Perentie: Modular Trace Refinement and Selective Value Tracking
(Competition Contribution)

Franck Cassez[1,2], Takashi Matsuoka[1],
Edward Pierzchalski[1], and Nathan Smyth[1]

[1] NICTA*, Sydney, Australia
[2] Macquarie University and UNSW, Sydney, Australia

Abstract. Perentie is a software analysis tool based on iterative refinement of trace abstraction: if the refinement process terminates, the program is either declared correct or a counterexample is provided and the program is incorrect.

1 Overview

Perentie is a software analysis tool based on iterative refinement of trace abstraction [1,2], which is a CEGAR-like automata-based technique. The control flow graph (CFG) of a program is viewed as a finite automaton. The accepting states of the CFG are the states reached after a program assertion is violated. This finite automaton generates a language, the *trace abstraction*, of traces that are sequences of uninterpreted instructions. Consequently, all the (uninterpreted) traces accepted by the CFG are *error traces* leading to an error state.

Checking whether a program is correct amounts to determining whether the language of the CFG contains a *feasible* error trace. This is performed by an iterative refinement of the trace abstraction.

Our version of refinement of trace abstraction builds on top of our modular inter-procedural analysis algorithm [3]. Moreover, as the iterative refinement may not terminate, Perentie limits the number of iterations of the refinement phase and if it is inconclusive, it complements it with a second more precise refinement analysis, where it tracks the values of some variables that precisely

* NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 439–442, 2015.
DOI: 10.1007/978-3-662-46681-0_39

define some branching conditions. If this second phase is inconclusive as well, the overall analysis is inconclusive (output is UNKNOWN), otherwise the correctness status of the program is settled (TRUE, FALSE).

2 Software Architecture

Perentie's core engine is developed in Scala. Perentie is flexible and can be configured from the command line by setting a maximum number of iterations for the first refinement phase, and a maximum state space size for the second phase. In this second phase, where some variable values are tracked, the state space may become infinite and this is why we set a bound to ensure termination.

Front end: The front end parser is built on top of the Edison Design Group (EDG) parser. It reads a C source file and generates an XML representation of the C program. The representation is passed on to our own XML parser (written in Scala) that builds a CFG for every function in the source file.

Middle end: Perentie implements a library for manipulating automata including operations like product, union, (lazy) complement, DFS. This allows to extract candidate witness (uninterpreted) error traces from the CFG. Feasibility of a trace is checked using an SMT-solver by encoding the trace in static single assignment (SSA) form into a logical formula and checking for satisfiability. When the trace is infeasible, an *interpolant automaton* [1,3] is computed from a sequence of interpolants [1]. The standard construction requires an interpolating SMT-theorem prover to compute the interpolants from the infeasible trace. As those theorem provers are generally unable to produce interpolants for formulas containing arrays, we have implemented an alternative construction in the style of the weakest pre-condition computation that can compute inductive interpolants, and thus handle programs with arrays.

Back end: Perentie uses SMTInterpol [4] to check satisfiability of SSA formulas. When a program does not contain array variables, it is also used to generate inductive interpolants. Our software architecture is designed to accommodate any SMTLIB2 compliant solver and Z3 is currently being interfaced (although too late to be used for this competition).

3 Strengths and Weaknesses

This first version of Perentie has limited capabilities in terms of supported data structures. Pointers or structs, or arrays of non-integer type are not supported yet, and Perentie will abort the parsing phase with an inconclusive result. Moreover, data types such as `unsigned int` are treated as `int`, and we assume unbounded integers. Although our analysis is sound with unbounded integers, it may generate some false negatives when the actual data type is a bounded integer (overflows/underflows are ignored).

One of the major strengths PERENTIE is that it can discover loop invariants and prove correctness (generate Hoare triples) for programs with parameterised loop bounds (e.g., in the `loop-new` sub-category). The drawback is that to compute useful loop invariants, an interpolating SMT-solver is needed. For the time being, SMTINTERPOL [4] supports interpolants only for the theory of Linear Integer Arithmetic. This prevents us from automatically discovering good loop invariants when the SMT-solver theory does not support interpolation, e.g., when arrays or non-linear arithmetic expressions are used in the program[1]. Another nice feature of PERENTIE is its modular analysis [3] that avoids inlining function calls but this feature is not exercised in SV-COMP 2015.

4 Set Up and Configuration

Participation statement: PERENTIE opts-out from all categories (including Overall) and participates in the Loops.set sub-category of the *Control Flow and Integer Variables* category.

Set up and configuration: PERENTIE is available at `http://ssrg.nicta.com.au/projects/software-verification/perentie/`. The submitted version to SV-COMP 2015 is version 2014-10-31. The current version of PERENTIE requires a 64-bit (x86-64) Linux system, Java (JRE) 6 or higher and gcc. Command line usage is `bash perentie.sh <c-file>`. Usage, set up and configuration is described in the `README.txt` file in the tarball. For this competition, we use PERENTIE in sound[2] mode: when we can determine the result TRUE/FALSE, we output it, otherwise our analysis is inconclusive (parse errors, unsupported data types, theory not supported by the solver) and the output is UNKNOWN.

5 Software Project and Contributors

PERENTIE is developed and hosted by NICTA, Australia, and is currently closed source software. We would like to thank Pablo Gonzalez de Aledo Marugan, University of Cantabria, Spain, for helpful discussions.

References

1. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
2. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013)

[1] This happens only a handful of times in the `Loop` category.

[2] Due to our assumption that integers are unbounded, our analysis is sound only when no overflows occur. Two programs do have overflows related bugs and results in false negatives in our analysis.

3. Cassez, F., Müller, C., Burnett, K.: Summary-based inter-procedural analysis via modular trace refinement. In: FSTTCS 2014, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, New Dehli, India, December 15-17, vol. 29, pp. 545–556 (2014)
4. 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)