GoannaSMT – A Static Analyzer with SMT-based Refinement

Mark Bradley, Franck Cassez, Ansgar Fehnker,
Thomas Given-Wilson and Ralf Huuck

NICTA, University of New South Wales,Sydney, Australia
firstname.lastname@nicta.com.au

Maximilian Junker

Technische Universität München, Munich, Germany
junkerm@in.tum.de

Abstract

We present an industrial strength static analysis tool for automated bug detection in C/C++ source code called GoannaSMT. The underlying technology of GoannaSMT is an automata-based approach to static analysis, where high-level syntactic source code abstractions are subjected to a custom-built explicit state model checker. Resulting error traces are then subjected to an SMT solver in a path-refinement loop for closer inspection of their feasibility. As a result GoannaSMT is highly precise while at the same time scaling to millions of lines of code. We present the core technology, architecture, and experiences.

Keywords: SMT solving, refinement, static analysis, model checking, tools, C/C++

Core Technology. GoannaSMT is a static analysis tool for detecting programming flaws and security vulnerabilities in C/C++ source code that implements the CEGAR paradigm (see Figure 1). The GoannaSMT core uses a program abstraction consisting of control flow graphs (CFG) annotated with locations of interest. Given a property to check for, the possible paths through the CFG are then investigated by a tailored explicit-state model checker. The output of the model-checker is that the property is true or false. In the latter case, the model-checker provides an (abstract) counter example which is a trace in the abstract model. Some counter examples are artifacts of the abstraction and are not real bugs: such a counter example is *spurius* and

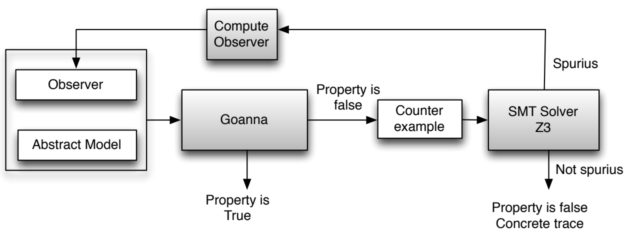

Fig. 1. SMT-based path refinement loop

the warning issued by the analysis tool is a *false positive*. To be useful for a programmer, a static analysis tool should produce very few false positives.

In GoannaSMT, the counter example provided by the model-checker is fed to an SMT solver (Z3) to check whether it is a feasible trace in the program. This check is performed using a fine-grained semantic analysis (including a memory model for pointers.) If the counter example is feasible, then a concrete program trace is fed back to the programmer. Otherwise if a trace is spurius, an *observer* is computed that prevents this trace (and many similar paths) to occur in the refined model. Typically this rules out many paths at once and so this refinement process usually only runs a few times (an upper limit for the number of refinements is set by the user.)

GoannaSMT checks for around 150 different classes of bugs both within functions and interprocedurally across function boundaries. Despite doing so many checks, GoannaSMT remains efficient by only invoking the SMT solver when a potential bug is found (which is relatively rare). Further performance improvement is gained by exploiting the results of the SMT solver.

Architecture. GoannaSMT is a modified and improved academic version of Goanna.[1] It employs several state-of-the-art techniques including: pattern-matching based on tree-automata, explicit-state CTL model checking, abstract interpretation for range checking, and SMT solvers for refinement. Goanna-SMT is available as both command line (Linux and Windows) and IDE versions that integrate with Eclipse and Visual Studio.

Experience. On the *Wireshark* code base written in C/C++ with 1.4 million *lines of code* (LoC) (16 million LoC after pre-processing) GoannaSMT takes slightly under 3 hours, which is a run-time overhead of 15% compared to the commercial version of Goanna. The SMT refinement loop is able to remove 48 relevant false positives correctly, which in this case equates to 100% of known false positives. For the approximately 53,000 functions in the same code base, the SMT refinement loop reached the user-defined upper limit of 20 iterations 11 times. While those cases were not false positives in that code base, in general this might count as unrefuted spurious bugs. Given the size of the code base and much room for optimization, these are very good early results.

[1] http://www.nicta.com.au/goanna