Skink: Static Analysis of Programs in LLVM Intermediate Representation (Competition contribution)

Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram,
Pongsak Suvanpong, and Pablo Gonzalez de Aledo

Macquarie University, Sydney, Australia

Abstract. Skink is a static analysis tool that analyses the LLVM intermediate representation (LLVM-IR) of a program source code. The analysis consists of checking whether there is a feasible execution that can reach a designated error block in the LLVM-IR. The result of a program analysis is "correct" if the error block is not reachable, "incorrect" if the error block is reachable, or "inconclusive" if the status of the program could not be determined. In this paper, we introduce Skink 2.0 to analyse single and multi-threaded C programs.

1 Overview

Skink is a static analysis tool that analyses the LLVM intermediate representation (LLVM-IR) of a source program. For instance, Skink can analyse C/C++ programs using the LLVM-IR as generated by the Clang compiler. The objective of the static analysis is to check whether a program is correct w.r.t. a given specification. For C/C++ programs, the specification is provided via `assert(condition)` statements in the C program. The aim of the analysis is to determine whether a condition can be violated. In the SV-COMP setting, `assert` calls `__VERIFIER_error` if the condition is false.

The LLVM-IR representation consists of a collection of *functions* made up of *blocks*. A block represents a sequence of simple instructions (e.g., `store, load`, function calls) and ends with a terminating instruction such as a branch that points to the next block(s). The LLVM-IR we analyse contains a designated "error" block that corresponds to a call to `__VERIFIER_error`. A (feasible) program trace (execution) that contains the error block is an *error trace*. A program is incorrect if and only if it can generate an error trace.

2 Verification Approach

Skink's strategy to determine whether an error trace exists uses the iterative *refinement of the trace abstraction* algorithm of Heizmann [1,2]. First the LLVM-IR of a source program is mapped to a control flow graph (CFG) which is a finite labeled automaton. The labels are the "basic blocks" and the "choices" (branching) of the LLVM-IR. In the automaton the labels are letters and do not carry any special meaning.

The (regular) language accepted by the CFG is the set of traces *leading to an error block*. These traces are *abstract error traces*: the CFG does not give any semantics to the labels, and it is not guaranteed that any such trace is actually *feasible* in the concrete program.

Checking whether a program is correct reduces to determining whether an abstract error trace is *feasible* or equivalently whether the language of the CFG contains a *feasible* abstract error trace. To determine whether a trace is feasible, we take into account the semantics of the instructions of the basic blocks. This is achieved by encoding a trace of the CFG as a logical statement and checking whether this statement is satisfiable or not. If satisfiable, a feasible error trace has been found and the program is incorrect. Otherwise, if a trace t is spurious, an *interpolant automaton* can be computed that accepts t and other traces that are infeasible for the same reason as t [1,2]. In the latter case, we can *refine* the CFG and look for an error trace in the language of the CFG minus the language accepted by the interpolant automaton. When this iterative refinement process stops[1] either no error traces remain and the program is correct or a feasible error trace is discovered and the program is incorrect.

This algorithm can also be used for checking *concurrent* programs using the *product of the CFGs* in each thread. This is in general not very effective as the number of interleavings grows exponentially in the number of threads. The algorithm implemented in Skink 2.0 to analyse multi-threaded C programs extends our previous work that showed how to combine trace abstraction refinement and *partial-order reduction* [3].[2] In Skink 2.0 we have combined trace abstraction refinement with state-of-the-art *dynamic partial order reduction* techniques [4].

3 Software Architecture

Skink 2.0 is developed in Scala and can directly analyse LLVM-IR programs. Skink 2.0 is currently able to analyse programs in C (via Clang) but it is trivially expandable to any language that can be compiled to LLVM-IR.

Front-end: Skink's front-end is written using our sbt-rats parser generator [5][3] and our Kiama Scala library for language processing [6].[4] We have developed a Scala-LLVM parser that can read LLVM-IR and build an abstract syntax tree (AST). Semantic analysis is performed on the AST to recover information such as variable types. Skink 2.0 constructs CFGs for functions from the AST using the Kiama attribute grammar methods.

[1] It may never stop and in this case the analysis is inconclusive.

[2] The implementation introduced in [3] has nothing in common with Skink; it was limited to analysing programs written in a custom input language (but not C) and implemented static partial-order reductions algorithms.

[3] https://bitbucket.org/inkytonik/sbt-rats

[4] https://bitbucket.org/inkytonik/kiama

Middle-end: Our SCALA library AUTOMAT[5] provides the automata-theoretic operations (union, intersection, DFS, partial order reduction) that are needed in the refinement algorithm. This is used to obtain candidate abstract error traces (via a test for language emptiness) and to construct the refinements (difference between two regular languages). On top of the automata-based refinement algorithm, SKINK 2.0 provides two core functionalities. The first is the encoding of an abstract error trace into an SSA form and eventually a logical formula; this logical formula is satisfiable if and only if the trace is feasible. Satisfiability is determined by an SMT-solver (see *Back-end* section below). The second is the computation of an *interpolant automaton* which is based on an annotation of an infeasible trace with *invariants*.

Back-end: To check whether a (symbolic) abstract error trace is feasible we use a SCALA abstraction over the SMTLIB standard for common languages and interfaces for SMT solvers. Our library MQ-SCALA-SMTLIB[6] provides this abstraction. MQ-SCALA-SMTLIB was also developed using SBT-RATS and KIAMA.

We support most of the SMT-solvers (including Z3[7], SMTINTERPOL[8] and CVC4[9]) via a common SCALA abstract interface. As a result we can choose which solver to use at run time, and we may use multiple and different solvers during the same program analysis. In the current implementation, SKINK 2.0, we mostly use Z3, SMTInterpol and CVC4, depending on the theories and operations we need (linear integer arithmetic, arrays, bitvectors, interpolants) and on the SV-COMP categories to analyse.

4 Strengths and Weaknesses

SKINK 2.0 does not support the full LLVM-IR assembly language and our front-end parser may fail to parse some LLVM-IR input. Another limitation of SKINK 2.0 is that we use the LLVM *inlining* capability (`opt -inline`) to obtain a single CFG for each LLVM-IR. This may fail preventing the subsequent program analysis. These limitations should be overcome in the next months by extending our front-end Scala-LLVM parser and implementing our modular analysis technique [7]. We may assume unbounded integers in the analysis and this may result in false negatives due to overflow/underflow errors (in our tests it happened once in the ControlFlow category).

On the positive side, SKINK 2.0 can analyse programs that can be compiled into LLVM-IR which makes it usable on a variety of languages including C/C++, Objective C and Swift. A major strength of SKINK 2.0 is that it can discover loop invariants (interpolants) and is able to establish program correctness. Our abstract SCALA solver library MQ-SCALA-SMTLIB provides access to a number of theories (Arrays, BitVectors) and solver capabiltities (generate interpolants).

[5] https://bitbucket.org/franck44/automat

[6] https://bitbucket.org/franck44/mq-scala-smtlib

[7] https://github.com/Z3Prover/z3

[8] https://ultimate.informatik.uni-freiburg.de/smtinterpol/

[9] http://cvc4.cs.nyu.edu/web/

Skink 2.0 is, to the best of our knowledge, the only tool that combines trace abstraction refinement with a version (source-DPOR) of the optimal state-of-the-art *dynamic partial order reduction* algorithm [4]. This enables us to efficiently verify some programs in the Concurrency benchmarks category.

5 Set Up and Configuration

Participation statement: Skink opts-out from all categories except *Integer and Control Flow*, *Concurrency* and *BitVectors*.

Set up and configuration: Skink 2.0 is available from http://science.mq.edu.au/~fcassez/sw/skinkv2.0.tgz. The archive includes all dependencies needed to run it on Ubuntu Xenial Xerus 64-bit (16.04.1). `skink.sh` is the simplest and the recommended way to run this Skink 2.0 distribution[10]. `skink.sh` should be passed the C file on which analysis is to be performed. It will place along that file the verification output (`.verif`) and the witness file (`.graphml`) as appropriate.

6 Software Project and Contributors

Skink 2.0 is developed by F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Gonzalez, P. Suvanpong at the Department of Computing, Macquarie University. The libraries used in Skink 2.0 are open-source software. More information can be found at http://science.mq.edu.au/~fcassez/software-verif.html.

References

1. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In Palsberg, J., Su, Z., eds.: SAS. Volume 5673 of LNCS., Springer (2009) 69–85
2. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In Sharygina, N., Veith, H., eds.: CAV. Volume 8044 of LNCS., Springer (2013) 36–52
3. Cassez, F., Ziegler, F.: Verification of concurrent programs using trace abstraction refinement. In Davis, M., Fehnker, A., McIver, A., Voronkov, A., eds.: LPAR-20 2015. Volume 9450 of LNCS., Springer (2015) 233–248
4. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In Jagannathan, S., Sewell, P., eds.: POPL '14, San Diego, CA, USA, January 20-21, 2014, ACM (2014) 373–384
5. Sloane, A.M., Cassez, F., Buckley, S.: The sbt-rats parser generator plugin for Scala (tool paper). In: SCALA 2016, New York, NY, USA, ACM (2016) 110–113
6. Sloane, A.M.: Lightweight language processing in Kiama. In GTTSE 2009, Braga, Portugal, July 6-11, 2009. Volume 6491 of LNCS., Springer (2009) 408–425
7. Cassez, F., Müller, C., Burnett, K.: Summary-based inter-procedural analysis via modular trace refinement. In FSTTCS 2014. Volume 29 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014) 545–556

[10] The required Scala run-time and libraries are bundled into the `skink.jar` file.