JayHorn: A Java Model Checker
(Competition Contribution)

Temesghen Kahsai[1], Philipp Rümmer[2], and Martin Schäf[3]

[1] The University of Iowa
[2] Uppsala University
[3] SRI International

Abstract. JayHorn is a model checker for verifying sequential Java programs annotated with assertions expressing safety conditions. JayHorn uses the Soot library to read Java bytecode and translate it to the Jimple three-address format, then converts the Jimple code in several stages to a set of constrained Horn clauses, and solves the Horn clauses using solvers like SPACER and Eldarica. JayHorn uses a novel, invariant-based representation of heap data-structures, and is therefore particularly useful for analyzing programs with unbounded data-structures and unbounded run-time. JayHorn is open source and distributed under MIT license.[4]

1 The JayHorn Approach

JayHorn is a model checker for verifying the absence of assertion violations in sequential Java programs by automatically inferring program annotations that are sufficient to witness program safety. Annotations are quantifier-free formulas in first-order logic modulo relevant theories like LIA. The choice of annotations is inspired by refinement types [1] and liquid types [7], and consists of:

- for each method `m`, a pre-condition `pre_m` defining conditions under which the method can be invoked, and a post-condition `post_m` stating the effect of the method in terms of the method parameters, the method result, possible exceptions, and certain ghost variables encoding the state of the heap;
- for each control location `l`, a state invariant `loc_l` describing the possible values of local variables that are in scope;
- for each class `C`, an instance invariant `inv_C` describing possible values of object fields and the dynamic object type.

The sufficiency of annotations is characterized by a set of constrained Horn constraints expressing that state invariants in a method body are ensured by the method pre-condition and preserved by all statements in the method body, that methods establish their post-conditions, and that updating the fields of an object preserves the instance invariant `inv_C`; more details are provided in [4]. Given the complete set of conditions on the annotations, the actual annotation inference can be carried out with the help of off-the-shelf Horn solvers, like SPACER [5], which uses a variant of PDR/IC3, and Eldarica [3], which uses CEGAR.

[4] https://github.com/jayhorn/jayhorn

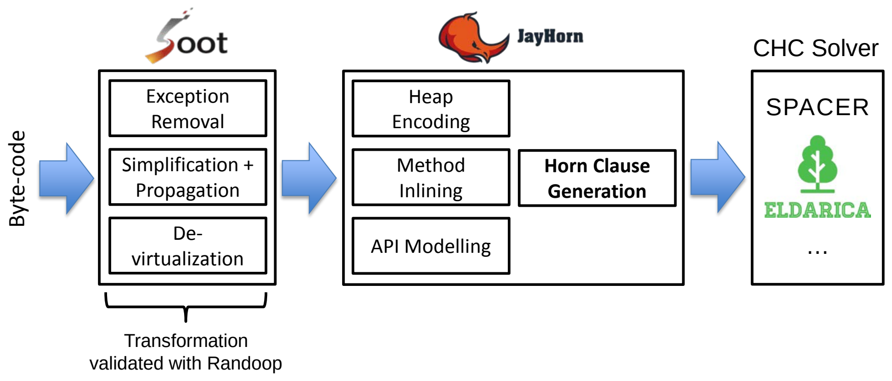

Fig. 1. Architectural overview of JayHorn.

The representation of heap data-structures using instance invariants in general over-approximates the program behavior, since instance invariants have to hold for the possible states of *all* objects of some class (as well as all elements of encoded arrays), at *any* point during program execution, and they cannot refer to local variables or to fields of other objects. The encoding (and JayHorn) is therefore incomplete, and it is easy to construct correct Java programs that cannot be verified using any choice of annotations [4]. To prevent incorrect answers, JayHorn applies a counterexample validation step whenever the generated Horn clauses are found to be unsatisfiable (see Section 2).

2 Architecture of JayHorn (Fig. 1)

Program transformations: In its default configuration, JayHorn takes Java bytecode as input and checks if Java `assert` can be violated. JayHorn accepts any input that is supported by the Soot framework [8]: Java class files, Jar archives, or Android apk. For code that is not annotated with `assert` statements, JayHorn also provides an option to guard possible `NullPointerExceptions`, `ArrayIndexOutOfBoundsExceptions`, and `ClassCastExceptions` with assertions.

Soot is used to translate Java bytecode to the simplified Jimple three-address format, followed by a set of transformations to further simplify a program, among them elimination of exception handling and implicit exceptional control-flow; replacement of `switch` statements by `if` statements; and de-virtualization of method calls in the input program. 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 [6] to automate this test.

On the simplified input program, JayHorn performs one abstraction step to eliminate arrays, again implemented as a bytecode transformation in Soot. Arrays in Java are objects, so 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 later generation of Horn clauses, we transform arrays into real objects, and introduce a `get` and `put` method to access the array elements.

The next step is the replacement of all heap accesses with `push`/`pull` instructions that copy all fields of an object in a single step to/from local variables [4], preparing the ground for the later representation of heap using invariants. The placement of `push`/`pull` is optimized to use as few statements as possible, this way reducing the size of generated constraints, and minimizing the effect of later over-approximations.

Horn clause generation: The transformed simplified Java program is then encoded as a set of constrained Horn clauses, using uninterpreted predicates to represent the annotations from Section 1. The encoding is mostly standard, and follows the rules given in [2]. The `push`/`pull` instructions are replaced with assertions and assumptions of the corresponding instance invariant `inv_C` [4].

In order to mitigate incompleteness due to the instance invariants, JayHorn implements number of refinements of the basic encoding, extending the set of programs that can be captured using instance invariants. *Flow-sensitive* instance invariants rely on a separate static analysis to determine which `push`es a `pull` instruction can read from, and can this way distinguish different object states. *Vector references* enrich references with additional information about an object, for instance the dynamic type, the allocation site, or values of immutable fields.

Counterexample validation: Since the encoding of programs using instance invariants over-approximates program behavior, there is a possibility of spurious assertion violations. JayHorn therefore implements a separate counterexample validation step with a precise, but bounded representation of heap (i.e., an under-approximate program encoding). This step is applied when the encoding with instance invariants leads to an inconclusive result. If neither over-approximate nor under-approximate encoding are able to infer a conclusive result, JayHorn reports `UNKNOWN` as overall verification result.

3 Weaknesses and Strengths

Weaknesses: The development of JayHorn is ongoing, and at this point several key Java features are not fully supported yet, including (i) strings; (ii) enums; (iii) bounded integer data-types; (iv) floating-point data-types; (v) reflection and dynamic loading; (vi) concurrency. The JayHorn model of the Java API is rudimentary, so that JayHorn assumes arbitrary behavior for most API functions. Some parts of JayHorn also need more optimization to reduce the run-time of the tool, in particular some of the program transformation steps. The Horn encoding could be optimized to use fewer relation symbols with smaller arity.

Strengths: Due to way heap is encoded, JayHorn is particularly suitable for the analysis of relatively shallow properties of programs with unbounded iteration, unbounded recursion, or unbounded heap data-structures; examples illustrating the capabilities of JayHorn are given in [4].

SV-COMP 2019: The mentioned features make JayHorn a relatively bad match for the Java benchmarks used in SV-COMP 2019, which are predominantly regression tests checking the correct handling of language features and of the Java string API. A large fraction of the benchmarks (the `MinePump` family) relies on correct handling of enums, and could therefore not be solved by JayHorn. Only a few of the SV-COMP benchmarks contain unboundedness in the form of loops, recursion, or heap data-structures.

JayHorn gave a wrong answer for two benchmarks in the competition. The program `UnsatAddition02` was incorrectly classified as correct (true), since JayHorn assumes unbounded integers. `synchronized` was incorrectly reported to be incorrect (false) due to an incomplete model of the `synchronized` construct, JayHorn does not support concurrency yet.

The results in the competition are overall promising, but do not represent a typical application scenario of JayHorn. The JayHorn team plans to address this for 2020 by submitting further benchmarks to SV-COMP, and by completing Java support of JayHorn, in particular fully supporting strings.

4 Download and Use of JayHorn

JayHorn is fully implemented in Java, and uses the libraries mentioned in Fig. 1. The version submitted to SV-COMP 2019 is JayHorn version 0.6.[5] In the configuration used in the competition,[6] JayHorn only applies the Horn solver Eldarica. Since Eldarica is itself implemented in Scala, this means that no native code was used in JayHorn in the competition. The Benchexec tool info module is called `jayhorn.py` and the benchmark definition file `jayhorn.xml`. JayHorn competes in the Java category.

To run JayHorn 0.6, it is enough to download the Jar file `jayhorn.jar` from the link below, and run it on bytecode:

```
wget https://raw.githubusercontent.com/jayhorn/jayhorn/devel/ \
  jayhorn/src/test/resources/horn-encoding/classics/UnsatMccarthy91.java
wget https://github.com/jayhorn/jayhorn/releases/download/v0.6/jayhorn.jar
mkdir tmp
javac UnsatMccarthy91.java -d tmp
java -Xss40m -Xmx3000m -jar jayhorn.jar -inline-size 10 -solution -j tmp
```

Acknowledgement. We are grateful for contributions to JayHorn by Daniel Dietsch, Rody Kersten, Huascar Sanchez, and Valentin Wüstholz. The development of JayHorn is funded in parts by AFRL contract No. FA8750-15-C-0010, NSF Award No. 1422705, by the Swedish Research Council (VR) under grants 2014-5484 and 2018-4727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

[5] https://github.com/jayhorn/jayhorn/releases/tag/v0.6

[6] Java options `-Xss40m -Xmx3000m`, JayHorn options `-inline-size 10`

References

1. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, New York, NY, USA, 1991. ACM.
2. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
3. H. Hojjat and P. Rümmer. The ELDARICA Horn solver. In N. Bjørner and A. Gurfinkel, editors, *FMCAD*, pages 1–7. IEEE, 2018.
4. T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR-21*, volume 46 of *EPiC*, pages 368–384, 2017.
5. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.
6. C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA*, pages 815–816, New York, NY, USA, 2007. ACM.
7. P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.
8. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON*, 1999.