Symbiotic: Synergy of Instrumentation, Slicing, and Symbolic Execution[*]
(Competition Contribution)

Jiri Slaby, Jan Strejček, and Marek Trtík

Faculty of Informatics, Masaryk University
Botanická 68a, 60200 Brno, Czech Republic
{slaby,strejcek,trtik}@fi.muni.cz

Abstract. Symbiotic is a tool for detection of bugs described by finite state machines in C programs. The tool combines three well-known techniques: instrumentation, program slicing, and symbolic execution. This paper briefly describes the approach of Symbiotic including its strengths, weaknesses, and modifications for SV-COMP 2013. Architecture and installation of the tool are described as well.

1 Verification Approach

Symbiotic implements our technique [4] that combines instrumentation, program slicing, and symbolic execution in order to detect bugs described by finite state machines. More precisely, we instrument a given program with code that tracks runs of state machines representing various erroneous behaviors. If an instrumented state machine enters an error location during a program execution, then the original program contains a bug specified by the machine. After instrumentation, we slice [5] the program to reduce its size without affecting runs of state machines. Finally, we symbolically execute [3] the sliced program to find bugs in the program.

As reachability of an `ERROR` label is the only bug considered in the SV-COMP, we have modified our instrumentor to put `assert(0)` function calls at `ERROR` labels in the code. Given the instrumented code, we execute Clang to produce an Llvm bitcode. This is in turn interprocedurally sliced with respect to slicing criteria, which are the instrumented `assert` calls. In other words we remove all the code except the one that has an effect on reachability of `assert` calls. The sliced Llvm bitcode is finally symbolically executed by Klee [1]. There are several possible outputs that Klee may generate. It can either find a reachable `assert` and report it, or finish the computation without any report, or terminate in some errant way (out of time, out of memory, invalid memory dereference, or some internal error for example). We map these to the demanded answers: `UNSAFE`, `SAFE`, or `UNKNOWN` respectively. This is taken care of in a simple scripted filter.

[*] This work has been supported by The Czech Science Foundation (GAČR), grant No. P202/12/G061.

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 632–634, 2013.

2 Software Architecture

Libraries/External Tools. For the code translation and for operations with the LLVM bitcode we use LLVM/CLANG 3.1[1]. The symbolic executor is KLEE which itself uses the STP constraint solver [2]. We obtained both KLEE and STP from the respective GIT repositories and used the snapshots. For more information about KLEE, see [1].

Software Structure and Architecture. The architecture described in Section 1 is summarized in Figure 1. The slicer is written as a plug-in for the optimizer `opt` from the LLVM suite. It is publicly available in a separate repository at `https://github.com/jirislaby/LLVMSlicer/`. The slicer improves the symbolic execution considerably. For ease of use, all parts of the tool pipeline are one by one run by a single script `runme`.

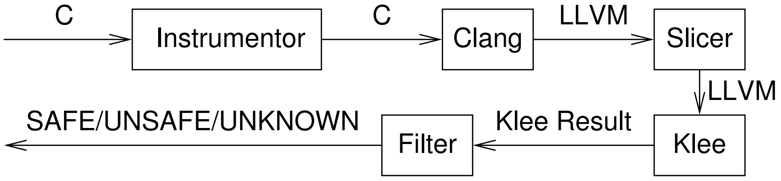

Fig. 1. Pipeline of the tool

Implementation Technology. The instrumentation is performed by a `bash` script using `sed`. The final filter also uses `bash` with the help of `grep`. The rest of the toolchain is written in C++ and compiled using `gcc`.

3 Discussion of Strengths and Weaknesses of the Approach

The strength of the tool lies in its high precision of the answers. In theory, the only source of incorrect answers is the slicer: it can completely remove an infinite loop in some cases and thus an unreachable `ERROR` label located below the loop may become reachable. However, there is no such case in the competition benchmarks.

All incorrect answers produced by our tool in the competition are due to bugs in implementation. Since the tool submission, we have fixed most of the bugs and improved the implementation a bit. The biggest change on the competition benchmarks can be seen in the category *FeatureChecks* (118 files):

[1] `http://llvm.org`

	Competition Version	Current Version
Correct Answers `SAFE/UNSAFE`	81	116
Incorrect Answers `SAFE/UNSAFE`	17	0
`UNKNOWN` (including timeouts)	20	2

In general, the main weakness of the tool is a high percentage of `UNKNOWN` results. These results come mainly from high computation cost of the symbolic execution of programs with loops or recursion. This problem is relieved by slicing, but there are still many cases where the sliced code remains complex and symbolic execution runs out of time or memory. Other sources of `UNKNOWN` results are internal errors of Klee and general limitations of constraint solving.

4 Tool Setup and Configuration

Download and Installation Instructions

- *Requirements:* `llvm-3.1`, `clang-3.1`.
- Download Symbiotic 1 at: `https://sf.net/projects/symbiotic/`.
- Change the current directory to `/opt` (this location is needed for Klee).
- Untar the archive with the tool.
- Change directory into `/opt/symbiotic`.
- Run `./runme <benchmark.c>` for each source file in the set.

Results Reported `SAFE/UNSAFE/UNKWNOWN` answers match the competition rules. The counterexample for an error in some `<benchmark.c>` is generated for each error path in the benchmark to `<benchmark.c>-klee-out/`. There is also a sliced code referred by all the error paths.

5 Software Project and Contributors

The concept of the tool has been developed by the authors of this paper. The tool was implemented by Jiri Slaby (contact person), Marek Trtík, and Ben Liblit (several fixes). It is available under the GNU GPLv2 License and is hosted by the Faculty of Informatics, Masaryk University.

References

1. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, pp. 209–224. USENIX Association (2008)
2. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
3. King, J.C.: Symbolic execution and program testing. Communications of ACM 19(7), 385–394 (1976)
4. Slabý, J., Strejček, J., Trtík, M.: Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 207–221. Springer, Heidelberg (2012)
5. Weiser, M.: Program slicing. In: Proceedings of ICSE, pp. 439–449. IEEE Press (1981)