Ultimate Kojak

(Competition Contribution)

Evren Ermis, Alexander Nutz*, Daniel Dietsch,
Jochen Hoenicke, Andreas Podelski

University of Freiburg, Germany

{ermis,nutz,dietsch,hoenicke,podelski}@informatik.uni-freiburg.de

Abstract. ULTIMATE KOJAK is a symbolic software model checker for C programs. It is based on CEGAR and Craig interpolation. The basic algorithm, described in an earlier work [1], was extended to be able to deal with recursive programs using nested word automata and nested (tree) interpolants.

1 Verification Approach

ULTIMATE KOJAK computes inductive invariants from interpolants to prove the correctness of a program. A program is represented by a program graph. In a program graph, a vertex is a pair consisting of a program location and an invariant describing the abstract program state. An edge is labelled with a transition formula that corresponds to a block of program statements. A program assertion is represented by a transition to an error state where the transition is labelled with the negated assertion. The goal is to show the unreachability of all error states. The program graph is refined by the algorithm presented in a paper by Ermis et al. [1]. This algorithm computes a sequence of interpolants for an infeasible error path and adds them to the invariant annotated at the corresponding vertices. Since the interpolants are only invariants for the particular error path, we also have to add new vertices for the case where the interpolants do not hold. This is achieved by splitting every vertex on the error path into two new vertices where each receives a new invariant: the old invariant conjoined with the interpolant for the first, and the old invariant conjoined with the negated interpolant for the second. Afterwards, the algorithm removes all infeasible edges, thereby refining the abstraction.

The newest version of ULTIMATE KOJAK implements this algorithm and extends it by handling inter-procedural control flow. We use nested word automata to represent programs containing procedures [2]. These automata have **call** and **return** transitions and support procedure summaries to prove the correctness of recursive programs. A **return** transition conceptually has two predecessors: the node representing the call site and the node representing the exit point of the called procedure. Therefore our error paths are in fact trees. To obtain

* Corresponding author.

Fig. 1. We split a node that represents program location ℓ , and that has earlier been annotated with the invariant formula Inv , with the interpolant I . The node and its incoming and outgoing edges are duplicated and one copy of the node is labeled with I and the other with $\neg I$.

interpolants for these error paths, we use tree (nested) interpolation [2,3]. ULTIMATE KOJAK utilizes block encoding [4] to summarize loop-free segments of the program, such that the focus is put on loops.

2 Software Architecture

ULTIMATE KOJAK is a toolchain in the ULTIMATE¹ Verification Framework, which is implemented in Java. ULTIMATE manages different representations of a program and passes them between its plug-ins which may analyse, transform, or visualize the representation. ULTIMATE also provides an interface for communication with SMT-LIBv2 compatible SMT solvers. For parsing C programs, we use the C parser provided by the CDT² project. We use Z3³ for feasibility checks of error paths and transition formulas. Interpolation is done by our own algorithm, which is not yet published [5].

3 Discussion – Strengths and Weaknesses

In our approach, every refinement of the abstraction tends to introduce a rather high number of new edges into the graph, especially when dealing with hyper-edges (i.e. **return** edges), which may lead to a quickly growing model. Currently, interpolants are computed by our own interpolation method which may introduce quantifiers. This makes ULTIMATE KOJAK capable of handling arrays, however quantifiers that can not be easily eliminated lead to a heavy load on the SMT solver.

Due to some remaining issues in our C translation, we have problems with certain constructs, which makes us unable to verify many benchmarks, especially those where some standard library header files were inlined.

¹ <http://ultimate.informatik.uni-freiburg.de>

² <http://eclipse.org/cdt/>

³ <http://z3.codeplex.com/>

In principle, ULTIMATE KOJAK can handle any program that can be formalized in a logic the attached SMT solver is capable of. Currently, we do not support bit-precise treatment of integers or concurrent programs.

4 Tool Setup and Configuration

A commandline version of ULTIMATE KOJAK can be downloaded from

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

The downloaded archive contains a python script `KojakSVComp.py` that provides support for the SV-Comp-compatible input and output of the tool. The directory where the content of the archive lies is used as the working directory of the tool. The verification is started by a command like

`python KojakSvComp.py prop.prp example.c errorPathOutput.txt`

An installation of the SMT solver Z3 is required.⁴ The Z3 executable must be in your PATH environment variable.

5 Software Project and Contributors

ULTIMATE KOJAK has been developed at the Chair of Software Engineering at the University of Freiburg as part of the ULTIMATE verification framework. Over the years numerous contributors have helped to transform a project, which initially started as a students' project, into a chair maintained verification framework on which model checkers such as ULTIMATE KOJAK and ULTIMATE AUTOMIZER⁵ rely on. We would like to thank all the developers and contributors, particularly Matthias Heizmann, Jürgen Christ, Mohamed Abdelazim Sherif, Mostafa Mahmoud Mohamed, Markus Lindenmann, Betim Musa, Christian Schilling, and Stefan Wissert.

References

1. 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)
2. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 471–482. ACM (2010)
3. Christ, J., Hoenicke, J.: Extending proof tree preserving interpolation to sequences and trees (work in progress). In: SMT Workshop, pp. 72–86 (2013)
4. Beyer, D., Cimatti, A., Griggio, A., Erkan Keremoglu, M., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
5. Musa, B.: Trace abstraction with unsatisfiable cores. Bachelor's thesis, University of Freiburg, Germany (2013)

⁴ We use version 4.3.2 for Windows, any recent version should work.

⁵ `http://ultimate.informatik.uni-freiburg.de/automizer/`