A Satisfiability Modulo Theories Memory-Model and Assertion Checker for C

Jakob Zwirchmayr

Institute of Computer Languages (E185) - Compilers and Languages Group (E185-1)
Vienna University of Technology, jakob@complang.tuwien.ac.at

Abstract. This paper presents SmacC, a tool for software verification and SMT benchmark generation. It builds upon a state-of-the-art SMT solver, Boolector, developed at FMV institute at JKU, Linz. SmacC gets as input a program that lies in the supported subset of C and transforms it to SMT formulas. The SMT representation allows verification of properties that are required to hold on the program. SmacC symbolically executes the programs source code, establishing an SMT (memory-) model for the program. Some statements and expressions require the construction of SMT formulas specifying properties about them, the SMT solver decides their satisfiability. If properties checked do not hold on the SMT representation, they do not hold on the real program. SmacC can generate SMT benchmarks by dumping the SMT instances for those checks.

1 Introduction

SmacC symbolically executes a C program in order to find defects or to create benchmarks to be replayed by an SMT solver. A program consists of a set of instructions and some memory storing instructions and data of the program. When the program is executed, instructions are fetched from memory and then executed by the CPU, repeatedly, in some cases altering data in memory. When the program is symbolically executed by SmacC, instructions are extracted from the source and stored in abstract syntax trees (ASTs), organised in a code-list (CL). The CL is then analyzed, extracting paths through the program. A Boolector array variable models the memory of the program. Execution of a path establishes constraints on the array, reflecting valid memory. Additionally, some statements executed can be checked for defects by constructing an SMT formula representing an error condition and checking its satisfiability using the SMT solver.

The front-end of SmacC consists of an input buffer, a lexer and a parser that parses the source code into ASTs and CLs, respectively. The CLs, containing syntax trees, represent paths through the program. Code-lists are the connection to the back-end that symbolically executes them one ofter another, establishing a memory-model in SMT for each path through the program. Loops are handled by loop-unrolling which transforms loops to sequential if statements. When execution of the program might branch, CLs for each branch are generated.

Checks can be dumped to a file as BTOR or SMT-LIB formula to be used as benchmark for an SMT solver. A check is a BTOR formula that must be satisfiable or unsatisfiable on the SMT representation depending on the statement or expression that triggered it. One can differ between two kinds of checks, verification checks:

- Assertion statement: verify that assertion statement cannot fail,
- Return statements: check if the program returns a specified value in all cases or check if a specified return value is possible,
- Path conditions: check if an if / else condition is unsatisfiable

and defect checks:

- Assignment: checks validity of address a value is assigned to,
- Indirection: checks validity of address being dereferenced,
- Division by zero: checks if division by zero is possible,
- Overflow: checks for overflow on arithmetic operations,

2 Boolector and BTOR Format

BTOR was developed initially as native format for SMT solver Boolector, supporting the theory of bit-vectors and the theory of one-dimensional arrays, as supported by SMT solver Boolector. In addition it supports an extension that can be used for model checking [1].
The SMT solver Boolector was developed at the Institute for Formal Models and Verification of the Johannes Kepler University and is an efficient SMT solver for the combination of the quantifier-free fragment of the theory of bit-vectors and extensional theory of arrays and equality. The quantifier-free theory of bit-vectors enables Boolector to solve formulas including modular arithmetic, comparison, two's complement, logical operations, shifting, concatenation and bit-extraction. The theory of arrays allows natural modelling of memory. Fig. 1 shows the basic usage of Boolector in its stand-alone version.

3 Front-End and supported C Subset

Programs supplied to SmacC must compile with an ANSI C compatible compiler, erroneous programs cannot be handled. Gcc was used as compiler to build SmacC and to compile C examples against which the behaviour of SmacC was tested. In general, a program supplied to SmacC should compile with gcc without warnings, with extra warning flags enabled. The following listing summarizes supported constructs:

- A valid translation unit may only contain global variable declarations of the supported types and one function declaration
- `if-else`, `for`, `assert`, `malloc`, `free`, `sizeof`, `return`, `#include`

```
$> cat example.btor
1 array 8 32
2 var 32 index
3 const 8 00000000
4 write 8 32 1 2 3
5 eq 1 1 4
6 root 1 5
$>
```

```
$> boolector example.btor -m -d
sat
index 0
1[0] 0
$>
```

Fig. 1. BTOR file `example.btor` and output of invoking Boolector. Boolector prints a (partial) model in the SAT case when supplying `-m`, while `-d` enables decimal output. In line 1 an array with element width 8 bit and index width 32 bit is constructed. Line 2 declares a 32 bit bit-vector variable named `index`. Line 3 declares an 8 bit bit-vector constant with value 0 that is written to array 1 on position index (2) in line 4, constructing a new array. Line 5 states that array 1 is equal to array 4. Line 6 sets line 5 as root node such that the formula can be checked with Boolector stand-alone version. Boolector returns 'satisfiable' because it is possible that the element at index `index` of array 4 has the same value as the element at index `index` in array 1.

– Non-augmented assignment statements, compound statement, valid C expressions (some restrictions)

The front-end gets as input a C file that contains a translation unit which lies in the supported subset of C. The lexer tokenizes the input stream and the parser creates ASTs according to the expression grammar, organizing them as statement elements in a CL.

4 Back-End

The back-end gets as its input the full CL that was generated by parsing the translation-unit. It extracts and executes paths through the program symbolically by writing to and reading from the BTOR array representing the memory of the program. It generates SMT formulas for the memory layout and checks satisfiability of properties that must hold. Symbolic execution is split into two phases called path-generation (`pathgen`) and BTOR-generation (`btorgen`). Phase one, path-generation, flattens the full CL by unrolling loops up to a certain bound. After flattening path-generation processes the CL, generating a new CL until meeting an element that represents a branching point in the program. When a branching point is met, the CL is duplicated and both paths are processed further. When a path through the program was extracted, BTOR-generation is responsible for the generation of SMT formulas representing the state of the memory of the program. Some elements in the path require construction of SMT formulas to check for certain programming errors. SmacC can also be configured to dump them in both SMT-LIB or BTOR format.

4.1 Path-Generation

Path-generation phase flattens the CL by unrolling iteration-statements to nested sequences of selection-statements. It can be configured up to which bound SmacC unrolls `for` loops. The resulting flat CL is further processed in the path-generation phase, creating separate CLs for branches through the program. When an element in the flattened CL is of kind selection-statement and execution could branch, the CL representing the path through the program up until this point is duplicated and path-generation is called for both branches, generating a CLs for each of them. When a path through the program is fully extracted either after reaching the last element of the input CL or by processing a return-statement element `pathgen` calls `btorgen` which then symbolically executes the path.

```
                    path 0:                 path 1:
                   CSENTER @ (1,10)        CSENTER @ (1,10)
int main ()          CSENTER @ (1,12)        CSENTER @ (1,12)
{                     CDECLL @ (2,8)          CDECLL @ (2,8)
  int cond;           CIF @ (4,0)             CELSE @ (5,0)
  if (cond)            CBBEG @ (4,0)          CRET @ (5,8)
    return cond;        CRET @ (4,11)         CSEXIT @ (6,0)
  return 0;            CBEND @ (5,0)        CSEXIT @ (6,0)
}                     CRET @ (5,8)
                     CSEXIT @ (6,0)
                    CSEXIT @ (6,0)
```

Fig. 2. Translation-unit and CLs for both paths through the program.

4.2 Btor-Generation

Btor-Generation constructs BTOR expressions for C statements and expressions resulting in SMT formulas. Additionally constraints for the array modelling the programs memory are generated. If an entry in the CL contains an AST representing C expressions the tree is transformed to BTOR expressions by calling `btorgen_generate`. Some entries lead to (verification- or defect-) checks, usually resulting in one or more SAT-checks by Boolector. Variable declarations require the construction of Boolector variables, stored with the symbols and used as addresses for the memory array. When an identifier is parsed in an expression the Boolector variable for the symbol can be looked up in the AST node for the expression. Variable declarations in the code also require updates to the SMT formula representing constraints on the programs memory.

4.3 Memory Model

The memory model is inspired by the memory model usually used in UNIX systems. It is established by an SMT formula that constrains the array variable

```
int
main ()
{
  return 0;
}
```

```
 2 const 32 000...000
 4 var 32 stack_beg
 5 var 32 global_beg
 6 var 32 heap_beg
 7 eq 1 2 2
 8 ult 1 5 5
 9 ult 1 4 4
10 ult 1 6 6
11 ult 1 5 6
12 ult 1 6 4
13 and 1 7 -8
14 and 1 13 -9
15 and 1 14 -10
16 and 1 15 11
17 and 1 16 12
18 root 1 17
```

Fig. 3. A C Program and the BTOR instance for the return statement. The BTOR instance for the return statement `return 0;` is depicted on the right and will briefly be explained: line 2 represents the constant 0, line 4, 5 and 6 represent the BTOR variables necessary to construct the memory model. The BTOR formula for the return statement is constructed in line 7. The rest of the lines form the constraints for the memory layout and are conjuncted with line 7 and selected as root in line 18. Lines 8 to 10 are used negated in line 13 to 15 to formulate the properties that the end of stack, global and heap area must be greater or equal to the beginning of stack, global and heap area. Initially the addresses that represent the end of the memory areas are equal to the addresses that represent the begin of the memory areas. Line 10 and 11 establish the general memory layout which requires that the highest global address is smaller than the lowest heap address which is smaller than the last lowest stack address. Line 17 is the conjunction of the properties mentioned and the formula specifying the return value to be equal to zero.

which models the memory of the program. This allows to check whether memory accesses in the program are valid. If a memory access is invalid for the SMT representation it is also invalid for the real program. The UNIX memory model divides memory for a process into three segments [6]:

- Text Segment: machine instructions, executable code
- Global / Data Segment: global variables, constant strings, but also dynamic memory
- Stack Segment: local variables, parameter variables, grows from high address to low address

SmacC simplifies the UNIX memory model, there is no text segment, the data segment is called global area and is only used for global variables. Memory that is allocated in the data segment by calls to `malloc` is modeled by the heap area. The left-hand side of Fig. 4 is a visualization of the memory layout right after initialization, no variables declared, represented by the following formula:

$$\begin{aligned} &\mathit{global\_beg} \leq \mathit{global\_end} \wedge \mathit{global\_end} < \mathit{heap\_beg} \wedge \mathit{heap\_beg} \leq \mathit{heap\_end} \wedge \\ &\mathit{heap\_end} < \mathit{stack\_end} \wedge \mathit{stack\_end} \leq \mathit{stack\_beg} \wedge \mathit{global\_beg} = \mathit{global\_end} \wedge \\ &\mathit{stack\_beg} = \mathit{stack\_end} \wedge \mathit{heap\_beg} = \mathit{heap\_end} \end{aligned}$$

When variables are declared or dynamic memory is allocated the memory-model needs to be updated to include constraints about the variable. Consider the

right-hand side of Fig. 4, visualizing the memory model after a few variables were declared, represented by the following updates to the memory model:

$$i = global\_beg \wedge j = global\_beg + 4 \wedge global\_end = global\_beg + 8$$
$$heap\_v1 = heap\_beg \wedge heap\_end = heap\_beg + 4$$
$$p = stack\_beg - 4 \wedge c = stack\_beg - 4 - (4 * 1) \wedge stack\_end = stack\_beg - 8$$

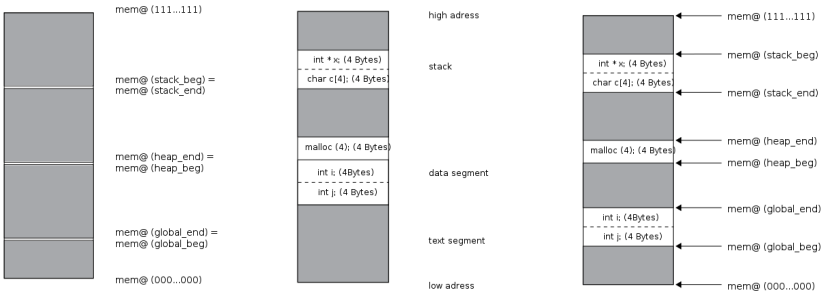

Fig. 4. Simplified view of the UNIX memory-model of a C program and its representation in SmacC. In the left example no variables are declared. In the right example the program has integers `i` and `j` declared as global variables, integer pointer `x` and character array `c` as local variables and 4 bytes allocated on the heap by a call to `malloc`.

SmacC considers the following memory accesses invalid, for the sake of brevity only the first is discussed in this paper.

- Access out of valid memory: an access is considered out of valid memory if it accesses indices that are not indices representing stack area, global area or heap area. Invalid regions are marked grey in Fig. 4.
- Access out-of-bounds: an access is considered out-of bounds if it crosses boundaries of data elements, for example when data from two valid regions is read or written. Out of bounds access can happen at all addresses.

5 Checks

While a path is symbolically executed certain statements and expressions lead to checks. A check is an SMT formula that must be SAT or UNSAT when added to the formulas of the memory-model and checked via Boolector. SmacC checks include those that verify that a memory access is valid in the memory's SMT representation and hence valid in the C program. Furthermore they are used to verify assertions, show that an operation does not lead to an error or show that a path condition cannot be satisfied. The assertion check and the basic memory check are presented.

5.1 Assertion Check

Variations of assertion checks are used to verify program return values and to check for division by zero. Consider the example in Fig. 5.

```
void main ()
{
  int i;
  assert (i);
}
```

btor_vars =
{
$global\_beg, global\_end,$
$heap\_beg, heap\_end,$
$stack\_beg, stack\_end,$
mem, i
}

layout := $global\_beg \leq global\_end \wedge$
$global\_end < heap\_beg \wedge$
$heap\_beg \leq heap\_end \wedge$
$heap\_end < stack\_end \wedge$
$stack\_end \leq stack\_beg \wedge$
$global\_beg = global\_end \wedge$
$heap\_beg = heap\_end \wedge$
$i = stack\_beg - 4 \wedge$
$stack\_end = stack\_beg - 4$

assert := $read(mem, i) = 00000000 \wedge$
$read(mem, i + 1) = 00000000 \wedge$
$read(mem, i + 2) = 00000000 \wedge$
$read(mem, i + 3) = 00000000 \wedge$

Fig. 5. On the left: assertion statement in a C program and declared Boolector variables. On the right: assumptions about memory layout and the formula representing the assertion.

The conjunction of formulas **layout** $\wedge$ **assert** must be unsatisfiable, otherwise the assertion might fail.

5.2 Basic Memory Check: Arbitrary-but-Fixed

The basic memory check constructs a Boolector bit-vector variable *abf* and uses the SMT formulas for the general memory layout to let *abf* point to an arbitrary address in memory but it is fixed to be outside any valid memory. Then it is checked if the variable *abf* can be equal to the address *addr* being read from or written to. If it is satisfiable that *addr* is equal to *abf* it is shown that the memory access could address an illegal memory address (outside any known memory region, or in a region that was freed by `free`). SmacC checks both the first and last byte of a value being read or written from or to memory. A problem of the basic memory check is that the results of the check can depend on the order in which variables were declared. This effect can also occur in C programs and is hard to capture. The formulas constructed for the check are presented in Fig. 6.

abf_invalid :=
$abf > stack\_beg \wedge$
$abf > global\_end \wedge$
$abf < heap\_beg \wedge$
$abf > heap\_end \wedge$
$abf < stack\_end \wedge$
$abf < global\_beg$

abf_freed :=
$abf \geq free\_var_i \wedge$
$abf < free\_var\_i + free\_var_i\_size$

check :=
$abf = addr$

Fig. 6. Basic Memory Check: constraining a variable to be outside valid memory or equal to a freed address.

Clearly, because of the constraints on *abf*, if the SMT formula (**abf_invalid** $\vee$ **abf_freed**) $\wedge$ **check** is satisfiable for any byte of *addr*, then invalid memory is accessed.

6 Limits of the Model

The array memory check (not treated in this paper) has the weaknesses that expressions using pointer arithmetic can fool the array (out-of-bounds) memory check, nevertheless it can be used to verify some pointer arithmetic expressions. If memory allocated by `malloc` is deallocated by `free` it is not used again in following calls to `malloc`. This can lead to out-of-memory situations where `malloc` cannot allocate requested memory, leading to a contradiction in the memory model and hence invalidating reported results. This could even occur if memory deallocated by `free` was reused. If a program allocates all available memory by a call to `malloc` and then allocates additional (unavailable) memory, the assumptions used to construct the memory model can be contradicting, invalidating results of checks following the second call to `malloc`. Assume that the first call to `malloc` allocates all memory from the lowest address to the highest address. Assumptions established for the memory model are (omitting assumptions for global and local memory regions): SmacC assumes that there is no overflow on address calculations. Because of the assumption that the first call to `malloc` forces *heap_end* to be equal to the highest address, overflow is unavoidable for address calculation of *m2*, a contradiction follows. Another problem emerges from the way path conditions of loops are handled: after unrolling the loop up to the specified bound the loop condition is assumed to be true. If it is the case that the state of the memory contradicts the assumption, then the checks following the loop return wrong results.

7 Related Work

CBMC is a Bounded Model Checker for ANSI C and C++ programs. It allows verifying array bounds, pointer safety, exceptions and user-specified assertions [5]. CBMC takes as input C files and translates the program, merging function definitions from the input files. Instead of producing a binary for execution,

CBMC performs symbolic simulation on the program [4]. CBMC translates refined programs to SAT instances and uses MiniSAT to verify properties.
Recently, preliminary support for SMT solvers (Boolector, CVC3, Yices, and Z3) has been added via the SMT-LIB theory QF_AUFBV [5].
CBMC can also be used to check behavioral consistency of C and Verilog programs (Hardware and Software Equivalence and Co-Verification) [3].
The major difference to SmacC is that CBMC does not establish a full representation for the memory of the program and its layout, instead it uses intermediate variables when accessing variables. CBMC unwinds loops and recursive function calls and transforms the program until it only consists of `if` instructions, assignments, assertions, labels and `goto` instructions [2]. An assertion for each loop verifies that the unwinding bound [2] is large enough, otherwise the bound is increased. Then it is transformed into static single assignment form, consisting of bit-vector equations for constraints and verification conditions. The conjunction of the constraints and the negation of the property is checked for satisfiability. If the conjunction is satisfiable, the property is violated.

8 Benchmarks

The following C files and algorithms were transformed to a BTOR representation, and can be used as benchmarks, timing results are presented in Tab. 1.

- Memcopy: A simple memcpy implementation, copying memory from the source buffer to the destination buffer. Assert that destination buffer contains the same elements as the source buffer after copying.
- Palindrome: implements algorithm to check if a string is a palindrome. If the algorithm concludes that a string is a palindrome, assert that the string fulfills palindrome properties.
- Stringcopy: Similar to memcpy but omitting the third parameter, the number of bytes that must be copied. The loop terminates if null character is read in source buffer which is then copied to the target buffer.
- Power of 3 equality: Compares if a method to compute n^3 using a loop always yields the same result as a method without a loop.

Benchmark	**Bound**	**Boolector**	**SmacC**	**CBMC**
memcpy.c, array size 30	30	287s	1496s	0.25s
memcpy.c, array size 40	40	565s	5595s	0.33s
memcpy.c, array size 50	50	1114s	7350s	0.34s
palindrome check, n 11	11	639s	3718s	0.18s
palindrome check, n 15	15	1614s	13406s	0.22s
palindrome check, n 16	16	3344s	16220s	0.26s
strcpy array, n 20	20	231s	timeout	0.11s
strcpy array, n 30	30	1430s	timeout	0.15
strcpy array, n 40	40	7684s	timeout	0.20s
power 3 equality	3	timeout	timeout	timeout

Table 1. Benchmarks were run on an Intel®CPU at 2.66GHz with 2GB main memory. Time was measured using the UNIX *time* command. The table compares Boolector stand-alone version to library usage in SmacC and to CBMC.

References

1. Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Lecture Notes in Computer Science (LNCS)*, volume 5505. Springer, 2009. TACAS'09.
2. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. Carnegie Mellon University, 2004.
3. Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. 2003.
4. CProver. The cprover user manual. Available at http://www.cprover.org/cbmc/doc/manual.pdf.
5. Daniel Kroening. Bounded model checking for ansi-c. Available at http://www.cprover.org/cbmc/.
6. Andrew S. Tannenbaum. *Modern Operating Systems, 3rd Edition*. Pearson, Prentice Hall, Upper Saddle River, New Jersey 07458, 2007.