FSHELL: Systematic Test Case Generation for Dynamic Analysis and Measurement*

Tool Paper

Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith

Technische Universität Darmstadt, Germany

Abstract. Although the principal analogy between counterexample generation and white box testing has been repeatedly addressed, the usage patterns and performance requirements for software testing are quite different from formal verification. Our tool FSHELL provides a versatile testing environment for C programs which supports both interactive explorative use and a rich scripting language. More than a frontend for software model checkers, FSHELL is designed as a database engine which dispatches queries about the program to program analysis tools. We report on the integration of CBMC into FSHELL and describe architectural modifications which support efficient test case generation.

1 Introduction

This paper introduces our prototype tool FSHELL which supports both interactive and scripted test case generation for real-world C code. We have consciously designed FSHELL in analogy to a database engine; FSHELL uses a query language tailored for program analysis, and dispatches queries about program paths to software analysis tools. The query language is built around the concept of a *test job*, i.e., a specification of program paths along with coverage requirements and other parameters, and provides the system engineer with convenient primitives for test job management and test job execution.

In the first iteration of the project, we have integrated CBMC [1] as model checking backend. We are using SAT enumeration techniques to generate families of test cases subject to coverage criteria in a *single run* of the model checker. We note that the program-as-database metaphor has been previously used in the BLAST project [2]. The query language of BLAST [3,4], however, is tailored towards model checking, and testing activities focus on basic block coverage only [5]. Other tools using model checkers for test case generation are Java PathFinder [6] and SAL2 [7]. Unlike FSHELL, neither of them supports full C semantics. Conversely, the CUTE toolkit implements concolic testing [8], which results in a tool that uses directed testing to form a model checker.

2 Features of the FSHELL Environment

The FSHELL environment assists the user to create, manage and execute test jobs. With each source file, the tool automatically associates a *generic test job* which the user

* Supported by DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1).

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 209–213, 2008.

annotates and modifies for the purpose at hand: On the one hand, *structural constraints* restrict the test job to program paths matching a regular expression over program locations annotated by program predicates. To this end, FSHELL provides convenient primitives for structural notions of C programs such as function headers, labels etc. Internally, FSHELL represents structural constraints as *path automata*. On the other hand, *quality constraints* require the *set of test cases* generated by a test job to meet minimal requirements such as basic block coverage, condition coverage or predicate coverage (cf. [9]). Importantly, different quality constraints can be enforced locally in the program. An example of the representation of these requirements is given in Sec. 3.

Additional commands for *test job management* enable the user to maintain and develop multiple test jobs for complex applications. Thus, test jobs are viewed as objects which are loaded, stored, duplicated, merged, etc. When a test job is prepared, the user invokes *test case generation* commands to compute a *test suite* which satisfies the constraints expressed by the test job. FSHELL also provides support for *test case execution* to observe non-functional properties not only on desktop machines, but also on embedded devices. Fig. 1 presents an example session of FSHELL. We show the query used to generate a test suite with predicate coverage for `bubble.c` for an array of size 20.

```
1 void bubble(int a[], int N) {
2   int i, j, t;                          > ADD_SOURCE(bubble.c);
3   for (i = N - 1; i >= 0; i--)          > GENERATE_TC(COVERAGE(ENTRY(main),
4     for (j = 1; j <= i; j++)              EXIT(main), PREDICATE)) AS tc1;
5       if (a[j - 1] > a[j])              > SHOW(tc1);
6         SWAP(a[j - 1], a[j]);           IN: a={0,1128267777,...,1465438334}
7 }                                       IN: a={0,-2139095040,...,1709483866}
8 void main() {int a[20]; bubble(a, 20);}
```

Fig. 1. Source code of `bubble.c` and corresponding FSHELL session

3 Tool Architecture

We chose CBMC 2.4 as the first backend for FSHELL because (1) it supports full C syntax and semantics, (2) BMC is conceptually closer to testing than an abstraction/refinement approach, (3) the source code is available, and (4) it is well engineered and offers a very clean design and a stable code base. FSHELL is, like CBMC, implemented in C++ and accounts for 13k lines of code.[1]

The design of FSHELL is based on three main layers. The *frontend* handles user interactions with a command line interface. There, job control commands, such as loading source files into the test job, and constraint specifications are entered by the user. The *management layer* implements control commands and redirects queries to the appropriate *backend*. The backend performs the actual path feasibility analysis and test case

[1] Visit `http://code.forsyte.cs.tu-darmstadt.de/fshell` to follow development.

generation. Both the management and the backend access a single shared cache. This cache stores all queries and their respective results. Fig. 2 gives an overview of the collaboration of components. Dashed rectangles represent parts reused from CBMC. Both path feasibility analysis and test case generation follow a bounded model checking workflow. In path analysis, the CBMC modules in use are unmodified and only invoked from within our code. Test case generation, however, requires an additional control flow graph analysis. To this end, we collect conditions along possible control flow paths and identify the relevant literals used in the SAT encoding. Further, to allow for highly efficient test case generation of large test suites, the SAT solver is invoked in an incremental fashion, retaining the conflict clause database. In this process, we use both blocking clauses and assumptions in a way that guarantees the conflict clause database to remain consistent. By the design of our procedure, the resulting SAT solver invocation returns UNSAT only if there is no further matching test case, i.e., all coverage criteria are satisfied.

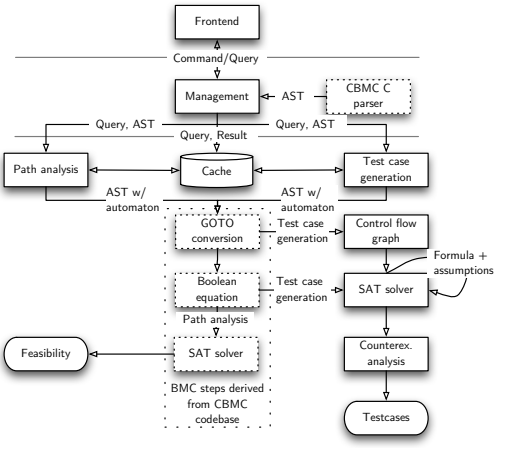

Fig. 2. Main components of FSHELL

As an example, consider decision coverage. In the test job the user specifies a segment of the program where all reachable branches have to be covered. From the AST and the CNF formula, FSHELL obtains a set B_0 of Boolean variables which correspond to the branches to be taken, and thus characterize the coverage criterion. Initially, we add B_0 as a clause, thus requiring that at least one of the variables is set to true, i.e., at least one of the branches is taken within the critical segment of code. Subsequently we compute clause B_{i+1} from B_i by removing already satisfied variables from B_i, and add the respective clause to the SAT instance. This incremental SAT solving process ends if either $B_{i+1} = \emptyset$ or the instance is found to be unsatisfiable. In this case, B_{i+1} marks the set of infeasible branches.

4 Experimental Results

In our experiments, we first analyzed an industrial engine controller which was autogenerated from a MATLAB/Simulink model. The resulting C source code of 2033 LLOC[2] was, without applying any abstraction, tested for basic block coverage. FSHELL achieved coverage with only five test cases, taking 18.18 seconds on a 3.2 GHz Intel P4 equipped with 3 GB of RAM.

[2] logical lines of code, i.e., number of occurrences of ';'

As the concepts implemented in FSHELL are related to both directed testing and model checking, we chose BLAST as benchmarking reference. Still, it should be noted that BLAST is a full fledged model checker and is not as optimized towards testing as FSHELL is. Their set of benchmarks, presented in [5], is well documented and all source files are publicly available. To achieve equivalent test goals, we generated test suites with full basic block coverage. Apart from `parclass.i`, in the tests of Table 1 FSHELL was run on the P4 system mentioned above. The file `parclass.i` required a cleanup of conflicting `typedef`'s and more than the addressable 2 GB of main memory. On a 3.0 GHz AMD64 system we succeeded with a memory usage of 2.3 GB. The results for BLAST are taken literally from [5], because the version of BLAST performing test case generation is currently unavailable. The hardware used, however, is very similar. We observe that FSHELL typically returns a higher number of test cases to achieve basic block coverage, but it takes less time to do so. We believe that these performance improvements outweigh the larger test sets. Nevertheless we plan to include minimization strategies in FSHELL.

Table 1. Results on device drivers

		BLAST		FSHELL		Speed-
Source file	LLOC	#cases	Time[s]	#cases	Time[s]	up
`kbfiltr.i`	4879	39	300	66	17	17.9
`floppy.i`	6435	111	1500	288	1305	1.1
`cdaudio.i`	8022	85	1500	159	748	2.0
`parport.i`	20698	213	5460	312	1999	2.7
`parclass.i`	45283	219	2520	716	1511	1.7

Additionally, we generated test suites for sorting algorithms literally taken from [10]. The experiments are parameterized by the size of the array to be sorted. In Table 2, we present the speedup achieved by generating covering test suites in a single test job, compared to naïve iterative invocations of the model checker. Note that in the latter case, coverage constraints are not even considered. For each algorithm and each array size, we show the speedup for basic block- and condition coverage.

Table 2. Speedup for basic block/condition coverage

Source File	5	10	15	20
`bubble.c`	1.88/1.98	1.86/1.89	1.96/1.58	1.74/1.26
`insertion.c`	1.63/1.95	1.56/2.19	0.95/1.39	0.98/2.13
`selection.c`	2.13/1.76	1.41/2.01	1.38/1.97	0.99/2.27

Conclusion. We presented FSHELL as an environment to facilitate white-box testing of C programs. By design, FSHELL treats a C program as a database to be queried by the user. FSHELL serves as a framework which integrates multiple program analysis backends. Our experimental results confirm the practical feasibility and relative efficiency of our approach.

Acknowledgments. We are grateful to Raimund Kirner, Sven Bünte, Ingomar Wenzel, and Michael Zolda for discussions on the topic of this paper. Further we thank Dirk Beyer for his help with BLAST.

References

1. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
2. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST Query Language for Software Verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
4. Beyer, D., Noack, A., Lewerentz, C.: Simple and Efficient Relational Querying of Software Structures. In: WCRE, pp. 216–225 (2003)
5. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE, pp. 326–335 (2004)
6. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with java PathFinder. In: ISSTA, pp. 97–107 (2004)
7. Hamon, G., de Moura, L.M., Rushby, J.M.: Generating Efficient Test Sets with a Model Checker. In: SEFM, pp. 261–270 (2004)
8. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: FSE, pp. 263–272 (2005)
9. Ntafos, S.C.: A comparison of some structural testing strategies. IEEE Trans. Software Eng. 14(6), 868–874 (1988)
10. Sedgewick, R.: Algorithms in C. Addison-Wesley Publishing Company, Inc., Reading (1990)