SmacC: A Retargetable Symbolic Execution Engine *

Armin Biere[1], Jens Knoop[2], Laura Kovács[3], and Jakob Zwirchmayr[2]

[1] JKU Linz, [2] TU Vienna, [3] Chalmers University of Technology

Abstract. SmacC is a symbolic execution engine for C programs. It can be used for program verification, bounded model checking and generating SMT benchmarks. More recently we also successfully applied SmacC for high-level timing analysis of programs to infer exact loop bounds and safe over-approximations. SmacC uses the logic for bit-vectors with arrays to construct a bit-precise memory-model of a program for path-wise exploration.

1 Introduction

Symbolic execution executes a program by using symbolic instead of concrete data. Typically, the program is analyzed path-wise, i.e. paths are analyzed one-by-one in isolation. Splitting the analysis to focus on single paths can be exploited to track important information about the path under analysis and allows to check properties where other techniques fail, for example as illustrated in Fig. 2(c). However, for whole program analysis the costs of path-wise symbolic execution are often prohibitive because of the so-called path-explosion problem that the number of paths grows exponentially with the number of conditionals in a program. Fortunately, even analyzing only parts of the program, such as focusing on all paths within a certain function, still allows to infer valuable properties and catch subtle errors.

In this paper we present SmacC, a retargetable symbolic execution engine. SmacC is an acronym for *SMT Memory-model and Assertion Checker for C*. Retargebility, a term borrowed form [4] inspired the front-end implementation of SmacC, and refers to its capability of being retargetable to conceptually quite different applications in program analysis. SmacC supports a relevant fragment of (ANSI) C analyzing such programs by path-wise symbolic execution. It derives verification conditions for program statements and expressions, expressed as satisfiability modulo theory (SMT) formulas in the logic of bit-vectors with arrays. This allows bit-precise reasoning about the program, including reasoning about memory accesses and arithmetic overflow. The generated verification conditions precisely capture the memory-model of the program. Proving them to hold guarantees that the supported runtime- and memory-errors cannot occur. Violations in the symbolic representation constitute actual violations.

SmacC can be applied in a number of program analysis settings. The tool can prove absence of runtime-errors if full symbolic coverage is achieved. Further, it allows to perform bounded model checking by exhaustive symbolic execution up to a provided bound. Functional correctness, e.g. equivalence checking, is supported via assertions. Generated verification conditions can be dumped to files and used as SMT benchmarks

*This research is supported by the FP7-ICT Project 288008 T-CREST, the FWF RiSE projects S11408-N23 and S11410-N23, the WWTF PROSEED grant ICT C-050, the FWF grant T425-N23, and the CeTAT project of TU Vienna.

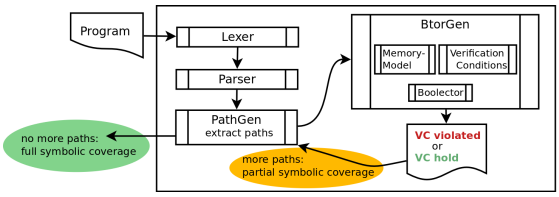

Fig. 1. Architecture of SmacC: path-wise execution leads to *partial symbolic coverage* if there are more paths to be executed. Exhaustive execution of all paths yields *full symbolic coverage*.

for testing or performance evaluation of SMT solvers. A new application for SmacC is the high-level worst-case execution time (WCET) analysis of programs. More specifically, the tool finds flow-facts, such as infeasible paths and safe loop bounds, required for successful WCET analysis. We use SmacC in combination with the WCET analysis toolchain r-TuBound [5].

SmacC is implemented in 10Klocs of C and is available at `http://www.complang.tuwien.ac.at/jakob/smacc/`

2 Tool Architecture

Figure 1 shows the architecture of SmacC. SmacC reads a C program as input file, which is then tokenized (`Lexer`) and parsed to abstract syntax trees according to the C expression grammar (`Parser`). The abstract syntax trees are stored as elements of a code-list. Paths through the program are extracted (`PathGen`) and symbolically executed (`BtorGen`), which consists of updating the symbolic representation of the executed path. This symbolic representation is used to generate verification conditions in form of SMT formulas, which express runtime-safety of statements occurring on the path. We use the SMT solver Boolector [2] for checking these SMT formulas in the quantifier-free logic of bit-vectors with arrays. In the sequel, we overview the main ingredients of SmacC, and refer to [6] and the url for further details.

PathGen. In the path-generation phase, in order to remove loops, the code-list is flattened, by unwinding program loops up to a certain bound. This way, for each program path, a code-list is constructed. Conditionals, which require to split the control-flow, will produce two paths to explore both branches of the condition. Each fully extracted path is then symbolically executed in `BtorGen`.

BtorGen, Memory-model and Verification Conditions. This step constructs a symbolic SMT representation of the memory used in the program, faithfully covering the semantics of each statement on the program path. Additionally, verification conditions are constructed as SMT formulas. The *program memory* is a collection of symbolic values and modeled by a contiguous array. The memory layout, e.g. the set of declared addresses, is represented by bit-vector variables indexing the memory array. Additional bit-vector variables symbolically track allocated memory regions. Unwritten memory is treated as uninitialized. Verification conditions supported by SmacC include reasoning about *return* statements (check if the program can or returns a specified value), *path conditions* (check satisfiability of conditionals), *division by zero*, and *overflow* of arithmetic operations. Our bit-precise memory-model allows us to construct verification conditions for memory accesses as follows: an access is considered *out-of-allocated* if the address can evaluate to an unallocated array index, i.e. outside the region constrained by $global\_beg$, $global\_end$, $heap\_beg$, $heap\_end$, $stack\_beg$ and $stack\_end$.

Output results. SmacC produces as output a textual report for each statement symbolically executed along all analyzed paths. For each verification condition, the tool reports whether the property is safe or violated on a specific path. If a verification condition is violated on at least one path, then the corresponding property can be violated by an actual run. If the verification condition holds on all paths, then the corresponding program property cannot be violated by any actual run.

3 Applications of SmacC

We have successfully applied SmacC to verify C programs and generate SMT benchmarks using our precise memory-model [6]. We illustrate the bit-precise memory-model and generation and proving of verification conditions using the examples in Fig. 2(a) and (b) below. We also integrated SmacC with r-TuBound to support timing analysis, and show its use on Fig. 2(c). For more details, we refer to the url of our tool.

```
1: int a[4];                 int main () {           int main() {          :1
2: int main () {               int x, y;              int i, flag;        :2
3:  int i;                     if (x > 0) {            for(i=0; i<5; i++) :3
4:  a[0] = 1;                    y = x * x;             if(i==4 && flag){ :4
5:  for(i=0; i<4; i++)           if (y == 0)             i = 0;           :5
6:   if (a[i] > 0)                 assert(0); }}         flag = 0; }}     :6
7:    i = i + 1;                                                          :7
8:  assert(i >= 4); }                                                     :8
```

(a) (b) (c)

Fig. 2. (a) a program with an assertion and a conditional update; (b) a program with a reachable, failing, assertion; (c) SmacC finds the loop bound, CBMC keeps unwinding the loop.

Example. The variable declarations in the program of Fig. 2(a) in lines 1 and 3 (a:1,3), result in the following SMT variable declarations, where variables that do not occur in the source are used to track allocated memory: $global\_beg$, $global\_end$, $heap\_beg$, $heap\_end$, $stack\_beg$, $stack\_end$, mem, i, x, a, where mem is an array and models memory. Symbolic execution of a path tracks declared memory constructing the formula $(a = global\_beg) \wedge (global\_end = global\_beg + 16) \wedge (heap\_end = heap\_beg) \wedge (i = stack\_beg) \wedge (stack\_end = stack\_beg - 4)$, while $(read(mem[i]) < 0 \ldots 100)$ is the verification condition for the assertion (a:8). The assertion holds for any variable assignment valid on the current path if the conjunction of the formulas is unsatisfiable. Fig. 2(b), taken from [1], illustrates the need for a bit-precise memory-model: both conditions (b:3,5) must evaluate to $true$ to reach the failing assertion (b:6). When reasoning about unbounded integers the assertion is unreachable due to unsatisfiable path conditions. SmacC infers overflow for the multiplication and thus a satisfiable path condition guarding the failing assertion, therefore the failing assertion is reachable.

Experiments. We analyzed a *memcopy* and a *stringcopy* implementation for bounded runtime-and memory-safety (with bounded array-size 50, respectively 40), verified the functional correctness of a *palindrome check* and checked equality of two *power-of-3 implementations*. Path-wise verification of the *memcopy* implementation up to bound 50 takes approximately two hours. Functional correctness for the *palindrome check* (bounded by word length 16) exhibits high run-times (4.5h), and complete equality checking of two *power-of-3 implementations* (with 32bit int) times out (10h). Varying

the bound of the input problems and dumping a conjunction of the verification conditions thus allows to generate SMT benchmarks with varying runtime.

We also integrated the memory-model of SmacC in r-TuBound and extended verification conditions to express arithmetic properties about conditional updates to the loop counter. This allows us to compute loop bounds in cases where the loop bound computation step of r-TuBound would fail. For example, the loop counter i in Fig. 2(a) is conditionally updated, therefore no safe loop bound can be computed initially. Verifying that the conditional update can never decrease the loop counter allows us to use the constant increment in the loop header to compute a safe over-approximation. For the conditional update $i' = upd(i)$, e.g. `i = i + 1` in Fig.2(a), (a:7), we verify that executing it can only increase the loop counter for the next iteration i', i.e. $i' < i$ must be unsatisfiable for arbitrary values of i, as for example in Fig. 2(a) where a loop bound of 4 can be computed using the update `i++` (a:5) in the loop header.

Fig. 2(a) illustrates another usage of SmacC for loop bound detection. Here, SmacC is called with an initial loop bound. If it reports that the negation of the loop condition is satisfiable along a path, the bound is increased. Upon termination, no execution of the program exhibits a higher loop bound. The loop counter i in Fig. 2(c) is reset in iteration 5 (c:5), therefore the loop is executed 4 more times. SmacC infers the exact loop bound 9, while a WCET analysis using the model checker CBMC [3] without SmacC does not terminate and keeps unwinding the loop.

4 Conclusion

SmacC has successfully been used in a number of applications, ranging from program verification to high-level WCET analysis. A key feature of SmacC is its bit-precise symbolic execution which enables it to find a number of typical and important program errors and to functionally verify programs via assertions. Verification conditions that exhibit high solving time can be dumped and used as regression and performance tests for SMT solvers. High-level WCET analysis turned out to be a another promising application field of SmacC and we successfully retargeted SmacC and the underlying memory-model towards integration into a WCET analysis toolchain, improving high-level analysis results. Currently, we are working on implementing the memory-model for binaries and extending SmacC with generation of test-inputs guiding actual program executions towards the WCET path. To improve the runtime of SmacC, we also investigate techniques shown effective for symbolic execution, such as query caching.

References

1. N. Bjørner, L. de Moura, and N. Tillmann. Satisfiability Modulo Bit-precise Theories for Program Exploration. In Proc. of CFV, 2008.
2. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In Proc. of TACAS, pages 174–177, 2009.
3. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In Proc. of TACAS, pages 168–176, 2004.
4. Christopher Fraser and David Hanson. lcc, A Retargetable C Compiler for ANSI C. The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1995.
5. J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis. In Proc. of LPAR, pages 435 – 444, 2012.
6. J. Zwirchmayr. A Satisfiability Modulo Theories Memory-Model and Assertion Checker for C. Master's thesis, JKU Linz, Austria, 2009.