Proving Reachability using FShell*
(Competition Contribution)

Andreas Holzer[1], Daniel Kroening[2], Christian Schallhart[2], Michael Tautschnig[2], and Helmut Veith[1]

[1] Vienna University of Technology, Austria
[2] University of Oxford, United Kingdom

Abstract FShell is an automated white-box test-input generator for C programs, computing test data with respect to *user-specified* code coverage criteria. The pillars of FShell are the declarative specification language FQL (FShell Query Language), an efficient back end for computing test data, and a mathematical framework to reason about coverage criteria. To solve the reachability problem posed in SV-COMP we specify coverage of ERROR labels. As back end, FShell uses bounded model checking, building upon components of CBMC and leveraging the power of SAT solvers for efficient enumeration of a full test suite.

1 Overview

FShell implements automatic white-box test-input generation according to a user-defined coverage specification given in the declarative language FQL [5,6]. To resemble formal verification and solve the reachability problem presented in the SW Verification Competition we specify coverage of all ERROR labels.

FQL is built on top of a concise mathematical framework for formalising code coverage criteria. This framework enables automatic processing of FQL queries and together with FQL makes our approach oblivious to the algorithmic details of test-input generation. As this overall architecture is analogous to that of databases, we refer to our approach as *query-driven program testing* [4].

As back end for solving FQL queries, i.e., computing test inputs, FShell uses components of the C bounded model checker (CBMC) [1], which enables support for full C syntax and semantics, and makes efficient use of SAT solving (using MiniSat 2.2.0 [2]). An overview of the architecture is presented in the next section. The technical approach was first sketched in [3], further refined in [4] and full details of the current implementation can be found in [7].

2 Architecture

FShell comprises two main parts: The *front end* handles user interactions with a command-line interface. There, control commands such as loading source files,

* This research is supported by the FWF research network RiSE, the WWTF grant PROSEED and EPSRC project EP/H017585/1

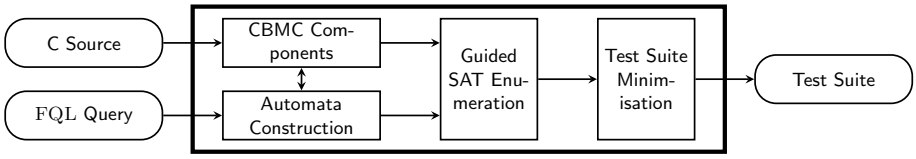

Figure 1. Query processing

macro definitions, and FQL queries are entered by the user. The *back end* performs the actual test case generation. Figure 1 sketches the conceptual steps:

1. We use the code base of CBMC to first obtain an intermediate representation (GOTO functions) of the program under test.
2. We translate the given FQL query into automata over statements in the GOTO functions. We refer to these automata as *trace automata*.
3. CBMC builds a Boolean equation describing all states yielding a violation of a given property (assertion) of the input program. It is then able to produce an example of such a violation (counterexample).
   In test case generation, we use this scheme by adding the property stating that *no* program execution accepted by the trace automata exists. Any counterexample (i.e., test case) then describes a path that fulfills the query. We efficiently enumerate test cases using guided SAT enumeration [4]. The resulting test suite satisfies the given coverage specification by construction.
4. To remove redundant test cases, we perform test suite minimisation. This problem is an instance of the minimum set cover problem, which we reduce to a series of SAT instances.
5. We display the generated test suite as a list of initial values of variables.

3 Strengths and Weaknesses

The main advantage of FShell over its underlying back end CBMC is computing counterexamples for *all* reachable error locations. Apart from this fact, however, FShell mainly acts as bounded model checker in this competition. As such, FShell was successful as it was, together with SatAbs, one of only two competition participants that never reported a spurious "UNSAFE".

As bounded model checking is necessarily incomplete for all benchmarks with unbounded loops, a choice had to be made how to handle those. Without further options, FShell performs an additional step to prove given loop unwindings (see below) to be sufficient. This step would ensure that FShell does not return "SAFE" without having properly proved safety, but instead aborts early.

As consequence, however, FShell would not have scored any points on programs with unbounded loops. Therefore we decided to disable the early abort and instead perform verification under the assumption that the given loop bounds suffice. On the one hand, this is helpful in bug finding and permitted to prove unsafety on several benchmarks with unbounded loops. Whenever this loop bound is insufficient for finding paths to the error location, however, FShell wrongly reports "SAFE" and becomes unsound.

This is the classic bounded model checking setting, but proved to be even less successful under the presented scoring system: in the categories "ControlFlowInteger" and "DeviceDrivers" FShell correctly determined the result in more than 70% of the instances, but scored only 19% of the possible points. For all other categories (except for "Concurrency", which is presently unsupported) problems in the back end caused verification to fail; these have mostly been fixed by now and future versions are expected to perform much better.

4 Tool Setup

The competition participant is FShell version 1.3, which can be downloaded from http://code.forsyte.de/fshell. To avoid the interactive operation of FShell, a file "query" should first be set up containing the following statements:

```
cover @label(ERROR)
quit
```

Then, FShell can be run as

```
fshell --unwind 10 --no-unwinding-assertions --query-file query FOO.c
```

for a source file "FOO.c". By default, FShell will assume a 64-bit memory model as this is the competition platform. For those benchmarks written with a 32-bit memory model in mind, the option `--32` must be given in addition.

By definition, FShell produces test inputs instead of full counterexample traces; each set of inputs uniquely determines a single execution, however. An instance is found to be "SAFE" if no test inputs are found. In this case, FShell prints `Test cases: 0` – an "UNSAFE" instance yields a non-zero count.

Software Project. FShell is maintained by Michael Tautschnig as an extension of CBMC. FShell was jointly designed by the authors. FShell is released at the above web site as binary for several platforms under an Apache 2.0 license.

References

1. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176. Springer (2004)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518. Springer (2003)
3. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: CAV. pp. 209–213. Springer (2008)
4. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: VMCAI. pp. 151–166. Springer (2009)
5. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite ? In: ASE. pp. 407–416. ACM (2010)
6. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: An introduction to test specification in FQL. In: HVC. pp. 9–22. Springer (2010)
7. Tautschnig, M.: Query-Driven Program Testing. Ph.D. thesis, Vienna University of Technology (2011)