Ultimate Kojak with Memory Safety Checks
(Competition Contribution)

Alexander Nutz*, Daniel Dietsch, Mostafa Mahmoud Mohamed, and Andreas Podelski

University of Freiburg
{nutz,dietsch,amin,podelski}@informatik.uni-freiburg.de

Abstract. Ultimate Kojak is a symbolic software model checker implemented in the Ultimate framework. It follows the CEGAR approach and uses Craig interpolants to refine an overapproximation of the program until it can either prove safety or has found a real counterexample.

This year's version features a new refinement algorithm, a precise treatment of heap memory, which allows us to deal with pointer aliasing and to participate in the memsafety category, and an improved interpolants generator.

1 Verification Approach

Ultimate Kojak starts verification by constructing a program graph for the input program. Nodes in the program graph are labelled with formulae that represent abstract program states, the edges are labelled with transition formulae. Procedure calls and returns are represented by special edges such that the program graph can be seen as a nested word automaton [4]. A failed emptiness check on this automaton yields an error path as a nested word. From the error path we build an SMT formula that is satisfiable if and only if the path is feasible. If the error path formula is satisfiable, we retrieve a model from the solver and translate the path together with the model back to an error witness. If the error path formula is unsatisfiable, we start our interpolant generator which uses an unsatisfiable core that the solver yields together with strongest post computation and live variable analysis to obtain a nested interpolant for the error path.

For refining the graph, we employ the Impulse (working title) algorithm [7]. In the first step of the refinement, we make a copy of each node on the error path, this copy's formula is conjoined with the interpolant formula we obtained for this position in the trace. We also make copies of the outgoing edges of each copied node. At first, we let them point to their original target. In the second step of the refinement, we attempt to redirect edges nodes with a stronger invariant formula such that, in the end, we may disconnect the initial location in the graph from the error location as soon as possible. These steps are depicted in Figure 1.

* Corresponding author.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 458–460, 2015.
DOI: 10.1007/978-3-662-46681-0_44

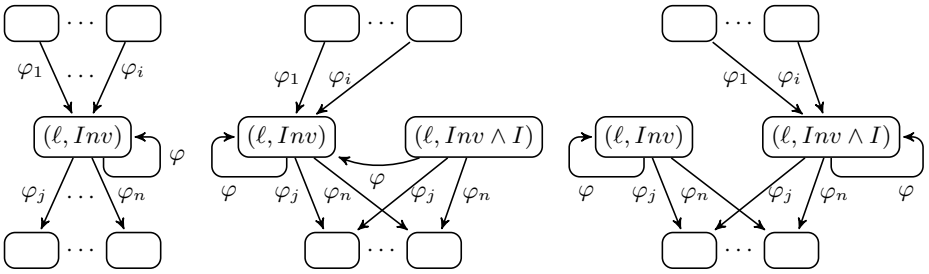

Fig. 1. The steps of the Impulse algorithm. Left: before refinement. Middle: after copying the node (ℓ, Inv). Right: after redirecting. Note that only edges corresponding to valid Hoare triples (in this picture: all) are redirected.

For dealing with memory safety properties, pointer aliasing and related problems, we make use of a simple but sufficient model of the heap used by C programs: A cell (byte) in the heap has its value stored in either an (SMT) integer array, an (SMT) real array or a pointer array. We use additional arrays to store which memory cells are allocated. Those are updated according to the specifications we introduce for malloc, free, and related procedures. Our memory safety checks are implemented by adding additional specifications to these procedures.

2 Software Architecture

Ultimate Kojak is a toolchain in the Ultimate framework. Ultimate is a framework for program analysis and software model checking. It is kept modular such that all tools based on it may use a common infrastructure which, among others, consists of an interface to a SMT-LIBv2 compliant SMT solver, access to interpolation algorithms [2,6], a C parser and translator from C to Boogie [5], other parsers (for Boogie and AutomataScript, an language for describing automata), a plugin that builds a program graph from a Boogie program, a plugin which does large block encoding [1]. Furthermore, Ultimate provides an integration into Eclipse CDT that lets users verify their C programs directly from their IDE and also displays resulting error paths like a debugger.

3 Discussion – Strengths and Weaknesses

Our memory model is simple but sound. The solving algorithm is conceptually sound, too, so we expect to have only correct results. However, the memory model provides no further analysis of the heap, for instance for making restrictions on which pointers may be aliases. That may be a weakness with regards to scalability. Another disadvantage of the memory model is that it is only byte-precise. Thus we cannot deal with bitfields at the moment.

We hope that the new Impulse algorithm needs fewer solver calls and thus scales better than the refinement algorithm we used before which relied on splitting and slicing [3].

4 Tool Setup and Configuration

Ultimate Kojak will compete in all categories of SV-COMP 2015 except *2. Bitvectors 3. Concurrency*, *11. Floats* and *12. Termination*.

The competition version of Ultimate Kojak is available from

`https://ultimate.informatik.uni-freiburg.de/kojak/`

An installation of the SMT solver Z3 is required.[1]

The downloaded archive contains a Python script `Ultimate.py` that provides support for the SVCOMP-compatible input and output of the tool. The directory where the content of the archive lies has to be used as the working directory of the tool. The verification is started by the following command.

```
python Ultimate.py prop.prp inputfile 32bit|64bit simple|precise
```

After a successful run, the script produces the following files:

- `Ultimate.log` A log file containing all output of Ultimate Kojak during the verification run.
- `UltimateCounterExample.errorpath` If we found a counterexample, a human readable version of it will be written to this file.
- `witness.graphml` This file contains an error witness as specified by the SV-COMP rules[2] in GraphML.

References

1. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
2. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
3. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 186–201. Springer, Heidelberg (2012)
4. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 471–482. ACM (2010)
5. Leino, K.R.M.: This is Boogie 2. Manuscript working draft. Microsoft Research, Redmond (2008), `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`
6. Musa, B.: Trace abstraction with unsatisfiable cores. Bachelor's thesis, University of Freiburg, Germany (2013)
7. Nutz, A.: Impulse: a new interpolating software model checker. Master's thesis, University of Freiburg, Germany (2011)

[1] We currently use version z3-4.3.3.f50a8b0a59ff-x64 from `http://z3.codeplex.com/downloads/get/924047`, the directory .../z3/bin must be in the PATH.

[2] `http://www.sosy-lab.org/~dbeyer/cpa-witnesses/`