Joogie: Infeasible Code Detection for Java

Stephan Arlt[1] and Martin Schäf[2]

[1] Albert-Ludwigs-Universität Freiburg
[2] United Nations University, IIST, Macau

Abstract. We present Joogie, a tool that detects infeasible code in Java programs. Infeasible code is code that does not occur on feasible control-flow paths and thus has no feasible execution. Infeasible code comprises many errors detected by static analysis in modern IDEs such as guaranteed null-pointer dereference or unreachable code. Unlike existing techniques, Joogie identifies infeasible code by proving that a particular statement cannot occur on a terminating execution using techniques from static verification. Thus, Joogie is able to detect infeasible code which is overlooked by existing tools. Joogie works fully automatically, it does not require user-provided specifications and (almost) never produces false warnings.

1 Introduction

We present Joogie, a static analysis tool to detect infeasible code in Java programs. Infeasible code is code which does not occur on any feasible control-flow paths and hence has no feasible execution. That is, infeasible code is either not forward-reachable or not backward-reachable on a feasible execution. Common examples of infeasible code are unreachable code, or guaranteed null-pointer dereference.

Infeasible code tends to occur in a very early stage of development and should be found at the latest during testing. An intrinsic property of infeasible code is that it has *no* feasible execution. That is, a code fragment can be detected to be infeasible without knowing its full context. Extending its context can only restrict its feasible executions and thus an infeasible code fragment will remain infeasible in any larger context. Hence, infeasible code lends itself to be detected by static analysis: it can be detected for code fragments in isolation using relatively coarse abstractions of the feasible executions, and with a very low rate of false warnings.

Infeasible code can, e.g., be detected using data-flow analysis tools such as Findbugs [8] or the built-in static analysis of Eclipse which, among other things, also detects infeasible code. We claim that, among all static analysis tools, those detecting infeasible code are some of the most widely used. Programmers do not suppress Eclipse-warning that an object is always null when dereferenced or that a particular code fragment is unreachable. That is, improving infeasible code detection can have a large impact in practice.

In contrast to existing tools that detect infeasible code, Joogie uses techniques from static verification to prove the presence of infeasible code. This results in a higher precision than pure syntactic analysis. Joogie first translates a given program into the Boogie language [10] as described in Section 3. Then, a modified version of the Boogie program verification system [1] is used to prove the presence of infeasible code as described in Sect. 4. We show the ability of Joogie to detect infeasible code which is not found using existing tools by applying our tool to three real world applications in Sect. 5. Joogie works fully automatically, does not require any user interaction, and is able to detect real errors while almost never producing false warnings.

2 Joogie Overview

Figure 1 gives an overview of Joogie. Joogie takes a Java program as input. Joogie splits the task of proving the presence of infeasible code in two steps. In a first step, the Java program is translated into Boogie. During this translation, the type system and memory model are replaced by more abstract concepts which facilitate the use of existing verification techniques. The details of this translation are described in Sect. 3. Note that this translation is neither sound nor complete. That is, some feasible executions might be lost which can result in false warnings, and the translation may add feasible executions which can result in false negatives.

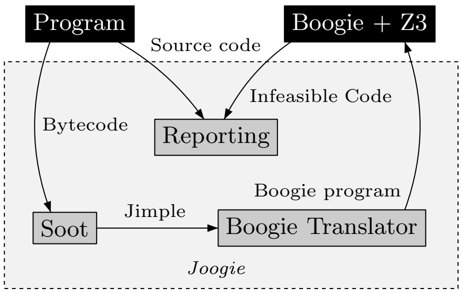

Fig. 1. Overview of Joogie

In a second step, Joogie calls a modified version of the Boogie program verifier to prove the presence of infeasible code in the Boogie program. The underlying decision procedure is based on the weakest liberal precondition calculus and uses a sound abstraction of the given Boogie program. Section 4 gives more details on the used algorithms. For each infeasible statement in the Boogie program, Joogie reconstructs the corresponding statement in the Java source code and returns an error message. Joogie works fully automatically. Joogie does not require specification statements, but in general it is possible to further annotate the generated Boogie program to increase the detection rate or check for additional properties.

3 Bytecode Translation

Joogie translates Java to Boogie using the Java optimization framework Soot [11]. Soot translates the Java program into a 3-address intermediate representation of the program's bytecode, which significantly simplifies the translation to Boogie, as only 15 different kinds of statements have to be considered.

One of the most vital parts of translating an object-oriented language into an intermediate verification language is the used memory model. For a sound infeasible code detection it is sufficient to preserve all feasible executions of the original program (contrary to partial correctness proofs, where all infeasible executions have to preserved). Thus, Joogie can use a simple Burstall-Bornat-style heap-as-array model (see e.g., [4, 10]). The heap is represented by a two-dimensional array, where the first index refers to the address of an object in the heap and the second index refers to the field that is to be accessed. Soot ensures that references to objects are *null* by default. Assertions to guard the heap access are introduced automatically by Joogie. For brevity of exposure, we do not explain this model in detail. Similar approaches can be found, e.g., in Spec# [2] or ESC/Java [5]. Note that using assertions is not sound, as the Java program would throw an exception rather than terminate when the exception is violated.

Integers, Chars, and Bytes are represented using the Boogie built-in type for unbounded natural numbers. Using an unbounded representation for bounded variables is an unsound abstraction. Hence, Joogie uses uninterpreted functions for arithmetic operators, which can be redefined using axioms if a sound handling of primitive types is needed. However, unless the programmer deliberately makes use of Java's overflow and underflow handling, this is a feasible abstraction and, so far, we did not encounter false warnings resulting from this unsoundness. String variables are treated like any other object. Doubles and floats are treated in a similar way as objects. They are represented as arbitrary values and operators on them are represented as uninterpreted functions. This abstraction is coarse and certainly leaves room for improvements, but it is sound and efficient for our purpose of detecting infeasible code. Arrays are represented as one-dimensional unbounded arrays of an appropriate type. The size of an array is stored outside the bounds of the original array. Array-bounds checks are modeled using assertion statements, which is unsound for the general case, as out-of-bounds exceptions might be handled in the code. However, this can be changed easily depending on the user's preferences.

Exceptions are modeled as multiple return parameters of a method. If an exception is thrown, the corresponding return parameter is assigned to the instance of the exception, and the method returns, or, if possible, jumps to an adequate catch block. After each method call, conditional choices are added to redirect the control-flow if an exception has been thrown by the called method.

In general, this translation is not sound as it does not consider aliasing of method parameters and global variables. This unsoundness could be eliminated by, e.g., modeling the aliasing explicitly which would increase the complexity of the translated program significantly. However, our experiments show that this simplification does not introduce false warnings.

4 Infeasible Code Detection

We check for the existence of infeasible code in the Boogie program using the algorithms described in [7] and [3]. These algorithms are implemented as an extension to the Boogie program verification system. For each control location in a program P, we introduce a statement assigning an auxiliary *reachability variable* r_i to the constant 1, where i ranges over the number of all program statements. This allows us to check the existence of an execution that passes this location, by checking if any terminating execution starting in an initial state where $r_i = 0$ terminates in a state where it is still 0. If this is the case, then no terminating execution passes the assignment $r_i := 1$ and hence no execution passes the considered statement. This check is automated by augmenting the program P with reachability variables, computing a formula representation of the weakest-liberal precondition of this program, and then using a SMT solver (here: Z3) that checks if $(r_i = 0) \models wlp(P, r_i = 0)$ holds (a similar concept is used in [6]).

To compute a formula representation of *wlp*, we first eliminate the loops in our program P using the abstract loop unwinding from [7]. A loop is replaced by three unwindings. The first and the last unwinding represent the first and the last iteration of the loop, respectively. To every entrance and exit of the middle unwinding, we add non-deterministic assignments to all variables modified inside the loop body. This *abstract unwinding* represents all other unwindings. Note that, for copied locations, we do not create fresh r_i variables, and thus, the abstraction does not remove feasible executions from the program (proof in [7]).

Joogie does not do any inter-procedural analysis. Any procedure call is replaced by a non-deterministic assignment to all variables that might be modified by this procedure. Still, this is a sound abstraction.

For the resulting loop-free program, we compute a formula representation of the weakest-liberal precondition using standard techniques which are already provided by Boogie. The algorithm to detect infeasible code in Boogie programs is sound w.r.t. infeasible code detection under two preconditions: procedure parameters do not alias, and the program is single-threaded. The first one can be lifted by adding switch cases. For multithreading, we do not have a sound solution yet. If a statement is only executed on interleaved executions, it will be reported as infeasible. That is, in general Joogie is not sound. We evaluate its feasibility in the experiments in the next section.

5 Experiments

Joogie, all experimental data, and additional results can be found on the website[1]. We apply Joogie on 3 real-world Java applications, TerpWord 4.0, Rachota 2.4, and FreeMind 0.9, to check the performance of Joogie, whether it can find infeasible code, and whether it does produce false warnings . We also apply Joogie on Joogie itself. All experiments are executed several times on a standard notebook (Dual Core 1.6 GHz, 2 GB RAM, 5400 rpm HDD). Note that infeasible code should be detected at the latest during testing, and it should not occur in any stable release of a program. That is, we expect to find hardly any or even no infeasible code. For a detailed evaluation including reports on detection rate, experiments with seeded infeasible code are needed. Table 1 shows the summary

[1] `http://code.google.com/p/joogie/`

of our experiments, and Figure 2 gives a more detailed view on the computation time per method.

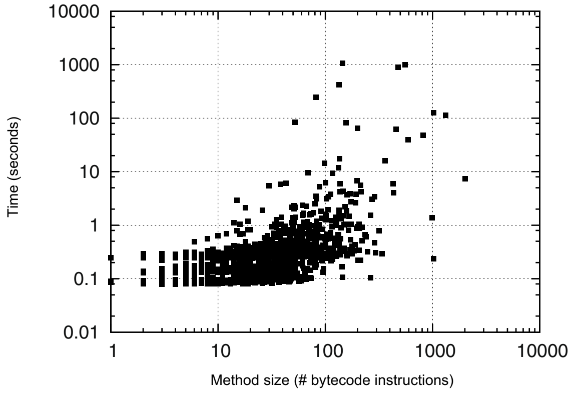

Fig. 2. Computation time of Joogie per Java method

Table 1. Results of applying Joogie to the test applications

Program	LOC	# checked methods	# found bugs	# false warnings	Time (min)
TerpWord	6842	965	4	2	2.95
Rachota	13750	1835	1	0	49.13
FreeMind	40922	8008	12	1	64.41
Joogie	5433	781	0	0	1.37

Observations. Joogie is able to detect infeasible code in the stable releases of 3 applications. Some of it is simple unreachable code, some of it is code that will cause a run time error when reached. Examples of detected infeasible code are given on the Joogie website. We did encounter two false warnings in TerpWord: one is due to a bug when parsing the Java program, the other one is a statement that is only reachable due to interleaving. Joogie does not deal with interleaving. The other sources of unsoundness of the translation from Java to Boogie wrt. infeasible code detection did not cause any false warnings. In Rachota we found one bug. In FreeMind, we found 12 bugs but also 1 false positive due to bugs in Joogie which we could not fix until the deadline.

Figure 2 shows, the average computation time per method is way below one second for most methods. As Joogie is meant to be used incrementally on recently modified program fragments similar to, e.g., the static analysis in Eclipse, the computation time can be tolerated. Larger or more complex methods can be split in smaller parts which are analyzed in isolation.

6 Conclusion

Joogie is useful: it does not require any user interaction, it is fully automatic, it detects errors, and it does almost never produce false warning. The experiments show that Joogie can be applied to real programs and that it does find infeasible code, even in sufficiently tested code. Our long term goal is to make Joogie efficient enough to run in the background while the programmer is typing. Until then, there is still much room for improvements. The complexity of the generated Boogie program can be further optimized by sharing variables between independent program fragments, techniques from verification could be used to infer invariants, or more efficient ways to represent the heap could be applied.

By using Boogie as an intermediate representation, Joogie can be easily extended by other researchers. E.g., the translation from Java to Boogie could be modified to identify different classes of errors, or specification statements could be added to further increase the detection rate.

We observe that it is not always trivial to understand why code is infeasible. In contrast to, e.g., run-time errors, where a trace counterexample is sufficient to explain why the error occurs, infeasible code can be witnessed by this way. In our future work we will explore techniques like e.g., BugAssist [9] that can be used to explain infeasible control-flow.

Acknowledgements. This work is supported by the projects ARV and COLAB funded by Macau Science and Technology Development Fund.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# Programming System: Challenges and Directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
3. Bertolini, C., Schäf, M., Schweitzer, P.: Infeasible Code Detection. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 310–325. Springer, Heidelberg (2012)
4. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
5. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. SIGPLAN Not. 37, 234–245 (2002)
6. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
7. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: Doomed program points. Form. Methods Syst. Des. 37, 171–199 (2010)
8. Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: Companion to OOPSLA 2004, pp. 132–136. ACM, New York (2004)

9. Jose, M., Majumdar, R.: Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 504–509. Springer, Heidelberg (2011)
10. Leino, K.R.M., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
11. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999)