeasible code, and is therefore more lightweight, automatic, and scalable.

A translation of a substantial part of the **Java bytecode** language to Boogie was presented and proven sound in [15]. The translation was designed for functional verification of Java programs, but uses a heap model similar to the one defined in Section 3.1. In our work, the use of the Jimple language enables a simpler translation than defined in [15], since only a smaller number of instructions, and no operand stack has to be considered.

There are multiple tools for analyzing **C programs** by means of encoding into Boogie, including Havoc [4] and VCC [5]. Both approaches use detailed and accurate models of the heap in C, to the extent necessary to analyze low-level code such as operating systems or drivers. Like Spec#, such tools are tailored to functional verification, not to the detection of infeasible code, and demand a substantial amount of user interaction.

Dafny [17] and **Chalice** [19] are academic languages that support verification by means of translation to Boogie. Dafny is object-oriented, and its heap model can be represented in Boogie in a similar manner as the model defined in Section 3.1. Chalice was designed to explore verification methodologies for concurrent computations. Both Dafny and Chalice are tailored to functional verification, and require annotation of programs by the user.

[1] `http://joogie.org`

Program	LOC	# checked methods	# found	#true pos	# false pos (if-then-else)	# false pos (finally)	Time (min)
ArgoUML	156,294	9,981	63	28	22	13	51
FreeMind	53,737	5,613	13	10	1	1	16
Joogie	11,401	973	3	0	1	2	8
Rachota	11,037	1,279	1	1	0	0	19
TerpWord	6,842	360	7	3	1	3	3

Table 1. Results of applying Joogie to the test applications.

9. Conclusion

We have presented a translation from Jimple into Boogie that is tailored for infeasible code detection. The translation is not sound for technical reasons, still, experimental results show that most of the sources of unsoundness do not cause false alarms in practice. However, the experiments also show that some sources of unsoundness cannot be ruled out without considering the original Java code. This happens every time a control location in the Java code is represented by multiple control locations in the bytecode which are not on the same path. E.g., a finally block that is copied to the end of the `try` block and the end of the `catch` block, or a conditional choice with a conjunction in the conditional, where `else` blocks exist in the bytecode which do not exist in the Java code. Suppressing these warnings by looking up in the corresponding Java code if the location is a potential false alarm is one option. However, for our future work we interested in finding a more general solution to this problem by pre-processing the Java program.

Beyond these limitations, we have presented a relatively simple translation that is yet precise enough to reveal infeasible code even on well tested programs. In the future, we plan to integrate Joogie into an IDE to evaluate how much more can be found if we can analyze code while it is written, and how disturbing the above mentioned false positives really are for the programmer.

Acknowledgements. This work is in part supported by the grants COLAB and JOOGIE of the Macao Science and Technology Development Fund, and by Vetenskapsrådet (VR).

References

[1] S. Arlt and M. Schäf. Joogie: Infeasible code detection for java. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 767–773. Springer, 2012. ISBN 978-3-642-31423-0.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, pages 364–387. Springer, 2006.

[3] R. Bornat. Proving Pointer Programs in Hoare Logic. In *Proceedings of MPC 2000*, pages 102–126, London, UK, 2000. Springer. ISBN 3-540-67727-5. URL `http://dl.acm.org/citation.cfm?id=648085.747307`.

[4] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS 2007*, pages 19–33, 2007.

[5] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009. ISBN 978-3-642-03358-2.

[6] E. W. Dijkstra. *A discipline of programming / Edsger W. Dijkstra*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[7] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 435–445, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250784. URL `http://doi.acm.org/10.1145/1250734.1250784`.

[8] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502041. URL `http://doi.acm.org/10.1145/502034.502041`.

[9] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, pages 173–177, 2007.

[10] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 193–205, New York, NY, USA, 2001. ACM. ISBN 1-58113-336-7. doi: 10.1145/360204.360220. URL `http://doi.acm.org/10.1145/360204.360220`.

[11] J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. It's doomed; we can prove it. In *FM'09*, pages 338–353. Springer, 2009.

[12] J. Hoenicke, K. R. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *Formal Methods in System Design*, 2010.

[13] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to OOPSLA 2004*, pages 132–136, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: http://doi.acm.org/10.1145/1028664.1028717. URL `http://doi.acm.org/10.1145/1028664.1028717`.

[14] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[15] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. *Electronic Notes in Theoretical Computer Science*, 190(1):35–50, 2007.

[16] K. R. M. Leino. This is Boogie 2, 2008. Manuscript KRML 178.

[17] K. R. M. Leino. Specification and verification of object-oriented software. In *Summer School Marktoberdorf 2008*, NATO ASI Series F. IOS Press, 2009.

[18] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

[19] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

[20] A. Tomb and C. Flanagan. Detecting inconsistencies via universal reachability analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 287–297, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1454-1. doi: 10.1145/2338965.2336788. URL `http://doi.acm.org/10.1145/2338965.2336788`.

[21] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

[22] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999. URL `www.sable.mcgill.ca/publications`.