JayHorn: A Framework for Verifying Java programs

Temesghen Kahsai[1], Philipp Rümmer[2], Huascar Sanchez[3], and Martin Schäf[3](✉)

[1] Nasa Ames / CMU, Moffett Field, USA
[2] Uppsala University, Uppsala, Sweden
[3] SRI International, Menlo Park, USA
martin.schaef@sri.com

Abstract. Building a competitive program verifiers is becoming cheaper. On the front-end side, openly available compiler infrastructure and optimization frameworks take care of hairy problems such as alias analysis, and break down the subtleties of modern languages into a handful of simple instructions that need to be handled. On the back-end side, theorem provers start providing full-fledged model checking algorithms, such as PDR, that take care looping control-flow.

In this spirit, we developed JayHorn, a verification framework for Java with the goal of having as few moving parts as possible. Most steps of the translation from Java into logic are implemented as bytecode transformations, with the implication that their soundness can be tested easily. From the transformed bytecode, we generate a set of constrained Horn clauses that are verified using state-of-the-art Horn solvers. We report on our implementation experience and evaluate JayHorn on benchmarks.

1 Introduction

Building a software model checking tool has always been a strenuous endeavor. Established tools, such as CBMC [11] or Java Pathfinder [9] have amassed countless man-hours of engineering and testing. However, over the recent years, this task has become a lot simpler with the increasing availability of off-the-shelf front-ends such as LLVM [13], Wala [1], or Soot [20], and verification back-ends such as Z3 [5], Corral [12], or Eldarica [18].

With the increasing availability of such tools, the task of building a software model checker becomes just a matter of picking a front-end and a back-end and writing the glue code to connect them. Recent verification competitions have shown that this approach is feasible in practice. Tools like SMACK [17] or SeaHorn [8] which use LLVM as a front-end and off-the-shelf verification back-ends have been able to outperform established tools in many categories.

Motivated by these developments, we have implemented JayHorn, a software model checking tool for Java. JayHorn uses the Java optimization framework Soot as a front-end, generates a set of constrained Horn clauses (CHCs) to encode the verification condition. The Horn clauses are then sent to a Horn engine. For the construction of JayHorn we made the following design decisions:

© Springer International Publishing Switzerland 2016
S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part I, LNCS 9779, pp. 352–358, 2016.
DOI: 10.1007/978-3-319-41528-4_19

1. Perform as much of the translation work as possible in Soot: Translation of exception handling, de-virtualization, and control-flow simplification are implemented as bytecode transformation. These steps do not alter a programs behavior which allows us to use Randoop [16] to test their correctness.
2. Keep the glue code that generates verification conditions small, modular, and extensible: by performing many steps as bytecode transformation, the step of generating verification conditions becomes relatively simple and is implemented in a few hundred lines of Java code. The implementation is modular and extensible to allow, for example, different encoding of memory or numeric types.
3. Keep the back-end exchangeable: Horn solvers are constantly improving and new tools are being released frequently. To ensure that JayHorn builds on the most efficient back-end, we made it modular and replaceable. Currently, JayHorn supports Z3 and Eldarica as back-ends but can be easily extended to support other tools.

Roadmap. In Sect. 2 we discuss the architecture of JayHorn in more detail and address issues such as soundness and limitations. In Sect. 3 we evaluate JayHorn on a set of benchmarks. Then we conclude and discuss our next steps.

2 Architecture of JayHorn

In the following, we discuss architecture, soundness, and limitations of JayHorn. The big picture is outlined in Fig. 1. In it's default configuration, JayHorn takes Java bytecode as input and checks if any Java `assert` can be violated.

JayHorn accepts any input that is accepted by Soot. That is, Java class files, Jar archives, or Android apk's. For code that is not annotated with `assert`

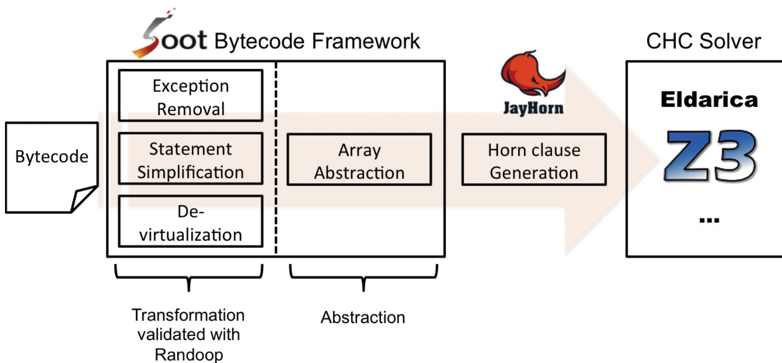

Fig. 1. Architectural overview of JayHorn. The tool takes Java bytecode as input. It then uses Soot to perform a set of transformations that do not alter the input/output behavior of the program, followed by an abstraction step to simplify arrays. The transformed bytecode is passed to JayHorn's glue code that constructs a system of CHCs which are passed into a Horn engine to check the safety of the input program.

statements, JayHorn also provides an option to guard possible NullPointer-Exceptions, ArrayIndexOutOfBoundsExceptions, and ClassCastExceptions with assertions (note that this affects soundness because developers may catch these exceptions on purpose even though it is not good practice).

Soot converts the given input into the Jimple intermediate format, which is a three-address code version of the Java bytecode. The benefit of operating on Jimple is that we only have to handle 15 different operations instead of over 200 as in the case of raw bytecode.

Program Transformation. We use the Soot infrastructure to apply (and implement) a set of bytecode transformations to simplify the input program. First, we eliminate exception handling and make all implicit exceptional control-flow explicit. To that end, we introduce a global variable (i.e., a public class with one public static field) to hold the last thrown exception. A thrown exception is then transformed into an assignment to this variable followed by a return (for primitive types we return a minimal values, for all other types we return `null`). After each method call, we first check if this exception variable has been set and, if so, ignore the return value and move control to the exceptional successor of the call statement. At the end of each entry point (e.g., `main`), we check if the exception variable has been set and throw the exception if necessary. This transformation does not alter the input/output behavior of the transformed program.

In the next step, we simplify the input program further by replacing `switch` statements by `if` statements and by removing unreachable code. In this step, we also add a public field `dyntype` to each class that carries the dynamic type of an object and set this field every time an object is created with `new`. For example, a Jimple statement `A a :=new B();` would be followed an assignment `a.dyntype = B.getClass();`. Like the previous step, this step does not change the behavior of the input program.

Then, we de-virtualize the input program. That is, any call to a virtual method is replaced by a distinction of cases over the `dyntype` of the base of that call and calls to the corresponding methods. The de-virtualize can significantly increase the size of a program but Soot's built-in alias analysis' (and de-virtualization) can be used to reduce the number of cases that need to be distinguished.

Note that, up to this point, JayHorn has significantly simplified the program by removing exceptional flow, virtual method calls, and some statements that are syntactic sugar, without altering the behavior of the input program. Hence, we can test the correctness (or soundness) of these steps by comparing input/output behavior of the original and transformed code. Since this step is crucial for the soundness of the overall system, we employ Randoop [16] to automate this test. We also allow the user to generate these tests for a given input program to increase confidence.

Array Abstraction. On the simplified input program, JayHorn performs one abstraction step to eliminate arrays. Like the previous steps, this step is implemented as a bytecode transformation in Soot and can easily be replaced or modified if requirements change. Arrays in Java are objects, however, there are

a few subtleties that makes it harder to handle them. For example, access to the `length` field of an array is not a regular field access but a special bytecode instruction. To simplify the generation of CHCs in the next step we transform arrays into real objects. To that end, we generate a new class for each array type used in the input program. The class extends Object and has a public field `length` and a field `elType` containing the type of the array elements. For each element of the array, we generate a private field of appropriate type. For reading and writing, the array class provides a `get` and `put` method with a distinction of cases over the used index. We can bound the number of fields that are generated for the array. If the used index exceeds this number, we return a dedicated constant that is later translated into an uninterpreted symbol. This step allows us to treat arrays like any other object, however, if we bound the number of fields, it introduces abstraction. Since this step is still a bytecode transformation we can again use Randoop to check how it affects our precision.

Generating Horn Clauses. The transformed program only contains primitive types and Objects, and a small set of statements, which simplifies the translation to CHCs; the entire translation takes less than 600 lines of Java code which keeps the risk of introducing bugs and thus unsoundess low. Our encoding as CHCs is inspired by the concept of refinement types [6,21], and uses uninterpreted predicates to represent:

– for each method `m`, pre-conditions `pre_m` and post-conditions `post_m` talking about parameters and result;
– for each control location `l`, invariants `loc_l` talking about local variables that are in scope;
– for each class `C`, instance invariants `inv_C` talking about the dynamic type and fields of objects.

The translation of programs to clauses then proceeds method by method, mapping each instruction to one clause. Accesses to heap and object fields are replaced by `unpack` instructions that apply invariants `inv_C` to determine the possible values of fields, and `pack` instructions that assert instance invariants after write accesses. Method calls are mapped to clauses that assert the precondition `pre_m` of the called method, and clauses that exploit the method postcondition `post_m` to determine possible results and effects of the call.

Soundness and Completeness. JayHorn is implemented in the spirit of soundiness [14]. Our analysis does not have a fully sound handling of the following features: JNI, implicit method invocations, integer overflow, reflection API, `invokedynamic`, code generation at runtime, dynamic loading, different class loaders, and key native methods We have determined that the unsoundness in our handling of these features has no effect the validity of our experimental evaluation. To the best of our knowledge, our analysis has a sound handling of all language features other than those listed above.[1]

[1] Generated using the Soundiness Statement Generator from http://soundiness.org.

Further, JayHorn over-approximates variables of type `double`, `float`, and `long` and may over-approximate array usage (as discussed above). Other than that, completeness mostly depends on the selected back-end.

3 Evaluation

We demonstrate the current capabilities and limitations of JayHorn on a set of examples. Table 1 show the results of our evaluation. We JayHorn with Eldarica and Z3 as a back-end to three sets of single threaded benchmark problems: CBMC-java tests, which are simple examples involving a variety of Java constructs provided by the authors of CBMC; MinePump, a product line provided by the authors of CPAChecker, and a Java version of the recursive benchmarks from SVCOMP 2015. For each benchmark problem we record if a tool produces the correct answer (✓), the wrong answer (✗), or times out (TO).

Table 1. Evaluation of JayHorn on three sets of benchmarks with Z3 and Eldarica back-end. For each benchmark problem we record how often JayHorn is able to find the correct answer (✓), how often it fails to prove a correct program (✗) and how often the execution times out after 60 s (TO).

Benchmark	# Problems	JayHorn + Z3			JayHorn + Eldarica		
		✓	✗	TO	✓	✗	TO
CBMC-Java tests	44	26	12	6	33	8	3
MinePump	64	36	0	28	39	25	0
SVCOMP Recursive	23	7	2	14	14	2	7

The experiments show that JayHorn is currently able to handle a range of examples even though some language features are not fully implemented. The cases where JayHorn fails to produce the expected result (✗) are all cases where an assertion holds but the tool is not able to show it (i.e., JayHorn is sound on these benchmarks). The wrong results are caused by our current encoding of integers as mathematical integers, the abstraction of float and double numbers are arbitrary values, and the missing implementation of bit operations. The experiments also show the importance of a modular translation: JayHorn performs significantly better with Eldarica than Z3 which can largely be attributed to our encoding of programs in CHC. A different encoding might lead to the opposite result.

We tried to compare JayHorn with Java Pathfinder (JPF) as a baseline but due to different approaches, no comparison was possible. On the MinePump benchmarks, JPF is significantly faster than JayHorn. On the other two benchmarks, however, JPF failed since those initialize variables with random number and as JPF is an explicit state model checker, it fails to enumerate those states.

4 Related Work

Fully automated program analysis is a very active research area with many high quality tools (e.g., [2,4,8–11,15,17,19]). The work on JayHorn is primarily inspired by the success of SMACK [17] and SeaHorn [8] in the previous verification competitions.

Currently, not many tools of this kind are available for comparison that target Java. AProVE [7] and Ultimate [10] competed on termination analysis for Java programs. During the writing of this paper, the authors of Ultimate, CBMC, and CPAChecker [3] were actively working on tools for Java (and we would like to thank them for their test cases and examples) but none of them was available for a direct comparison with JayHorn. We will make JayHorn and our test cases and benchmarks publicly available to contribute to this development with the hope of having a verification competition for Java in the near future.

5 Conclusion

We have presented a new software verification framework for Java in the spirit of SeaHorn and Smack. JayHorn is continuously evolving. It's current status does not yet have any of the amenities needed in industrial practice, such as built-in specification for common libraries. However, it performs well on a set of well known verification problems and we are eager to compete with other tools. In the future, we plan to extend JayHorn to also handle termination.

The main contribution of JayHorn is its modular design. The front-end phase of JayHorn significantly simplifies the input program without changing its behavior which makes it easy to develop new analysis tools on top of this front-end. Further, JayHorn can connect to different tools that accept Horn clauses as input which can serve as an interesting benchmark.

Acknowledgement. This work is funded in parts by AFRL contract No. FA8750-15-C-0010, NSF Award No. 1422705, and the Swedish Research Council.

References

1. T.J. Watson library for analysis (wala). http://wala.sf.net
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: applications to software engineering. Int. J. Softw. Tools Technol. Transf. **9**(5), 505–525 (2007)
3. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
4. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded Ansi-C software. In: ASE, Washington, DC, USA, pp. 137–148. IEEE Computer Society (2009)

5. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI 1991, New York, NY, USA, pp. 268–277. ACM (1991)
7. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS, vol. 8562, pp. 184–191. Springer, Heidelberg (2014)
8. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Heidelberg (2015)
9. Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. Int. J. Softw. Tools Technol. Transf. **2**(4), 366–381 (2000)
10. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate Automizer with array interpolation. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 454–456. Springer, Berlin (2015)
11. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
12. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
13. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: CGO, Washington, DC, USA, p. 75. IEEE Computer Society (2004)
14. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.-Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: aamanifesto. Commun. ACM **58**(2), 44–46 (2015)
15. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 458–460. Springer, Heidelberg (2015)
16. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: OOPSLA, New York, NY, USA, pp. 815–816. ACM (2007)
17. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Heidelberg (2014)
18. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 347–363. Springer, Heidelberg (2013)
19. Spoto, F.: The nullness analyser of Julia. In: LPAR, pp. 405–424 (2010)
20. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: CASCON (1999)
21. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 209–228. Springer, Heidelberg (2013)