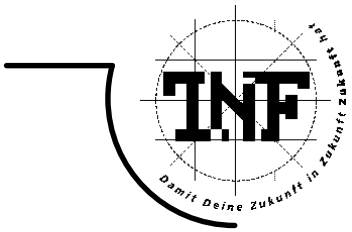
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



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

Masterarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

in der Studienrichtung

Informatik

Angefertigt am *Institute for Formal Models and Verification*

Betreuung:

Univ-Prof. Dr. Armin Biere
Dipl.-Ing. Robert Brummayer

Eingereicht von:

Jakob Zaches Zwirchmayr

Linz, Oktober, 2009

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 11. Juni 2010

Jakob Zaches Zwirchmayr

Abstract

The thesis presents the tool SmacC, an approach to software verification and SMT benchmark generation building upon a new state-of-the-art SMT solver, Boolector, developed at FMV institute at JKU, Linz.
The program gets as input a C program that lies in the supported subset of the programming language (ANSI) C and transforms the program to SMT formulas. The SMT representation allows verification of properties that must hold on the program and the generation of SMT benchmarks by dumping the SMT instances.
Part of the goal of this work includes, on the software verification side, to check the input code for certain programming errors and additionally prove or disprove assertion statements in the code.
The other goal of the work was to generate SMT examples for SMT solvers that can be used either as (regression) tests for newly developed SMT solvers, or as benchmarks to compare the performance of different SMT solvers that support the underlying formats (BTOR or SMT-LIB) and theories (bit-vectors, arrays, equality of arrays).
To reach the goals, the tool symbolically executes the programs source code, establishing a (memory-) model for the program, represented as SMT formulas. Then the tool generates SMT formulas and lets the SMT solver decide if certain properties hold on the SMT representation of the program. If properties checked do not hold on the SMT representation, they do not hold on the real program.

Keywords: *SMT, Satisfiability Modulo Theories, Boolector, BTOR, Assertion Proving, Memory Model, Programming Language C, Symbolic Execution, Symbolic Simulation, Benchmarks.*

Contents

Chapter 1

Introduction

1.1 Motivation

The thesis builds upon the SMT solver Boolector that was developed at the Institute for Formal Models and Verification (FMV) at Johannes Kepler University (JKU) in Linz, Austria.
One of the goals of logic in computer science is to develop languages to model situations encountered when designing hardware or software, that allow reasoning about them in an appropriate way [11].
Boolector is a new kind of verification engine, a Satisfiability Modulo Theories (SMT) engine, that received a lot of interest in both research and industry recently.
SMT generalizes pure boolean satisfiability (SAT) and provides first-order theories to express design and verification conditions of interest [5].
SMT solvers are programs that deduce the satisfiability of formulas with respect to certain background theories (also called background logics).
When the SMT solver Boolector was still in its early days of development benchmarks for extensive testing were essential.
An earlier tool (CVC2BAF [25]) aimed at the translation of benchmarks from SMT-LIB and CVC input format to Boolector Ascii Format (BAF), the predecessor of a format now known as BTOR, to enable the developers of Boolector to use existing benchmarks for testing.
The term benchmarks denotes a logical formula to be checked for satisfiability with respect to (maybe combinations of) background theories of interest.
Examples of background theories typically used in computer science include the theory of real numbers, the theory of integer numbers, and theories of various data structures such as lists, arrays and bit vectors [20].

The goal of SmacC is similar to the goal of CVC2BAF. But, instead of translating benchmarks from other SMT to BTOR , a subset of the programming language C is transformed to BTOR format.
For SMT benchmarks in other input formats translation to BTOR format is straightforward because the semantics of both the source format (for example SMT-LIB format) and the destination format, BTOR, are well defined by the SMT-LIB standard and the definition of the BTOR format.
BTOR is a quantifier-free word level format for formulas over bit-vectors in combination with one-dimensional arrays and allows the modeling of SMT problems [6], as does the SMT-LIB format.
When using (a subset of) the programming language C as input format, translation becomes more complicated: in the context of a C program, translation to BTOR format cannot be done in a straightforward approach because the semantics of BTOR and C differ in many ways.
In this case it is not just a translation of an SMT formula from one format to another, with corresponding operators existing in both formats, but it requires establishing an SMT model of the input source that captures the essence of the program and allows formulating properties of the program as SMT formulas that can then be checked for satisfiability by Boolector.
When transforming a C program to an SMT formula the following aspects have to be considered to establish an SMT model representing the program:

- when executed on a processor, a C program modifies the computer memory - values are read and written from and to memory.
- a C program, when executed, may branch on various points in the program.
- a C program is more than the check for satisfiability of a formula. Multiple SMT formulas must be constructed and checked to capture the program.

Considering the aspects mentioned above it becomes clear that the translation from C source code to BTOR format cannot be achieved easily. Viewing it as transformation consisting of multiple phases, rather than translation in one pass, seems more exact.
The techniques applied to transform an input source file to an SMT model capturing the program and SMT formulas specifying properties about the model are similar to those used in compilers and interpreters for programming languages.
The basic idea of the program SmacC is to parse a C program and extract possible paths through it. For each path a memory-model needs to be formulated in SMT.
Expression- and statement-semantics in the code can be checked on the SMT model. For example, assertion statements in the code can be verified to hold.

The name of the tool comes from this basic idea: ”an SMT Memory-model and Assertion-Checker for C”.
SmacC utilizes SMT solver Boolector to establish an SMT memory-model, check properties and dump benchmarks in BTOR or SMT-LIB format.

1.2 Introduction to SmacC

SmacC is written in C and uses Boolector as library. This allows incremental usage of the solver. At the time of writing, the solver is available in binary or library version for Linux platforms, therefore SmacC only supports Linux platforms, too.
The front-end consists of an input buffer, buffering the input programs source code, a lexer, tokenizing the input and a parser that parses the source code into abstract syntax trees (ASTs) and code-lists, respectively.
This compiler-like infrastructure handles only a subset of C (presented later) and is based on parts of the 'lightweight C compiler' (lcc) that is presented in the book 'A Retargetable C Compiler: Design and Implementation' [13]. It is a compiler for ANSI-C, therefore also SmacC considers ANSI-C, with some exceptions, when parsing a C file (the most important exception is that old-style function declarations, still legal in ANSI-C, are not allowed [16]). Additional limitations are described later.
The code-lists, containing syntax trees, represent execution paths through the program. These code-lists are the connection to the back-end of SmacC, that symbolically executes code-lists, establishing a memory-model in SMT for each path through the program.
Using the memory-model different checks can be performed to verify certain properties in the code. These checks can also be dumped to a file as BTOR or SMT-LIB formula to be used as benchmark for an SMT solver.
The goal is to formulate properties that must hold on the real program as SMT formulas and check satisfiability of the property in combination with the SMT representation of the program.
Loops are handled by loop-unrolling, transforming loops to sequential if statements. When execution might branch, code-lists for each branch are generated. The supported C subset does not include floating point numbers and operations, multi-dimensional arrays, storage-class specifiers, typedef, structs and unions or constructs altering program flow, details will be presented later.
SmacC supports various command line flags to enable and disable various checks and also control how detailed and in which way the memory-model being established will be checked for errors.
When an SMT representation of the source code is being established by symbolically executing the program certain statements and expressions lead to checks.
Per default checks (properties formulated in SMT) are performed on the SMT repre-

sentation but SmacC can be set to benchmark creation mode, in which case the checks are not executed but dumped as BTOR or SMT-LIB formulas.
A check is a BTOR formula that must, in some cases, be satisfiable, or unsatisfiable on the SMT representation. Checks include verifying that a memory access is valid in the SMT representation (and hence valid in the real C program), verifying that an assertion holds, showing that an operation does not lead to an error (overflow or division by zero) and showing that a path condition cannot be satisfied.
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
- 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

1.3 Text Overview

In order to check the supplied C code for the mentioned defects or to verify the properties, it is necessary to construct an SMT instance for it.
Before going into more detail it is necessary to discuss SMT, the SMT solver Boolector and the BTOR format in more depth in the next chapter. A short overview then leads to a precise description of the steps that SmacC performs to establish an SMT instance for the C source code.
After presenting results and benchmarks generated with SmacC, a tutorial that explains precisely how to use and control SmacC for error checking and benchmark generation and how to read its output follows in the last chapter.

Chapter 2

Satisfiability Modulo Theories

2.1 Theories

SMT, with its capability to express formulas not only in pure boolean propositional logic but also in various first-order theories, has recently become important both in research and industry. It is used to express designs and verification conditions.
These designs and verification conditions could well be modeled in boolean propositional logic, but the possibility to use constructs from other theories eases the creation of models for designs and the formulation of verification goals.
Theories that extend boolean propositional logic are finite or infinite sets of formulas, characterized by grammatical rules, allowed functions and predicates, and a domain of values [18].
Most SMT solvers consider only quantifier-free fragments of first-order theories. First-order theories of interest for verification include equality logic, linear arithmetic, arrays, bit-vectors, uninterpreted functions and others (detailed grammars, explanations and examples for each theory given in [18]).
The most important theories for this work are:

- Propositional logic
  - True and false as domain
  - Example: $x_1 \wedge (x_2 \vee \neg x_3) \Rightarrow x_4$
- Bit-vectors

  - Domain is the domain of finite bit-vectors, as is in computer systems (semantics of modular arithmetic unlike, for example, unbounded bit-vectors or natural numbers)

  - Semantics of some of the operators are known from programming languages like C

  - Example: $((a >> b)\&c < c)$

- Arrays

  - Expressions over arrays (maps from index types to element types)

  - *read*, *write* and compare array elements

2.2 BTOR Format

BTOR was developed initially as native format for SMT solver Boolector, supporting the theory of bit-vectors and theory of one-dimensional arrays with semantics as described in SMT-LIB standard 1.2 [5].
In addition it supports an extension that can be used for model checking [5].

Overview

In 1998 SATLIB was created as an online repository of benchmark problems and solvers for SAT. The motivation was to facilitate and encourage empirical studies of SAT algorithms that make use of a common set of benchmark instances in order to enhance the comparability of empirical results [15].
During recent years interest in SMT systems has increased, and found application not only in formal verification but also in compiler optimization, scheduling and others [19] The motivation for the creation of the SMT-LIB initiative was to facilitate benchmarks that allow to evaluate and compare improvements of SMT systems. The initiative aims to establish a common standard of benchmarks and of background theories [20].
The format established by SMT-LIB is supported by most state-of-the-art SMT solvers and is used, for instance, at SMT-COMP as input format.

BTOR is a bit-precise word-level format that is easy to parse and has precise semantics. In its basic form BTOR allows modelling of SMT problems over quantifier-free theory

of bit-vectors and one-dimensional arrays [6], as supported by SMT solver Boolector. When using a sequential extension to the basic form it is also possible to express model-checking problems.
In principle, it is a world-level generalization of the AIGER format [2]. It is strongly typed, multi-rooted, and has precise semantics [6].

Consider the following example in BTOR:

```
1 var 32 a
2 constd 32 8
3 constd 32 12
4 saddo 1 1 2
5 saddo 1 2 3
6 and 1 -4 5
7 root 1 6
```

Listing 2.1: A first BTOR example

Column 1 represents a unique non-negative identifier for the BTOR expression. Column 2 represents the type (operator), column 3 the bit-width of the expression.
Expression 1 (line 1) declares a 32-bit bit-vector variable that is identified in other expressions by 1 but also has an optional name field, represented by string "a".
Line 2 declares the decimal constant 8 as bit-vector of width 32, represented in twos complement internally and line 3 declares the decimal constant 12.
The expression in line 4 represents signed addition overflow and exemplifies computation: line 4 is the result of checking for overflow when adding line 1 (the declared bit-vector variable with arbitrary value) and line 2 (the constant 8). The bit-width of *saddo* expression is 1.
Checking overflow when adding line 2 and 3 results in line 5. Line 6 logically *and*s line 4 and 5, negating 4. It formulates the property that there is no overflow when adding the variable to the constant and that there is overflow when adding the two constants.
Line 7 sets formula 6 as boolean root. In order to check the formula with Boolector it is necessary to supply exactly one root, which has to have bit-width 1. Although other tools supporting BTOR format may support more than one root expression.

Bit-vectors

BTOR supports arbitrary bit-widths for bit-vectors [6] and allows the construction of bit-vector variables and constants in decimal, hexadecimal and binary with operators *var*, *constd*, *consth* and *const*.

Constructors for the bit-vector twos complement representations for the numbers 1, -1 and 0 are supported natively. As shown in the example above, operators for constants and variables both take as third argument the bit-vectors width.
The fourth column represents the value of the bit-vector constant and is mandatory. The fourth column for variable expressions represents the name of the variable that can be omitted.
The semantics of bit-vector operators correspond to the semantics defined in SMT-LIB standard for quantifier-free theory of fix-sized bit-vectors (`QF_BV`), with the exception that SMT-LIB standard does not define the result of dividing by zero. The way Boolector handles division is modeled after how hardware circuits treat unsigned division: unsigned division by zero in BTOR results in the largest unsigned integer that can be represented in the operands bit-width.
The bit-width of the first operand of shift operations has to be a power of two. The bit-width of the second operand needs to be log_2 of the bit width of the first operand [6].
In addition to the operators defined in SMT-LIB standard bit-vector operators include: *redand*, *redor* and *redxor* reduction operators from hardware description language Verilog. VHDL rotate operators *rol* and *ror* and a set of overflow detection operators, as presented in the simple BTOR example.

The following tables list all bit-vector operators [6]. Column one lists the class of the operator, column two the name of the operator. The rest of the columns represent the operands bit-widths and the bit-width of the resulting expression, respectively:

class	**operators**	w_1	w_r
negation	not, neg	n	n
reduction	redand, redor, redxor	n	1
arithmetic	inc, dec	n	n

Table 2.1: Table of unary bit-vector operators

Negation operator *not* is negation in one's complement whereas *neg* is negation in two's complement representation.
Reduction operators *redand*, *redor*, *redxor* are known from HDL Verilog. Increment and decrement operators increment respectively decrement a bit-vector by one [1].

Arithmetic, relational, shift and overflow operators can be used in signed or unsigned versions.
Overflow operators correspond to the arithmetic operators, with the exception of *udiv*, as unsigned division cannot overflow. Overflow on signed division happens when dividing the smallest negative integer by -1.

class	**operators**	w_1	w_2	w_r
bitwise	and, or, xor, nand, nor, xnor	n	n	n
boolean	implies, iff	1	1	1
arithmetic	add, sub, mul, urem, srem, udiv, sdiv, smod	n	n	n
relational	eq, ne, ult, slt, ulte, slte, ugt, sgt, ugte, sgte	n	n	1
shift	sll, srl, sra, ror, rol	n	$log_2 n$	n
overflow	uaddo, saddo, usubo, ssubo, umulo smulo, sdivo	n	n	1
concat	concat	n_1	n_2	$n_1 + n_2$

Table 2.2: Table of binary bit-vector operators

Consider the following example from [6]:

```
1 var 32 v1
2 var 32 v2
3 redand 1 1
4 redand 1 2
5 umulo 1 1 2
6 and 1 3 4
7 and 1 6 -5
8 root 1 7
```

Listing 2.2: Unsatisfiable BTOR example

If all bits of bit-vector variable v1 are set to one redand in line 3 returns 1 as result, the same holds for the redand operation in line 4.
Line 6 requires both line 3 and line 4 to return 1 to hold, that is, both redand operations returned 1, hence both variables were set to one on all bits.
Line 5 checks for overflow on unsigned multiplication of variable v1 and v2. Line 7 requires the multiplication not to overflow and that the and-reduction of v1 and v2 both return 1.
The example given above is unsatisfiable because if the left operand (line 6, the expression that requires all bits set to one in both v1 and v2 to result in 1) of the and operator in line 7 evaluates to true, then it is not possible that the negation of the right operand of the and operator in line 7 evaluates to true (if all bits of both v1 and v2 are set to one then unsigned multiplication will overflow).
There are two more bit-vector operators that do not fit in the above categories, one being the conditional expression, the only ternary bit-vector operator that represents a functional if-then-else, as known for example from C-like programming languages. The last bit-vector operator is used for bit-extraction.

class	**operators**	w_1	w_2	w_3	w_r
conditional	cond	1	n	n	n

Table 2.3: Ternary bit-vector operator cond

If the condition holds (evaluates to 1) cond returns the second argument as result of the expression, the third argument otherwise.

class	**operators**	w1	**upper**	**lower**	w_r
extract	slice	n	u	l	$u - l + 1$

Table 2.4: Bit-extraction operator slice, using imemdiates

Slice operator slice takes as first argument a bit-vector expression and returns the bit-vector that results when only considering bits between argument upper and lower.

Arrays

The constructor array allows creation of one-dimensional arrays. The first argument represents the bit-width of array elements, the second argument represents the bit-width of index-elements or addresses. BTOR supports the array operations *read*, *write*, *acond* and *eq*, equality can be applied to both arrays and array elements [6].

Again, consider an example taken from [6]:

```
1 array 32 4
2 array 32 4
3 array 32 4
4 var 4
5 var 32
6 var 1
7 acond 32 4 6 1 2
8 write 32 4 7 4 5
9 read 32 8 4
10 eq 1 5 9
11 eq 1 3 8
12 and 1 10 11
13 root 1 12
```

Listing 2.3: BTOR array example

Lines 1, 2 and 3 declare arrays with elements 32 bits wide and index width of four bits, that is, three one-dimensional arrays that contain 16 elements of 32 bit.

In Line 7 array 1 is returned if condition (line 6) holds, array 2 otherwise, resulting in an array of element-width 32 bit and index-width 4 bit.
A value (line 5) is written to the array resulting from line 7 to an index that is specified by line 4.
The result of this *write* operation is an array, index-width 4 bits and element-width 32 bits. Element at position line 4 contains the value of line 5.
The other elements do not change and are equal to the elements in array 1 or 2 depending on condition 6.
Line 9 reads from the freshly constructed array at position 4. it must be equal to the value that was written to the array (line 5). This property is formulated in line 10.
Line 11 compares the full arrays 1 and 8. The comparison of the arrays (line 11) is anded with the comparison of values (line 10) and asserted in line 12, finally line 13 declares line 12 as root.

Model Checking Extension

The solver Boolector can also be used as incremental model checker for word-level safety properties of synchronous hardware systems with memories using operators *next* and *anext*, defined in BTOR to express state transitions of bit-vector registers and memories, but the concept of these operators is of no importance for this work [6].

2.3 SMT Solver Boolector

The SMT solver Boolector was developed at the Institute for Formal Models and Verification of the Johannes Kepler University and is an efficient state-of-the-art SMT solver for the combination of the quantifier-free fragment of the theory of bit-vectors and extensional theory of arrays and equality. It uses lemmas on demand for the extensional theory of arrays.
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 extensional theory of arrays in combination with the congruence axiom from the theory of uninterpreted functions enables Boolector to reason about arrays. The non-extensional part of the theory of arrays includes constructs to read and write on arrays, *read* and *write*, and to conclude equality on array elements [4].
The congruence axiom in combination with extensionality allows full array comparison [4].

The tool SmacC uses Boolector as underlying SMT solver to solve formulas that are constructed after parsing the supplied C code, while symbolically executing the SMT representation of C code.
Boolector can be used as a stand-alone solver taking a BTOR file as input, but also supports input supplied as file in SMT-LIB standard (`QF\_BV`, `QF_AUFLIA`)
Besides using Boolector as a stand-alone solver the library version of Boolector allows the usage of the solver in programs built around it incrementally.
When using Boolector as a library the user is able to construct an SMT formula on-the-fly and check its satisfiability as necessary (details are presented later in the work).
SmacC uses the Boolector library version, which allows formulation of check goals while symbolically executing the C code that was given as input.
SmacC is able to dump satisfiability checks in BTOR and SMT format, which allows usage of those dumps as benchmarks for the standalone version of Boolector and other solvers supporting BTOR or SMT format, respectively.
The following listings show the basic usage of Boolector in its stand-alone version and as library.

```
$> 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
$>
```

Listing 2.4: BTOR file example.btor

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

Listing 2.5: Usage of Boolector binary with input file in BTOR format

Boolector prints a (partial) model in the SAT case when supplying `-m`, `-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.

```
#include "../../include/boolector.h"
int main (void)
{
  Btor * btor;
  BtorExp * mem, * mem_1, * cnst, * var;
  BtorExp * read_1, * read_2, * formula;
  unsigned result = 0;

  btor = boolector_new ();
  boolector_enable_model_gen (btor);
  boolector_enable_inc_usage (btor);

  mem = boolector_array (btor, 8, 32, "mem");
  var = boolector_var (btor, 32, "index");
  cnst = boolector_int (btor, 0, 8);
  mem_1 = boolector_write (btor, mem, var, cnst);
  read_1 = boolector_read (btor, mem, var);
  read_2 = boolector_read (btor, mem_1, var);
  formula = boolector_ne (btor, read_1, read_2);

  boolector_assume (btor, formula);
  result = boolector_sat (btor);

  boolector_dump_btor (btor, stdout, formula);

  boolector_release (btor, mem);
  boolector_release (btor, var);
  // ...
  boolector_delete (btor);

  return result;
}
```

Listing 2.6: Boolector library usage

Listing 2.6 shows how the BTOR formula in the last listing can be generated by using Boolector as library in a C program.
Most of the `boolector_` function calls correspond exactly to the operators described in the BTOR format section. Instead of identifying expressions by line numbers `BtorExp` pointer variables are used.

- `boolector_new`: Creates a new instance of Boolector.

- `boolector_enable_model_gen`: Enables model generation, producing a model for the formula in the satisfying case.

- `boolector_enable_inc_usage`: Enables incremental usage of Boolector. Incremental usage allows to add assumptions and check satisfiability multiple times.

- `boolector_sat`: Solves SAT instance represented by assumptions and assertions combined by boolean *AND* ($\wedge$). Incremental usage has to be enabled to solve more than one SAT instance.

- `boolector_assert`: Adds a constraint to SAT instance.

- `boolector_assume`: Adds an assumption when incremental usage is enabled. Assumptions are discarded after each call to `boolector_sat`, in contrast to constraints. As in MiniSAT, assumptions and assertions can be used to check constraint satisfiability of the instance. A number of clauses will be treated as assumptions. The assumptions will be temporary asserted during solving the SAT problem [12]. Details can be found in [12] and [22].

- `boolector_dump_btor`: Allows to dump an expression in BTOR format, though SMT-LIB 1.2 is also supported.

Chapter 3

Overview

SmacC symbolically executes a C program in order to find defects in it or to create benchmarks. Prior to discussing how symbolic execution can simulate the execution of a program on a CPU, it is necessary to present some relevant definitions of a program and discuss execution of a program on a CPU.
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.
Processor instructions usually fall in one of the following four categories [23]:

- Processor-Memory: Data is fetched from memory or written to memory by the CPU.
- Processor-IO: Data is transfered between CPU and peripheral device.
- Data Processing: CPU performs logical or arithmetic operation on data.
- Control: Control flow is altered by changing the instruction to be executed by the CPU next.

When the program is symbolically executed by SmacC, instructions are extracted from the source and stored in abstract syntax trees, organised in a code-list. Instead of executing them on the CPU an SMT instance is constructed.
A Boolector array variable represents the memory of the program, it is modeled byte-wise as an one-dimensional array with index width 32 bit. If instructions modify data, they modify values of the array. In SmacC, in contrast to the execution of a program on a CPU, instructions are not stored in the memory array.
One must note that to execute a statement or expression in the programming language

usually involves executing multiple CPU instructions on the processor. The same holds for SmacC, a statement or expression in the programs source code usually involves multiple Boolector expressions that need to be executed.
The code-list, containing instructions of the program, is then analyzed, extracting paths through the program. After a path was identified it is executed. Execution of a path through the programs representation establishes constraints on the array representing memory (the memory model) as SMT formula.
Additionally, certain statements executed can be checked for defects by constructing an SMT formula representing an error condition and checking its satisfiability.

CPU instructions	**SmacC representation**
Processor-Memory	`boolector_write` and `boolector_read` on Boolector memory array
Processor-IO	IO is not supported by SmacC
Data Processing	`boolector_` logical and arithmetic operations
Control	Only a few instructions altering control flow are supported

Table 3.1: CPU instructions and SmacC representation

3.1 Symbolic Execution and Data

Symbolic executions means to run a program in the absence of concrete input data.
An argument supplied to a program running on a CPU has a concrete value: if the argument is, for example, of type `unsigned char` then it has a value between 0 and 255. If the program is symbolically executed, the argument to the program is constraint to be between 0 and 255, but it does not have a concrete value. When reasoning about a statement or expression containing the symbolic argument it is necessary to consider all 256 values.
Assume that `unsigned char j` is an argument to a program and consider expression `(i / j)`. When the expression is executed on a CPU, variable `j` has a concrete value that was supplied to the program. It would require between 1 to 256 runs of the program, supplying different values for `j` each run, to deduce that division by zero might occur when executing the program. If the program is symbolically executed, using symbolic data for `j`, only one run is necessary to deduce that division by zero is possible.
SmacC uses symbolic data for arguments to functions, uninitialized memory regions and to model addresses of variables.
The next program points out the benefit of using symbolic data to model uninitialized memory:

```
#include <assert.h>
int main ()
{
  unsigned char c;
  assert (c < 254);
}
```

`c` is an uninitialized memory region. If the program is compiled and run a few times the assertion holds in most cases. Nonetheless the value of `c2` is assigned non-deterministically and can be equal to 255, hence the assertion can fail. Executing the program symbolically detects the flaw in one run.
Other sources for non-determinism in programs are addresses of variables. Prior to running a program, it is not determined what concrete address identifies the memory location of a variable:

```
int main ()
{
  unsigned char c1, c2, c3;
  c3 = ((unsigned) &c1) + ((unsigned) &c2);
  return c3;
}
```

The program can return a different value each time it is executed. To model this behaviour SmacC uses symbolic addresses to access memory. A symbolic variable (*stack_beg*) identifies the address of the local variable that is defined first in function `main` (`c1`). Variable `c2` identifies the memory region at symbolic address $stack\_beg + 1$ and `c3` identifies the memory region at symbolic address $stack\_beg + 2$. The value assigned to `c3` contains symbolic variable *stack_beg*, hence the result is symbolic. When executing statement `return c3;` a non-deterministic value is returned by SmacC.
The benefit of symbolic execution is that it is possible to detect defects, usually very hard to detect via traditional testing, in one run. An example for algorithms where exhaustive testing is difficult are cryptographic algorithms and equality checking: verifying the en- and decoding procedure for all possible messages (of a certain length) can be nearly impossible. Using symbolic data for messages allows verification for all possible messages.
Another field symbolic execution achieves good results in is equality checking: verifying that there exist no input values for which two versions of an algorithm (complying the same specification) return different values.

3.2 Path Extraction

When a program is executed on a processor control flow can be altered to jump backwards or forward in the program. SmacC only allows a subset of instructions that use forward jumps.
During path extraction the program is represented as a forest of syntax trees containing instructions and operands, organized in a code-list, holding instructions in the same order as does the source code. The code-list is iterated through recursively, when finding an instruction that alters control flow, the code-list is duplicated and both branches at the branching point are processed further.

3.3 Memory Model

The memory used by the program is modeled as Boolector array expression. When instructions in the source of the program modify or read data from memory, the corresponding action is performed on the memory model of the program via `boolector_write` and `boolector_read` operations, respectively.
When a variable is declared in the source code it can be used to address a specified memory location. When SmacC symbolically executes a variable declaration a constraint Boolector variable is constructed and used to address the memory array. This enables SmacC to let Boolector not only modify or read values from addresses, as does (the binary) of the program, but also check the validity of an operation on a memory address.

3.4 Assertion Checking

Verifying an assertion in the source code is only a matter of transforming the statement to an SMT formula and checking its unsatisfiability in combination with the memory model established by symbolically executing the source up to the assertion statement. Using the same technique it is also possible to verify that a program returns a specified return value, or verify that a specified return value is possible. When symbolically executing an instruction that would alter control flow of the program, the satisfiability of the condition guarding it is checked.
Because Boolector supports operations for overflow detection it is possible to verify that an arithmetic expression cannot overflow, or that division by zero cannot occur in expressions. The memory model allows to check validity of addresses to which values are assigned to in assignment statements and validity of addresses being read from.

Chapter 4

SmacC

SmacC was written in the programming language C and uses Boolector library version in incremental usage as internal SMT Solver.
Using Boolector incrementally allows SmacC to use assumptions when executing the source of the program and to check satisfiability or unsatisfiability of multiple SMT formulas describing the state of the program. For example, SMT formulas representing error situations in the program are constructed and checked for satisfiability.
SmacC consists of a front-end that parses the source code from a subset of C into abstract syntax trees. The front-end is composed of an input buffer, a lexer to tokenize the input stream and a parser that generates syntax trees and the code-lists representing the source code of the program.
The front-end infrastructure resembles the front-end of a compiler, the code is based on lcc and the book 'A Retargetable C Compiler: Design and Implementation' [13].
The front-end operates on C expression level and generates syntax trees, composing them to one code-list on statement level. The code-list is then processed by the back-end of SmacC. The back-end extracts all execution paths through the program and generates a new code-list for each path through it. This step is called path-generation. Before identifying branching points loop constructs are unrolled and transformed to a sequence of if statements.
When a branch generated by path-generation is symbolically executed during BTOR-generation an SMT representation for the memory of the program is established. The representation allows modelling writing and reading memory and generating SMT instances to check memory access.
These SMT instance can be dumped to files in BTOR or SMT-LIB format to be replayed as benchmarks.
After a short example of the tool, the next sections give a detailed description and explanation of the techniques applied in SmacC.

4.1 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 checked.
In general it is safe to say that a program supplied as input to SmacC should not only compile with gcc without warnings, it should also compile without warnings with extra warning flags enabled, the most important ones being `-W -Wall -Wextra`.
In some cases this is important because certain implicit casts that are possible in C are not supported by SmacC. An exception is the return type of the `main` function which does not need to be of integer type `int`.
Appendix A describes the supported subset of C in detail, taking the Reference Manual presented in [16] as outline. Additionally, for unsupported constructs, it is stated whether support would require heavy changes in SmacC or if the implementation would require only modest effort.
The front-end of SmacC is based upon the ANSI C compiler lcc [13], with the exception that old-style [16] function definitions are not allowed. Additionally, non-constant initializations are not supported (for example: `int i = 4; int n = i;`.
Another exception is that global variables are not initialized to zero. When ANSI C does not specify the result of certain operations (for example integer overflow, treatment of division by 0 or right shifting a negative value [16]) then the behaviour of gcc was taken into account.
An ANSI C program consists of one or more translation units, SmacC does not support more than one translation unit. Only one input file is supplied as argument and it must not refer to code in other translation units. Only decimal representation is allowed for integer and character constants. The type system supported by SmacC includes only some of the basic types described in the ANSI C standard. Derived types can be constructed in addition to the basic types:

- `arrays`: Arrays of objects of a given type
- `functions` Return objects of a given type, partly supported
- `pointers`: Pointers to objects of a given type, partly supported

Functions are only partly supported, a translation unit may only contain one function declaration. Some function calls are supported as they are implemented in SmacC:

- `assert`: Asserts an expression
- `malloc`: Allocates memory
- `free`: Deallocates memory

`identifier`, `constant` and `( expression )` primary expressions are supported. Most postfix expressions are supported, some of them are not because the subset of C that requires them is unsupported (`.` or `->`).
All unary operators and most unary expressions are supported. Multiplicative and Additive and Shift operators with their usual semantics are supported by SmacC. Relational and equality operators evaluate to 0 if the specified relation is false and to 1 if it is true. All relational and equality operators are supported by SmacC. All bitwise and logical operators are supported by SmacC.
Only the non-augmented assignment operator `=` is supported by SmacC, augmented assignments are unsupported. The comma operator is supported for variable declarations, initializations and in `for` statements.
From the statements described in the ANSI C reference manual in [16], only a subset is supported: assignment and some function call expression-statements, selection-statement `if`, iteration-statement `for` and jump-statement `return`. It is mandatory to supply a condition expression for iteration-statement `for`.
The only preprocessor directive that is supported is the `#include` directive. `#include` has no effect on the SMT formula generated, it is only supported to be able to run examples that compile with gcc with SmacC, most important is the `#include <assert.h>` directive.

The next table summarizes supported constructs:

- Translation Unit: A valid declaration unit may only contain:
  - Global variable declarations of the supported types
  - One function declaration
- `if-else`, `for`
- `assert`, `malloc`, `free`

- `sizeof`
- `return`
- `#include`
- non-augmented assignment statements
- compound statement
- valid C expressions
- comments (`//` and `/* */`)

The following constructs are not supported:

- Translation units including multiple function definitions or function declarations.
- Preprocessor directives other than `#include`
- Floating point numbers, types and operations (`double`, `float`)
- No storage class and linkage specifiers (`extern`, `register`, `auto`, `static`, `volatile`, `const`)
- Jump statements, altering program flow: `break`, `continue`, `goto`
- `switch/case`, `(do-) while` Constructs
- `typedef`
- `struct`, `union`
- Bit-fields (there are no structs)

4.2 Front End

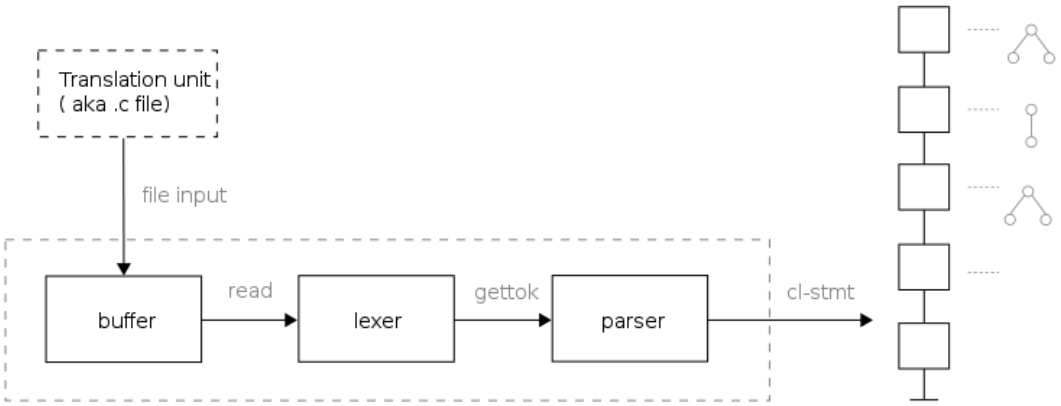


Figure 4.1: Front-End: Compiler infrastructure, from translation unit to code-list

The front-end gets as input a C file that contains a translation unit which lies in the supported subset of C.
In contrast to a C compiler, the first phase (textual replacement, the pre-processor) which carries out directives introduced by the `#` character is skipped.
After initializing the type system the lexer tokenizes the input stream and the parser creates abstract syntax trees according to the expression grammar, organizing them as statement elements in a code-list, as depicted in figure 4.1.
The code-list represents the connection to the back-end.

Abstract Syntax Trees (AST)

Expressions are stored in AST nodes which contain the type, the resulting type and the size of the expression.
AST nodes generally have two child nodes, some nodes may point to only one of those child nodes, special nodes may have three nodes (for example when the comma operator is used in a statement).
If necessary, nodes of AST store the symbol that is associated with the expression or the value that is associated with a node.
Additionally every AST node stores a Boolector expression that is evaluated when symbolically executing a code-list in the back-end. An operator that is worth discussion in more detail is the indirection operator `*` that in most cases is applied implicitly. The corresponding AST node is denoted by `INDIR` type.
An AST node for a variable is represented by node type `ADDR` because a variable is just a specific (named) address in memory, modeled by a Boolector bit-vector variable.

When accessing a variable (for example as in the return statement `return a`) it is necessary to return the content of the address identified by the variable not the address itself.
In this case indirection is applied implicitly, which means that the content of memory location `a` is fetched and then returned as result by the `return` statement.
`a = b + c` means write to the address identified by `a`, the sum of the values stored at memory location `b` and `c`.

The expression `&a` returns the address identified by the variable `a` and not its contents.

Figure 4.2: AST node resulting for expression '&a'

The next figure depicts the AST for the expression statement `a = 1`, the `CONST` node storing integer value 1.

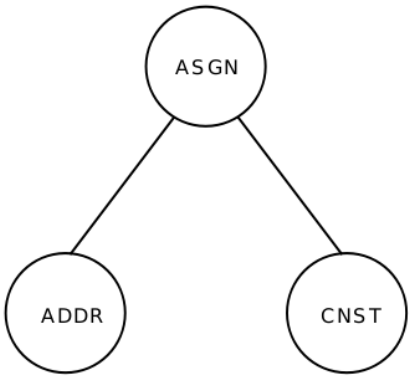


Figure 4.3: AST for the assignment 'a = 1'

As a last example for a C expression parsed into an AST consider expression statement `v + i` where both memory location `v` and `i` are dereferenced implicitly.

The node on top identifies the tree as addition of two values.

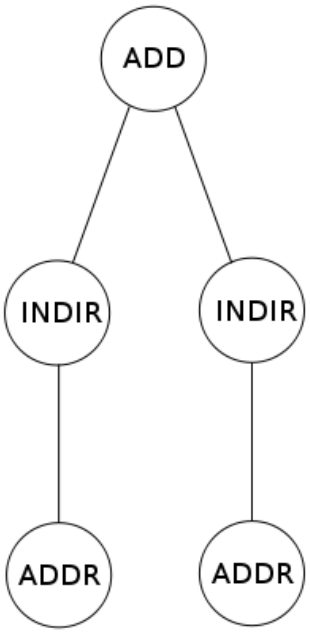


Figure 4.4: AST for expression statement 'v + i;'

Code-List

Abstract Syntax Trees that represent expressions are inserted into a code-list to be processed further by the back-end of SmacC.
Elements in the code-list represent statements. They contain sub-expressions in the form of AST. As an example consider the following short C translation unit, consisting of two variable declarations with initialization, two expression statements (assignments) and a jump statement.

```
int main (void)
{
  int i;
  int v;
  v = 1;
  i = 3;
  return v + i;
}
```

Listing 4.1: Simple C example

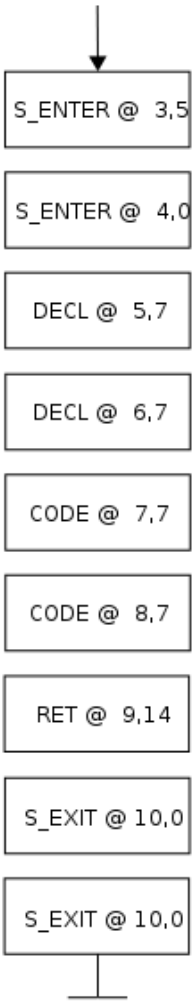


Figure 4.5: Code-list that results after parsing the simple example above

The resulting code-list contains entries representing the opening of two scopes (`S_ENTER`): one for the parameters of function main and another one for the block opened by the compound statement forming the functions body. The next two entries represent the declaration of local variables `i` and `v`. Initialization expressions for them are stored with the declaration elements AST. They are similar to the trees depicted in figure 4.3.
`CODE` entries stand for expression statements in most cases, two assignment statements in the example above.
The `RET` entry depicts the jump statement `return v + i`, the tree stored with this node is equal to the tree depicted in figure 4.4 but has a `RET` node on top of the `ADD` node. Integers after the `@` sign denote the line and column number of the entry in the translation-unit.

4.3 Back-End

The back-end gets as its input the full code-list that was generated by parsing the translation-unit. It detects and executes paths through the program symbolically by writing and reading to and from the BTOR array representing the memory of the program memory. It generates SMT formulas for the layout of the memory and checks satisfiability of properties that must hold. Symbolic execution happens in two phases called path-generation and BTOR-generation. Phase one, path-generation, flattens the full code-list by recursively unrolling loops up to a certain bound. After flattening path-generation processes the code-list, generating a new code-list until meeting an element that represents a branching point in the program. When a branching point is met, the code-list is duplicated and both paths are recursively processed further.
When a path through the program was generated, 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.

Path-Generation

Path-generation phase begins with flattening the code-list by unrolling iteration-statements to nested sequences of selection-statements.
At the time of writing only `for` iteration-statements are supported, they are unrolled to sequences of `if` statements. Transformation of the `for` statement is necessary:

```
for ( expression1 ;                    expression1
      expression2 ;          ⇒         if (expression2)
      expression3 )                      statement expression3
  statement                                ...
                                             if (expression2)
                                               statement expression3
```

Figure 4.6: For iteration-statement transformation

`expression1` of the `for`-statement, usually an initializer, must be pulled out and executed before executing the first `if`-statement the loop is transformed to. `expression2`, usually the condition of the loop, is used as condition for all `if`-statements. `expression3` must be executed after `statement` but only when the condition `expression2` evaluated to true.
It can be configured up to which bound SmacC unrolls `for` loops.

The next listing shows in pseudo code how the flattening step in path-generation phase is done, then follows an example of a code-list resulting from such a transformation, denoted as SmacC would output it.

`cl` denotes the code-list that was generated while parsing the input file, `flat_cl` is the resulting flat code-list. Expressions and statements are denoted as in figure 4.6 and as in 4.5 the integers denote the location in the translation-unit.

```
pathgen_flatten (in cl, inout flat_cl)
{
  while (cl has entries)
    if (cl entry != FOR statement)
      add entry to flat_cl
    else
      {
        identify for statement element
        insert expression3 into statement element
        add expression1 to flat_cl
        for (0 to bound k - 1)
          {
            add if element with expression1 as condition to flat_cl
            add pathgen_flatten (statement) to flat_cl
          }
      }
}
```

Listing 4.2: Path-generations flattening algorithm

The resulting flat code-list is further processed by path-generation phase, creating separate code-lists for branches through the program.
When an element in the flattened code-list is of kind selection-statement and execution could branch, the code-list representing the path through the program up until now is duplicated and path-generation is called recursively for both branches, generating a code-lists for each of the branches.
Currently `if`-statements are the only selection-statements supported, conditional (`?:`) expressions are translated directly into BTOR conditional expressions.
The condition of the selection-statement stays in the code-list because it will be used as a path-condition that is assumed while BTOR-generating the code-list later.
When a path through the program is fully extracted either after reaching the last element of the input code-list or by processing a return-statement element `pathgen` calls `btorgen` which then symbolically executes the path.

`pathgen` iterates over the flat code-list, adding entries to `path` representing the path through the program up until now. `RETURN` and `IF` elements in the code-list need special treatment. When processing a `RETURN` entry path-generation for the current path can

stop because after symbolically executing a return entry execution is finished.
`IF` entries require branching of the execution path. The path that is generated assuming that the condition of the if-statement holds is called if-path, the path that assumes the condition to fail is called else-path.
The current path through the program is copied and the entries constrained by the `IF` condition are added to the if-path. The if-path is continued by recursively adding entries after the if-statement. The else-path assumes that the condition does not hold by adding an else entry containing the negation of the condition and statements constrained by the condition and recursively continuing path-generation after the if-statement. When either a path through the program ends after a `RETURN` entry or reaching the end of the code-list, BTOR-generation symbolically executes it.

`cl` denotes the flat code-list, `path` represents the path through the program so far.

```
pathgen (in cl, inout path)
{
  while (cl has entries)
    if (cl entry == RETURN)
      {
        add entry to path
        finish cl
        btorgen path
        return
      }
    if (cl entry != IF statement)
      add entry to path
    else
      {
        copy path to else_path
        add if entry to path
        add else entry to else_path
        identify if block and add it to path
        identify else block and add it to else_path
        recurse with cl = entry after if statement and path
        recurse with cl = entry after else statement and else_path
      }
  btorgen path
  return
}
```

Listing 4.3: Path-generation

```
int main ()
{
  int i;
  int s = 0;
  for (i = 0;
       i < 4;
       i = i + 1)
    s = s + i;
  return s;
}
```

```
original codelist:
CSENTER @ (1,10)
  CSENTER @ (2,0)
   CDECLL @ (3,7)
   CDECLL @ (4,11)
   INIT @ (4,11)
   CFOR @ (6,4)
    CBBEG @ (6,4)
     CCODE @ (6,13)
    CBEND @ (7,2)
   CRET @ (7,10)
  CSEXIT @ (8,0)
CSEXIT @ (8,0)
```

```
flat codelist:
CSENTER @ (1,10)
  CSENTER @ (2,0)
   CDECLL @ (3,7)
   CDECLL @ (4,11)
   INIT @ (4,11)
   CCODE @ (6,4)
   CIF @ (6,4)
    CBBEG @ (6,4)
     CCODE @ (6,13)
     CCODE @ (6,4)
     CIF @ (6,4)
      CBBEG @ (6,4)
       CCODE @ (6,13)
       CCODE @ (6,4)
       CIF @ (6,4)
        CBBEG @ (6,4)
         CCODE @ (6,13)
         CCODE @ (6,4)
         CBEND @ (7,2)
        CBEND @ (7,2)
       CBEND @ (7,2)
      CBEND @ (7,2)
     CBEND @ (7,2)
    CASSUMEN @ (6,4)
   CRET @ (7,10)
  CSEXIT @ (8,0)
CSEXIT @ (8,0)
```

Figure 4.7: Translation-unit, the full code-list and the resulting flat code-list

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

```
path 0:
CSENTER @ (1,10)
  CSENTER @ (1,12)
   CDECLL @ (2,8)
   CIF @ (4,0)
    CBBEG @ (4,0)
     CRET @ (4,11)
    CBEND @ (5,0)
   CRET @ (5,8)
  CSEXIT @ (6,0)
CSEXIT @ (6,0)
```

```
path 1:
CSENTER @ (1,10)
  CSENTER @ (1,12)
   CDECLL @ (2,8)
   CELSE @ (5,0)
   CRET @ (5,8)
  CSEXIT @ (6,0)
CSEXIT @ (6,0)
```

Figure 4.8: Translation-unit and both paths through the program.

Btor-Generation

Btor-generation symbolically executes a path through the program. It generates BTOR expressions for C statements and expressions resulting in an SMT formula. Additionally constraints for the array modelling the programs memory are generated. If an entry in the code-list (`path`) 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 front-end to construct a Boolector variable for the variable in the source and store it with the symbol. When an identifier is parsed in an expression the Boolector variable for the symbol is stored in the AST node for the expression (`ADDR` as in figure 4.2). Variable declarations in the code also require updates to the SMT formula representing constraints on the programs memory when they are symbolically executed.
The next listings show how a code-list representing a path through the program is symbolically executed by functions `btorgen_path` that handles the program on statement level and `btorgen_generate` that handles it on expression level.

```
btorgen_path (path)
{
  while (path has entries)
    switch (entry type)
      {
      case DECL:
        btorgen_declaration (); // variable declaration
      case CODE:
        // assignment statement, expression statement, ...
        btorgen_generate (entry tree);
      case RETUN:
        // return-statement, boolector sat
        btorgen_generate (entry tree); return;
      case IF:
        // if-statement, boolector sat
        if (path is reachable) btorgen_and_add_path_condition
      case ELSE:
        // else-path of if-statement, boolector sat
        if (path is reachable) btorgen_and_add_path_condition
      case ASSERT:
        // assertion: check if assertion statements holds, boolector sat
        btorgen_generate (entry tree);
      }
  reset path conditions and memory model
}
```

Listing 4.4: Symbolic execution of a path through the program

`btorgen_path` may issue checks when processing the following entries:

- `RETURN`: Let Boolector check if it is possible that a specified return value is returned by the program or that it always returned.
- `IF`, `ELSE`: Let Boolector check if the condition can hold, if not abandon path and don't continue executing it.
- `ASSERT`: Let Boolector check if the assertion holds.

Transformation of AST to BTOR expressions is done by `btorgen_generate` that takes as input a tree representing a C expression.
Tree-nodes need to be transformed and may, as the mentioned code-list entries, lead to checks or updates in memory-model.
The postfix '-expression' in the table denotes the result of `btorgen_generat`ing a subtree of the node.

AST node	**Requires**	**BTOR Representation**
`RETURN`	check	asserted-return $\neq$ return-expression or asserted-return = return-expression
`CONST`	-	Construct Boolector constant
`CALL`	update	`free` or `malloc`, both update memory model
`ADDR`	-	Boolector variable in node
`INDIR`	check	read memory array at position indir-expression
`ASSIGN`	check	write to memory at position assign-expression1 the value assign-expression2
`COND`	-	Boolector conditional expression
relational operators	-	Boolector relational operators
bitwise `AND`, `OR`, `XOR` ones- twos-complement, `AND`, `OR`, `NOT`	-	corresponding Boolector operator
`ADD`, `SUB`, `MUL`, `DIV`, `MOD`	check	corresponding Boolector operator, overflow operator, division by zero
`RSHIFT`, `LSHIFT`	-	adjust bit-vector widths, Boolector shift operators
type conversions	-	conversions between data types, usually involves `boolector_slice`, `boolector_concat`, `boolector_sext`, `boolector_uext`

Table 4.1: AST nodes and their BTOR representation

```
BtorExp * btorgen_generate (Tree t)
{
  switch (ts node type)
    {
    case RETURN:
      // t->kid is an expression statement
      exp = btorgen_generate (t->kid);
      check asserted return value against exp;
    case CONST:
      // t->value represents the value of the const
      exp = boolector_const (t->value)
    case ADDR:
      // ADDR nodes store a Boolector variable
      exp = t->exp
    case INDIR:
      // read from (mem @ address),
      // where address can be ADDR or calculated
      exp = btorgen_read_address (t->kid);
      check if t->kid represents valid memory
    case ASSIGN:
      // write to (mem @ where),
      // where address can be ADDR or calculated
      where = btorgen_generate (t->kid1);
      what = btorgen_generate (t->kid2);
      btorgen_write_address (where, what);
      check if it is valid memory
    case COND:
      exp = boolector_cond (t->kid)
    case CVXY:
      // convert from X to Y
      exp = slice / extend accordingly
    case UNARY-OP:
      exp = boolector_op (t->kid);
    case BINARY-OP
      if (RSHIFT, LSHIFT)
        adjust width t->kid
      exp = boolector_op (t->kid1, t->kid2);
      if (ADD, SUB, MUL, DIV, MOD)
        check for overflow
      if (DIV, MOD)
        check division by zero
    }
  return exp;
}
```

Listing 4.5: Transformation of AST to BTOR expressions

Reading from Memory

If *mem* is the memory array with word-size 1 byte and address size 4 byte, and if assuming `int i` to be 4 bytes, then $read(mem, i)$ denotes the value of array *mem* at index i. $read(mem, i)$ reads the most-significant byte of integer `i` therefore reading an integer from memory requires more than one *read* operation on *mem* and additionally concatenation of the bytes read.

Writing to Memory

When v is a bit-vector of size 1 byte then $write(mem, i, v)$ denotes array *mem* overwritten at index i with v. Hence it is in most cases necessary to slice bit-vectors to bytes before writing the value to memory.

An Example

Consider the following short program on the left:

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

```
2 const 32 00000000000000000000000000000000
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
```

Figure 4.9: A C Program and the BTOR instance for its return statement

The BTOR instance for the return statement `return 0;` is depicted on the right and will briefly be explained, details follow after discussing the memory model of SmacC. Line 2 represents the constant 0, line 4, 5 and 6 BTOR variables necessary for constructing 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 marking the end of the memory areas are equal to the addresses marking the begin of the memory areas, therefore the same variables are used to formulate the properties. 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. As mentioned, line 17 is the conjunction of the properties above and the formula requiring the return value to be equal to zero.

Memory Model

The memory model is inspired by the memory model known to UNIX systems. It is an SMT formula that constrains the array variable modelling the programs memory during symbolic execution and allows to check whether memory accesses in the SMT representation of 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 UNIX process (in this case: the program being executed on the machine) into three segments [24]:

- Text Segment: machine instructions, executable code
- Global / Data Segment: global variables, constant strings, but also dynamic memory (C: `malloc`, system call `brk`)
- 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 would be allocated in data segment by calls to `malloc` is modeled by heap-area.
The stack segment is called stack area but holds, as in the UNIX model, parameter variables and local variables (but no return addresses). The SmacC memory model is a Boolector array variable representing the programs memory and an SMT formula that constraints valid regions in the array to those regions that are valid in the program.
If addresses were constants instead of variables, the first global variable would always have the same constant address in SmacC (the first stack or heap variable, too). This would allow to assert a constant values as addresses for a variable, an assertion that will in most cases fail in the compiled program (for example, SmacC would report that

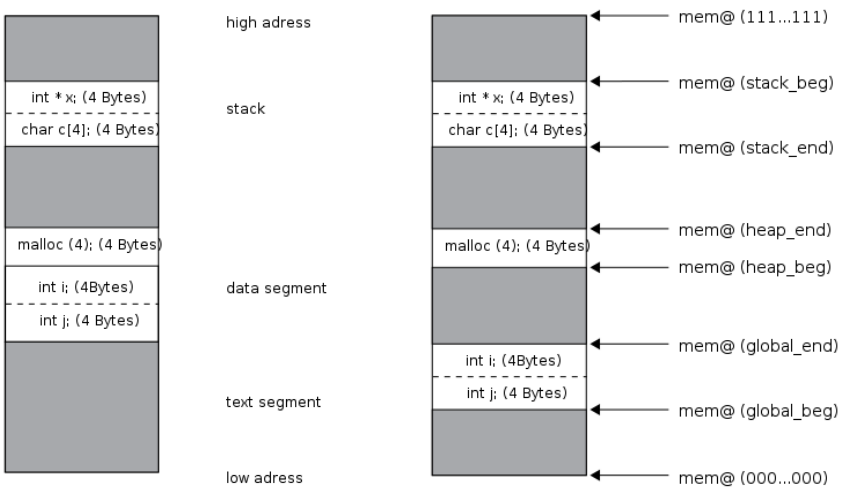


Figure 4.10: Simplified view of the UNIX memory-model of a C program and its representation in SmacC.

the assertion `assert (&i == 1);` holds if `i` is the first global variable in the program and global addresses were to start at constant address 1).
A valid region is in most cases a set of indices of the array. As an example for such a region consider a program that has only one integer variable of size 4 bytes declared, represented by 4 one-byte wide indices in the Boolector array.
The program may only access memory referenced by this integer variable, all other memory accesses are considered invalid. Hence only the 4 indices of the array representing the integer are valid, all other indices are considered invalid.
When the Boolector memory array is accessed outside valid regions then in the C program memory can be accessed incorrectly.
As was mentioned, valid memory addresses in the program are represented by Boolector variables used as indices for the memory array in the SMT representation.
The order of variables that is required to check the memory-model is established via Boolector assumptions that are added before each check.

The following memory accesses are considered invalid:

- 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 grey in the figure that follows.

- 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.

The program depicted in figure 4.10 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.

Global area is part of the data segment in the UNIX memory-model. In SmacC global area is established by storing indices, Boolector bit-vector variables, that represent the name of the address (symbol of the variable) and the size of the memory region.
Additionally an assumption about the distance between the variable declared and the beginning of global memory is stored.
Global area does not change during the (symbolic) execution of the program.

- `globals`: A list that contains Boolector expressions modelling a global variable declaration in the C program: its index in memory array (Boolector variable), size of the declaration (Boolector expression) and offset to beginning of global area (Boolector expression).

- *global_beg*: A Boolector variable representing the lowest index of valid global memory in the memory array.

- *global_end*: A Boolector variable representing the highest index of valid global memory in the memory array.

Heap area, in the UNIX memory-model also in data segment, is an independent memory area in SmacC. It is organized like global area but might change during symbolic execution of the program.

- `heap`: A list containing Boolector expressions modelling a call to `malloc` in the C program.

- *heap_beg*: Boolector variable representing the lowest index of valid heap memory in memory array.

- *heap_end*: Boolector variable representing the highest index of valid heap memory in memory array.

Stack area behaves like the stack segment of the UNIX memory-model: local or parameter variable declarations make the stack grow from top to bottom, leaving a scope where a local or parameter variable was declared lets the stack shrink, invalidating the memory location the local or parameter variable referenced.
As for global variables or heap memory it is necessary to store an index for a stack variable as Boolector variable and a size expression for the region. Because the stack

segment must be able to grow and shrink, the offset expression is calculated relative to the current lowest address of the stack (denoted stack-pointer).
To model variables in the stack area a stack data-structure stores a list of declared variables and a stack-pointer expression for each scope. If a variable is declared on the stack the stack-pointer is decreased by the size of the declaration and pushed onto the stack. If a scope is closed the old stack-pointer is restored and the list of local variables for the scope is thrown away.

- `stack`: A stack of variable-lists (similar to `globals`) and a Boolector variable expression. The top of the stack represents the current scopes local variables and the stack-pointer for this scope. Stack-pointer can be seen as changing *stack_end*.

- *stack_beg*: Boolector variable representing the highest index of valid stack memory in the memory array.

- *stack_end*: The Boolector expression on top of stack, stack-pointer.

To establish the representation of the UNIX memory-model it is necessary to add assumptions about Boolector variable expressions that represent memory addresses.
When BTOR-generation begins to symbolically execute a path through the program, the general memory layout is established by assuming the following formulas:

1. $global\_beg \leq global\_end$

2. $global\_end < heap\_beg$

3. $heap\_beg \leq heap\_end$

4. $heap\_end < stack\_end$

5. $stack\_end \leq stack\_beg$

   Additionally, in the absence of any variable declarations:

6. $global\_beg = global\_end$

7. $stack\_beg = stack\_end$ (`push` *stack_end* on `stack`)

8. $heap\_beg = heap\_end$

The next figure shows the memory-model of SmacC right after initialization and no variables declared.

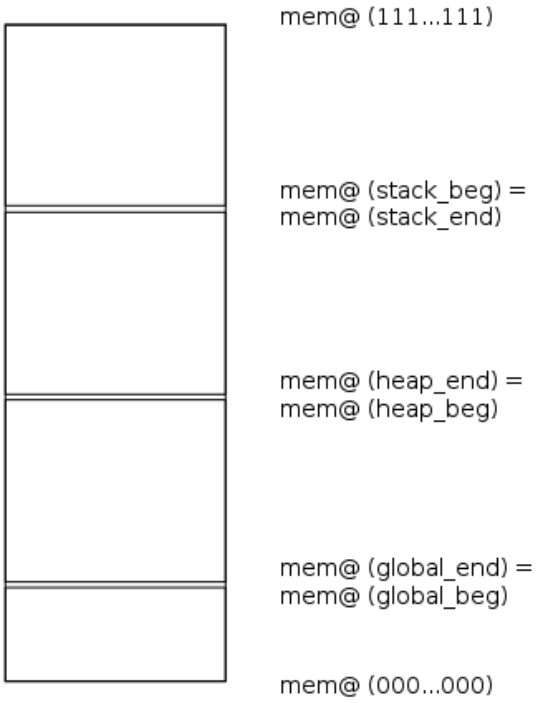


Figure 4.11: SmacC memory-model after initialization

When variables are declared or dynamic memory is allocated the memory-model needs to be updated to include the new constraints. Global declarations all occur after each other, before entering any scope. When a scope was opened, no global declarations will occur and *global_end* does not change anymore.
stack_end and *heap_end* can change during the programs symbolic execution, local declarations in different scopes require changes to the stack-pointer (*stack_end*), calls to `malloc` require *heap_end* to be updated.

When a global declaration for `global_var` is executed on a path, the memory-model needs to be updated:

- *global_var* and its size are added as entry to `globals` list
- equality *global_var* = *global_end* is added to list entry
- *global_end* needs to be updated, it is now *global_end* + *decl_size_in_bytes*

When entering a scope, the memory-model needs to consider local declarations for the scope.
Therefore the current stack-pointer needs to be pushed on top of `stack`, representing the new valid stack-pointer. When a local declaration `local_var` is executed the memory-model must be updated:

- *local_var* and its size are added as entry to the list on top of `stack`.
- stack-pointer needs to be updated to be $stack\_end - decl\_size\_in\_bytes$.
- equality $local\_var = stack\_end$ (the new stack-pointer) is added to the list entry.

When exiting a scope the old stack-pointer is restored and the list of local variables is dropped.

`malloc` and `free` modify assumptions about the heap area. A call to `malloc` can be seen as a global declaration, `free` marks the region as invalid, it will not be considered as valid for memory checks. Memory deallocated by `free` will not be used by `malloc` again.

Fig. 4.10 shows the memory layout of a C program and its representation in SmacC. It would require the following formulas to be assumed before accessing memory or checking properties:

Valid memory in the program:

- Declarations in stack area: `int * p`, `char c[4]`
- Dynamic memory: 4 bytes
- Declarations in global area: `int j`, `int i`

SMT formulas to model the memory of the program, beginning with formulas for general memory layout.

$$\begin{gathered} global\_beg \leq global\_end \wedge \\ global\_end < heap\_beg \wedge \\ heap\_beg \leq heap\_end \wedge \\ heap\_end < stack\_end \wedge \\ stack\_end \leq stack\_beg \end{gathered}$$

SMT formulas for global variables:

$$\begin{aligned} i = global\_beg \wedge \\ j = global\_beg + 4 \wedge \\ global\_end = global\_beg + 8 \end{aligned}$$

SMT formulas for heap memory:

$$\begin{aligned} heap\_var\_1 = heap\_beg \wedge \\ heap\_end = heap\_beg + 4 \end{aligned}$$

SMT formulas for stack variables (assuming only 1 scope in the example):

$$\begin{aligned} p = stack\_beg - 4 \wedge \\ c = stack\_beg - 4 - (4 * 1) \wedge \\ stack\_end = stack\_beg - 8 \end{aligned}$$

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.

They can be categorized into Verification Checks

- Assertions: 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

Before going into more details of checks, consider the steps that were performed up until now:

1. Parse C file into AST and code-list.
2. Flatten code-list and extract paths through the program.
3. Symbolically execute each path, some entries require additional treatment: declarations that modify the memory-model and entries that lead to checks.
4. A check is an SMT formula assumed in addition to the formula representing the programs memory layout, followed by a call to `boolector_sat`.

Assertion Check

Boolector returns UNSAT if an assertion holds on a path through the program. Consider the following example:

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

Listing 4.6: Assertion Check

The following Boolector variables exist:

$$global\_beg, global\_end,$$
$$heap\_beg, heap\_end,$$
$$stack\_beg, stack\_end,$$
$$mem, i$$

The above variables lead to the following assumptions:

$$\begin{gathered} global\_beg \leq global\_end \wedge global\_end < heap\_beg \wedge \\ heap\_beg \leq heap\_end \wedge heap\_end < stack\_end \wedge \\ stack\_end \leq stack\_beg \wedge global\_beg = global\_end \wedge \\ heap\_beg = heap\_end \wedge i = stack\_beg - 4 \wedge \\ stack\_end = stack\_beg - 4 \end{gathered}$$

The assumption that must be checked for this assertion check is:

$$\begin{gathered} read(mem, i) = 00000000 \wedge \\ read(mem, i+1) = 00000000 \wedge \\ read(mem, i+2) = 00000000 \wedge \\ read(mem, i+3) = 00000000 \wedge \end{gathered}$$

which must be UNSAT for the assertion to hold (in the implementation the 4 byte values would be constructed and compared).

Program Return Check

Boolector returns UNSAT if a program returns expected value on a path through a program. Consider the following example:

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

Listing 4.7: Program Return Check

The following variables are considered by the memory-model:

$$\begin{gathered} global\_beg, global\_end, \\ heap\_beg, heap\_end, \\ stack\_beg, stack\_end, \\ mem, i \end{gathered}$$

The above variables lead to the following assumptions:

$$\begin{gathered} global\_beg \leq global\_end \wedge global\_end < heap\_beg \wedge \\ heap\_beg \leq heap\_end \wedge heap\_end < stack\_end \wedge \\ stack\_end \leq stack\_beg \wedge global\_beg = global\_end \wedge \\ heap\_beg = heap\_end \wedge i = stack\_beg - 4 \wedge \\ stack\_end = stack\_beg - 4 \end{gathered}$$

The initialization of variable `i` leads to a value being written into the memory array:

$$\begin{gathered} write(mem, i, 00000000) \wedge \\ write(mem, i+1, 00000000) \wedge \\ write(mem, i+2, 00000000) \wedge \\ write(mem, i+3, 00000000) \wedge \end{gathered}$$

The assumption that must be checked for the program return check depends on whether option `--force-return` (`-fr`) was used as parameter to SmacC.
`-fr` requires the path to return the specified value, if it is possible that some other value is returned it is considered an error. Without `-fr` SmacC checks whether the specified return value can be returned.
The default value assumed for the programs return is 0, hence the default SMT formula checked for satisfiability is

$$\begin{gathered} read(mem, i) = 00000000 \wedge \\ read(mem, i+1) = 00000000 \wedge \\ read(mem, i+2) = 00000000 \wedge \\ read(mem, i+3) = 00000000 \wedge \end{gathered}$$

and in the case where `-fr` is supplied

$$\begin{gathered} read(mem, i) \neq 00000000 \wedge \\ read(mem, i+1) \neq 00000000 \wedge \\ read(mem, i+2) \neq 00000000 \wedge \\ read(mem, i+3) \neq 00000000 \wedge \end{gathered}$$

is checked for unsatisfiability (again, in the implementation values are compared after being concatenated to the `int` size).

Division by Zero Check

Division by zero check could be modeled by the following assertion in the code:

```
int a, b, c;
a = b / c;
```

⇒

```
int a, b, c;
assert (c);
a = b / c;
```

Figure 4.12: Division by zero check written as assertion check.

The SMT formula generated and checked for satisfiability is

$$\begin{aligned} read(mem, i) \neq 00000000 \wedge \\ read(mem, i+1) \neq 00000000 \wedge \\ read(mem, i+2) \neq 00000000 \wedge \\ read(mem, i+3) \neq 00000000 \wedge \end{aligned}$$

Overflow Check

Overflow checks are issued on all arithmetic operations and need no special treatment since operators to model overflow exist in Boolector [3].

```
char a, b = 2, c;
a = b * c;
```

Overflow on arithmetic operations is possible if Boolector returns satisfiable with the following assumption added:

$$smulo(read(mem, b), read(mem, c))$$

$smulo(a, b)$ denotes signed multiplication overflow when multiplying signed values a and b.

Path Condition Check

Path Conditions (`if`, `else` conditions) are `btorgen_generated` and added to the assumption before each check.

If a path condition is satisfiable it is added to the set of active path conditions, BTOR-generation continues on the path. If a path condition is unsatisfiable the current path is added to the set of unreachable paths and abandoned.
Path-generation selects the next path that is not in the set of unreachable paths and calls BTOR-generation.

```
char a;
if (a > 0)
  assert (a);
```

Because $read(mem, a) > 0)$ (the condition) is satisfiable, $read(mem, a) > 0$ is added to the set of path conditions. The assertion will hold because $read(mem, a) > 0$ is in the set of assumptions.
The following Boolector variables exist:

$$\begin{gathered} global\_beg, global\_end, \\ heap\_beg, heap\_end, \\ stack\_beg, stack\_end, \\ mem, a \end{gathered}$$

In addition it is assumed that:

$$\begin{gathered} global\_beg \leq global\_end \wedge global\_end < heap\_beg \wedge \\ heap\_beg \leq heap\_end \wedge heap\_end < stack\_end \wedge \\ stack\_end \leq stack\_beg \wedge global\_beg = global\_end \wedge \\ heap\_beg = heap\_end \wedge a = stack\_beg - 4 \wedge \\ stack\_end = stack\_beg - 4 \end{gathered}$$

The formula that needs to be satisfiable to continue on the path is:

$$read(mem, a) > 0$$

If Boolector returns satisfiable for the conjunction of the formulas above then the formula will be added to the set of assumptions (and assumed before the assertion is checked).

Indirection / Assignment Check

Indirection checks verify the validity of an dereferenced address, assignment checks the validity of an address written. Indirection and assignment checks are issued only for array and pointer expressions, that is AST nodes of type `INDIR`, `ASGN`, `SUBP` or `ADDP`. The difference between the checks lies in the addresses used for checking indirections and assignments.

Consider the following variation of an example of the EXE tool [7]. `a` is an array of 4 unsigned integers, initialized with some values. Assume the else path is analyzed after branching at point (1) in line 11. Because `i` is not initialized but constraint to be smaller than 4 by the path condition `p` points to one byte (the least significant) of an element in `a` after executing line 14. In line 15 the value of the array element `p` points to is decreased. In line 17 `t` gets assigned the array element with index equal to a value in the array.
It is easy to see that if `i` has value 2 in line 14 then in line 15 the value 5 will be decreased to 4 and then in line 17 out-of-bounds element 4 of array `a` is incorrectly accessed.In line 15 (2) checks are issued for the left-hand side address of the assignment and the right-hand side indirection `*p`.
In line 17 (3) indirection checks are issued for indirection `*p` and for array access `a[*p]`.

```
 1:    void main ()
 2:    {
 4:      unsigned a[4];
 4:      unsigned i, t; // introduces non-determinism
 5:      char * p;
 6:      a[0] = 1;
 7:      a[1] = 3;
 8:      a[2] = 5;
 9:      a[3] = 2;
10:
11:      if (i >= 4)              // 1
12:        return;
13:
14:      p = (char *) a + i * 4;
15:      *p = *p - 1;             // 2
16:
17:      t = a[*p];               // 3
18:   }
```

Listing 4.8: A variation of EXE example

SmacC implements two methods to check addresses in assignment statements or indirection expressions:

- Arbitrary but Fixed (to be outside valid memory): the basic memory check. It requires one satisfiability check to determine if a memory access is invalid.

- Out of Bounds: checks if an array access is invalid. It requires one satisfiability check to determine if an array access is out-of-bounds. It can also be used to check pointer arithmetic, but pointers can be modified in a way that invalidates the result of the check.

SmacC allows both methods used separately, combined or none at all.

Basic Memory Check: Arbitrary-but-Fixed

The basic memory check constructs a Boolector bit-vector variable *abf* a 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 checked. If it is satisfiable that the address 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.
Using only basic memory checks SmacC can miss incorrect memory access if `malloc` and `free` is involved.
Another problem of the basic memory check is that the results of the check can depend on the order in which variables were declared. This can also happen in C programs and is hard to capture.

In a more formal way the basic memory check assumes:

$$\begin{aligned} abf > stack\_beg \wedge abf > global\_end \wedge \\ abf < heap\_beg \wedge abf > heap\_end \wedge \\ abf < stack\_end \wedge abf < global\_beg \end{aligned}$$

and for each region $free\_var_i$ that was freed:

$$abf >= free\_var_i \wedge abf < free\_var + free\_var_i\_size$$

and checks if it is satisfiable that:

$$abf = addr$$

Clearly because of the constraints on *abf* if the above SMT formula is satisfiable for any byte of *addr* invalid memory is accessed. Note that *stack_end* is the stack-pointer currently valid.

Array Memory Check: Out-of-Bounds

The out-of-bounds memory check identifies in an AST an address and an offset for memory access against which is checked. It can be used to verify array accesses and in most cases pointer expressions. For basic array accesses this address, *check_addr*, is the address of element 0 in the array, for pointer expressions it is the address the pointer points to. The offset is, in both cases, a node in the AST because array access decays to pointer nodes, because of this, an out-of-bounds check is performed for all assignments or indirection statements involving pointers.
After identifying the address *check_addr* for which the check is performed, all variables in the order stack variables, global variables, heap variables are compared to the address. If it is satisfiable that *check_addr* is greater or equal to a variable, v, and *check_addr* + *offset* is smaller than v + *size_of_variable* then a candidate is found.
If additionally it is unsatisfiable that *check_addr* + *offset* is greater or equal to the v + *size_of_variable* a valid address was found. If the above conditions hold for no variable in the set of variables, then the memory access is invalid.
addr represents the address being accessed, *check_addr* the base address identified in the AST. *offset* is also identified in the tree, if there is no offset it is set to zero (for example when dereferencing a pointer `*p`).

Array memory check finds from the set of declared variables one variable v that suffices:

$$check\_addr \geq v \wedge check\_addr + offset \leq v + size\_of\_v$$

If no such variable is found in the set of global, local or heap variables the access is invalid. If there is such a v and

$$check\_addr + offset \geq v + size\_of\_v$$

is satisfiable, the access could cross variable boundaries and is considered invalid.

It is possible to reformulate the method described above to only need one satisfiability check of one big formula that is constructed from all variables in the program: an access is invalid for a variable v if the address *check_addr* can be smaller than the address of the variable or the highest address accessed ($check\_addr + offset + read\_size$) is greater or equal to the highest address of v ($v + v\_size$).
The formula must be unsatisfiable for one of the variables in the program, otherwise the memory access is invalid. The formula is constructed for each variable in the program. If the conjunction of the formulas is satisfiable, non of the variables can be the address that is accessed and hence the memory access is invalid. If the conjunction is unsatisfiable there was a variable that was accessed correctly.

Because an array check requires the formula to evaluate to unsatisfiable, no assignment is printed when it detects an access to be invalid.

Limits of the Model

The array memory check 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 reused 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.
Consequences are described in the Outlook section.

Memory Checks for EXE Example

Consider the variation of the EXE example listed above. After symbolically executing the if statement and examining the else-path the representation of the program consists of the following Boolector variables:

$$global\_beg, global\_end,$$
$$heap\_beg, heap\_end,$$
$$stack\_beg, stack\_end,$$
$$a, i, t, p, mem$$

The following SMT formulas represent assumptions for the memory model:

$$
\begin{gathered}
global\_beg \leq global\_end \wedge global\_end < heap\_beg \wedge \\
heap\_beg \leq heap\_end \wedge heap\_end < stack\_end \wedge \\
stack\_end \leq stack\_beg \wedge global\_beg = global\_end \wedge \\
heap\_beg = heap\_end \wedge a = stack\_beg - 4 * 4 \wedge \\
i = stack\_beg - 20 \wedge t = stack\_beg - 24 \wedge \\
p = stack\_beg - 28 \wedge stack\_end = stack\_beg - 28
\end{gathered}
$$

Additionally the path condition

$$read(mem, i) < 4$$

is valid for the rest of the path.

For the statement `*p = *p - 1;` consider its AST, the expression on the right-hand side, `*p`, gets checked:

To check if the address could be outside of valid memory, SmacC creates a Boolector variable *abf* restricts its value such that it does not overlap with an address that is known to be valid. Then Boolector checks if the SMT representation of `*p` can be equal to *abf*.

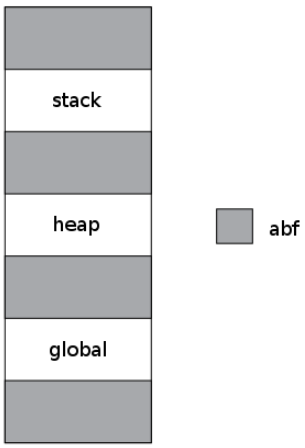


Figure 4.13: Grey color represents areas in memory *abf* might point to.

Now consider expression `a[*i]` from EXE example, depicted as AST in the next figure, and assume that the basic memory check did not find a problem. If array memory check is also executed SmacC checks not only for access out of valid memory but also for access out-of-bounds or across variable borders.
Array memory check would identify address of `a` as *check_addr*, `i * 4` as *offset* and *addr* (`a + i * 1 * 4`) as address being dereferenced.

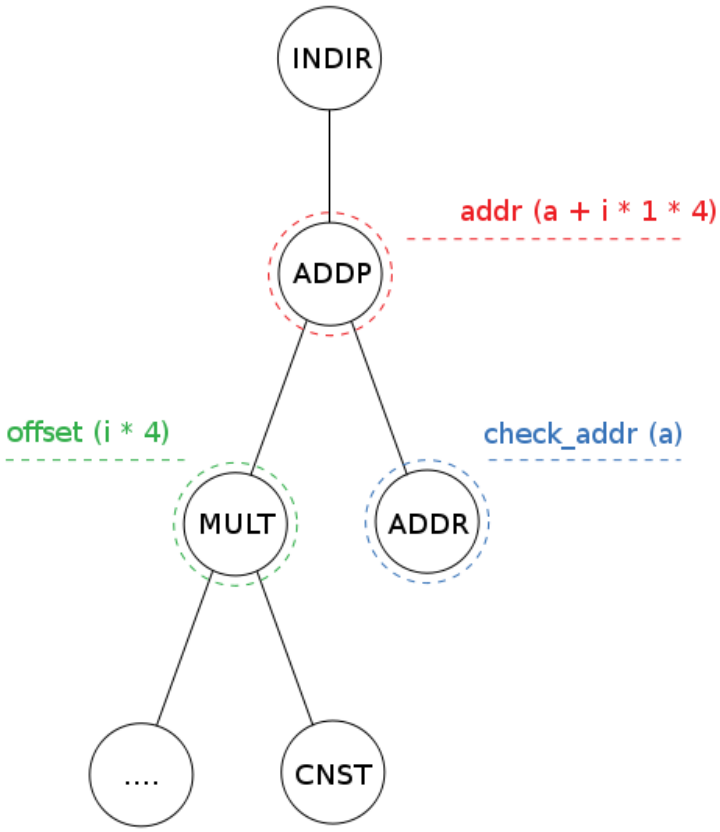


Figure 4.14: AST for a[*i] array access. Grey shows what nodes array memory check would identify to perform the check.

From the set of declared variables (`global`, `stack` or `heap` variables) array check searches for variable v that satisfies

$$check\_addr \geq v \wedge check\_addr + offset \leq v + size\_of\_v$$

selecting the variable that would fit the address.
If such a v is found, it is necessary to check that it is unsatisfiable that

$$check\_addr + offset > v + size\_of\_v$$

This unsatisfiability check is essential because *offset* could be symbolic (e.g.: when the target of `*p`, `i`, is uninitialized) and without checking unsatisfiability of the formula `a[i]` would be considered legal in this example.
If SmacC does not find such a v then the memory access was across variable boundaries and therefore invalid.
Elaborating on the statement from the EXE example, one can observe that $addr =$ `a + 4 * *p` with `p` pointing to a value of `a` that was decreased by one.
Hence possible values for *addr* are:

$$addr = a + 4 * 0 \; addr = a + 4 * 2 \; addr = a + 4 * 4 \; addr = a + 4 * 1$$

Identifying `a` as *check_addr* and symbolic *offset* for which holds:

$$offset = 0 \vee offset = 8 \vee offset = 16 \vee offset = 4$$

The algorithm finds `a` as fitting v from set of declared variables.
Hence

$$check\_addr \geq v \wedge check\_addr + offset \leq v + size\_of\_v \text{ (in source code terms: } a \geq a \wedge a + *p * 4 \leq a + 15)$$

is satisfiable (assuming for example `*p` $= 0$).
Therefore the second SMT formula needs to be checked:

$$check\_addr + offset > v + size\_of\_v \text{ (in source code terms: } a + *p * 4 > a + 15$$

Which can be satisfiable, assuming for example $*p = 4$), therefore SmacC concludes that the access is out-of-bounds for `a[*p]`.

Benchmark Generation

Benchmarks can be created by supplying the `--create-benchmarks` or the short form `-cb` to SmacC. Instead of using Boolector to check for defects or verify properties, a BTOR or SMT-LIB file is dumped as output.
For most examples it is then necessary to identify interesting paths through the program and select checks interesting as benchmark. Internally it is necessary to conjunct all assumptions (memory layout, path conditions) and then conjunct those with the Boolector expression that needed to be checked if `-cb` were not supplied.

4.4 Outlook

Problems with Array Memory Check

As mentioned, the array memory check can be used to detect memory errors in pointer expressions but one can easily lever the array check to report wrong results:

```
 1:    #include <assert.h>
 2:    void main ()
 4:    {
 4:      int * i;
 5:      int * j;
 6:      int ** l;
 7:      int k = 1;
 8:      i = &k;
 9:      assert (*i);
10:      j = &k;
11:      j = j - 1;
12:      assert (*(j + 1));
13:      l = &i;
14:      assert (**(l));
15:    }
```

Listing 4.9: Array memory check is fooled by pointer arithmetic

Calling SmacC (with overflow checks disabled) results in the following output:

```
[CHECK] legal indirection, abf @ (8,11)  OK

[CHECK] legal indirection, oob @ (8,11)  OK

[CHECK] assertion violation @ (8,11) OK

[CHECK] legal indirection, abf @ (11,17)  OK

[CHECK] legal indirection, oob @ (11,17)  ERROR:
dereferencing address out of bounds

[CHECK] assertion violation @ (11,17) OK

[CHECK] legal indirection, abf @ (13,14)  OK

[CHECK] legal indirection, oob @ (13,14)  OK

[CHECK] legal indirection, abf @ (13,14)  OK
```

```
[CHECK] legal indirection, oob @ (13,14)  OK

[CHECK] assertion violation @ (13,14) OK
```

Listing 4.10: Wrong out-of-bounds access is reported

As can be seen, array check handles pointer accesses correctly as long as the value identified as check address *check_addr* is usable When the value of the pointer is modified as for example in the assignment statement `j = j - 1`, the address identified as *check_addr* is not usable for out-of-bounds access detection.

In this example, after the second assignment to `j`, the address `&k - 1` is used as *check_addr*. `&k - 1` corresponds to the address of `l`, memory gets accessed at address `&k`. It is easy to see that using address of `l` as *check_addr* lets the out-of-bounds check fail because to succeed it would require the variable at address `&l` (which is `l`) to be of size 8 bytes, but `l` has size 4 bytes. Hence the access is reported out-of-bounds.

As mentioned, this inconsistency could be handled by treating array variables differently than pointer variables and omitting out-of-bounds check for non-array variables.

Problems with Memory Limits

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):

$$\begin{gathered} heap\_beg \leq heap\_end \wedge \\ m1 = heap\_beg \wedge heap\_end = heap\_beg + m1\_size \end{gathered}$$

where *m1_size* is the number of bytes allocated by `malloc`. It forces *heap_end* to be equal to the highest address 1111....1111. The second call to `malloc` allocates memory such that:

$$\begin{gathered} heap\_beg \leq heap\_end \wedge \\ m1 = heap\_beg \wedge m2 = heap\_beg + m1\_size \wedge \\ heap\_end = heap\_beg + m1\_size + m2\_size \end{gathered}$$

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. Consequences of contradicting assumptions are discussed in the next section.

Problems with Path Conditions for Loops

There is a problem that 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.
Consider the following example that contains two methods to round a value to the next power of two and asserts the output of both methods to be equal (the second one is taken from [14]:

```
#include <assert.h>
int main ()
{
  int x;
  int value;
  int result;
  x = value;

  if (value >= 0)
    {
      for (result = 1; result < value; result = result << 1)
        ;               // method 1

      x = x - 1;
      x = x | x >> 1;
      x = x | x >> 2;
      x = x | x >> 4;
      x = x | x >> 8;
      x = x | x >> 16;
      x = x + 1;        // method 2: hacker's delight

      assert (result == x);
    }
  return (cnt);
}
```

Listing 4.11: Next power of 2

If SmacC is called on the example, it will report that the assertion holds, no matter what bound for loop unrolling was specified. If `value` is, for example, 231 the loop would execute 8 times and yield the same result as method 2. If a bound of 3 is supplied to SmacC it reports that the assertion holds, when it does not hold.
As mentioned, SmacC assumes the loop condition to hold after symbolically executing it. In this example, after executing the loop 3 times it is assumed that $result \geq value$ but it is easy to see that after 3 iterations `result` is actually 4. The assumption can never be satisfied, because $4 \geq 231$ is unsatisfiable. When checking if `result == x` can evaluate to zero (failing assertion), Boolector will always report unsatisfiable, even if `result == x` clearly evaluates to zero because $4 \neq 256$.
Nevertheless SmacC can be used to detect the upper bound for the loop, which is, in this case, easy to see: if SmacC is called on this example with a bound of 32, then SmacC will report the last `if` condition for the unrolled loop to be unsatisfiable, or if `-cb` was specified boolector will report UNSAT if called with the dump for the condition. Therefore the uppermost bound for the loop is 31.
Knowing the upper bound for the loop allows to formulate the following verification condition:

"For all values of `value` that need the loop to execute *bound* (that needs to be ≤ 31) times, the two methods of calculating the next power of 2 that is greater than `value` always return the same result."

The verification condition can then be verified for a certain bound, for example 31, by calling `./smacc -if pow2roundup.c -no -mc 0 -nd -fp -nr -bk 31`. For all values that would execute the loop 31 times, both methods return the same result.
Calling SmacC with `-bk` 0..31 (and verifying that the assertion holds in each call!) could be seen as a bounded model-checking run on the example.

Future Work

Much could be done to improve usability and efficiency of SmacC:

- Allow multiple function definitions and calls (for example by in-lining)
- Improve or replace path-generation algorithm
- Treat arrays differently from pointers such that array memory check only considers arrays but not pointers (requires changes in front-end).
- Extend supported subset of C: loops (`while`, `do-while`), selection statements (`switch-case`) and data types (`struct`, `union`, `enum`.
- Support function calls usually defined in the C standard library: `memcpy`, `strcpy`, `memmove`, `memset` etc.
- Improve output, making it easier to identify the problem if there is an error.
- Improve output for unrolled loops.
- Develop a graphical user interface.

Allowing multiple translation units and function definitions and extending the supported subset of C would allow checking more complex programs.
With improved output capabilities it would be easier to find input that causes checks to fail and replay them on the compiled program.
Getting rid of the path-generation step and checking all paths in one pass would greatly improve the program, especially the benchmark generation capabilities.
Another topic worth investigating would be the use of separation logic [18]:
It introduces the binary separation conjunction operator $*$. The separating conjunction asserts that its sub-formulas hold for disjoint parts of the heap [21]. For example: $P * Q$ not only assert $P \wedge Q$ but also that P and Q reason about different portions of the heap.

Introducing operators with separation conjunction-like semantics could simplify the construction of a memory model and reasoning about it.

Related Work

CBMC - Bounded Model Checker for ANSI C

CBMC is a Bounded Model Checker for ANSI C and C++ programs. It allows verifying array bounds, pointer safety, exceptions and user-specified assertions [17]. 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 [10]. 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 [17].
CBMC can also be used to check behavioral consistency of C and Verilog programs (Hardware and Software Equivalence and Co-Verification) [9].
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 [8]. An assertion for each loop verifies that the unwinding bound [8] 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. Details can be found in [8] where the following example for SSA transformation is taken from:

```
x = x + y;
if (x != 1)
  x = 2;
else
  x++;
assert (x <= 3);
```

```
x1 = x0 + y0;
if (x1 != 1)
  x2 = 2;
else
  x3 = x1 + 1;
x4 = (x1 != !) ? x2 : x3;
assert (x4 <= 3);
```

$\mathbf{C} :=$
$x1 = x0 + y0 \wedge$
$x2 = 2 \wedge x3 = x1 + 1 \wedge$
$x4 = (x1 != 1)?x2 : x3$
$\mathbf{P} :=$
$x4 \leq 3$

Figure 4.15: SSA transformation of CBMC. The program on the left is transformed to the constraint and property on the right.

It is difficult to compare this approach to SmacC because of another obvious difference: path generation not necessary in CBMC. SmacC would translate the code snippet to 2 paths:

path0 :	**path1** :
$write(mem, x, y)$	$write(mem, x, y)$
$assume(x! = 1)$	$assume(!(x! = 1))$
$write(mem, x, 2)$	$write(mem, x, read(mem, x))$
$root(eq(0, slte(read(mem, x), 3)))$	$eq(0, slte(read(mem, x), 3))$

Figure 4.16: Sloppy notation of the transformed program in SmacC, omitting memory layout

CBMC supports Verilog, a much larger set of C and comes with a GUI that increases usability for users unfamiliar with formal verification tools. It seems to scale better and is faster than SmacC. Some results are listed in the Benchmark section.

EXE Tool

EXE is a bug finding tool that can generate input that crashes the program analyzed. EXE runs the code on symbolic input and constrains it by tracking constraints on symbolic memory locations. If symbolic values occur in a statement, EXE does not run the statement but adds it as an input-constraint. Symbolic memory locations need to be marked by the programmer.
If code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. When a path terminates or hits a bug, EXE automatically generates a test case by solving the constraints to find concrete values using the constraint solver STP. Supplying the concrete input to the original program will cause it to follow the same path and hit the same bug.
EXE scales well on real code, it was used to find bugs in various software projects, including the BSD and Linux packet filter implementations or the udhcpd DHCP server. SmacC's path generation algorithm was inspired by EXE, but instead of forking, paths are extracted from the program. One of the drawbacks of EXE seems to be the manual marking of symbolic input, which is not necessary in CBMC or SmacC [7].

Chapter 5

Results

5.1 Benchmarks

The following C files and algorithms were transformed to a BTOR representation, and can be used as benchmarks:

- Memcopy: A simple memcopy 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 memcopy 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.

- Tiny Encryption Algorithm:

5.1.1 Memcpy

To create a benchmark from the `memcopy` implementation it is necessary to supply a bound for source and destination array that is also used as bound for loop unrolling. The check that is used as benchmark is the last assertion in the code, which checks that all bytes below the bound were copied correctly, that is, elements at position `i` (less than bound `n`) in the destination buffer are equal to element at position `i` in the source buffer.
The next listing shows the code for `memcopy` with bound 40, note that variable `i` is constrained to be less than `n` in line 23.
The benchmark is created by executing: `./smacc -if memcpy_40.c -no -bk 40 -fp -cb` and extracting the BTOR expression for the last assertion check. `--first-path` can be used to minimize output, because the first path through the program contains the assertion statement in line 24. `-no` reduces output even further because overflow checks are omitted.

```
#include <assert.h>
void
main ()
{
  unsigned i;
  char * q;
  char * eos;
  char * p;
  char dst[40];
  char src[40];
  unsigned n = 40;

  p = src;
  q = dst;
  for (eos = src + n; p < eos; p = p + 1)
    {
      *q = *p;
      q = q + 1;
    }

  if (i < n)
    assert (src[i] == dst[i]);
}
```

Listing 5.1: Memcpy code - problem size 40

5.1.2 Palindrome

As for `memcpy` it is necessary to supply both array size and bound for loop unrolling to create a benchmark.
A string is considered to be a palindrome if it can be read the same way in either direction.
The algorithm is formulated in such a way, that the assertion being checked must hold on the first path through the program.
In line 25 variable `m` is constrained to be greater than zero but smaller than bound `n` - 1, line 27 constrains the string to be a palindrome on the first path through the program. The assertion in line 28 will checks if the character with the highest index is similar to the character with the lowest index, etc., until all characters are verified.
Note that actually the implementation compares groups of 4 characters to each other, because the input string is formulated as integer array, each byte of every 4-byte integer represents a character.
The following listing presents the code for a palindrome check of a 10 byte integer array (40 character string). To create a benchmark from the code, execute it with `./smacc -if palindrome_10.c -no -bk 10 -fp -cb` and extract BTOR expressions for the assertion check.

```
#include <assert.h>
int main ()
{
  int str[10], * begin, * end, n = 10, i, m, result = 0;

  i = n;
  begin = &str[0];
  end = &str[n - 1];

  for (i = n - 1; i >= 0; i = i - 1)
    {
      if (*begin == *end)
        result = result + 0;
      else
        result = 1;
      begin = begin + 1;
      end = end - 1;
    }

  if (m <= n - 1 && m >= 0)
    if (result != 1)
      assert (str[m] == str[n - m - 1]);
}
```

Listing 5.2: Palindrome code - problem size 10

5.1.3 Stringcopy

Copies the string stored in `src` to `dst`. The loop stops if the string terminating null-byte was read in `src`. After copying it to `dst` it is verified that all bytes copied are the same in both `src` and `dst` buffer and that the last byte in `dst` terminates the string.
What is special about this example is that the bound for the loop can be specified with the `-bk` parameter (because the loop condition does not use `size`.
After executing the last `if` statement of the unrolled loop, the negation of the loop condition is added as assumption, hence it holds that `*source` contains `'\0'` after executing the loop.
The assertion states that bytes from index 0 to index `source - orig_source` (the number of bytes copied) are similar in `src` and `dst` and that the last byte written to `dst` is the null byte.

```
#include <assert.h>

char src[4], dst[4];
unsigned size = 4, chk;

void
strcpy ()
{
  char * orig_target, * orig_src;
  char * target, * source;

  if (size > 0)
    {
      source = src;
      target = dst;
      orig_src = source;

      for (source = source; *source != 0; source = source + 1)
        {
          *target = *source;
          target = target + 1;
        }

      *target = 0;

      if (chk < size && chk < (source - orig_src))
        assert (dst[chk] == src[chk] && dst[(source - orig_src)] == 0);
    }
}
```

Listing 5.3: Stringcopy code - problem size 4

5.2 Timing Results

Benchmarks were run on an Intel® Core™ 2 Duo CPU E6750 @ 2.66GHz with 2GB main memory. Time was measured using the UNIX *time* command. Runtime for SmacC also contains the runtime of `if` condition checks which is only a small fraction (for example: 25 seconds, roughly 0.5%, for condition checks in memcpy40). Runtimes greater than 10 hours are considered timeouts.

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 10	10	734s	2010s	0.15s
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
tiny encryption	32	timeout	timeout	timeout
power 3 equality	3	timeout	timeout	timeout

Table 5.1: Comparing Boolector stand-alone version to library usage in SmacC.

It can be seen that runtime for the library version of Boolector exceeds the runtime of the stand-alone version many times over. Tiny encryption algorithm and power of 3 equality seem to produce hard instances. The following table lists the number of variables and clauses of CBMC SAT instances for all examples:

Benchmark	**variables**	**clauses**
memcpy.c, array size 30	12893	19522
memcpy.c, array size 40	22089	32517
memcpy.c, array size 50	33571	48834
palindrome check, n 10	10579	30221
palindrome check, n 11	11977	34899
palindrome check, n 15	18415	57040
palindrome check, n 16	20308	63322
strcpy array, n 20	601	367
strcpy array, n 30	795	692
strcpy array, n 40	1113	1117
tiny encryption	92284	322283
power 3 equality	15586	52637

Table 5.2: Size of SAT instances generated by CBMC.

Chapter 6

Tutorial

This chapter shows how to use SmacC either to verify small programs or to generate benchmarks.
The first part of the tutorial shows how examples can be constructed to be checked with SmacC, the second part will describe how to control output of SmacC by supplying specific command line arguments and how to interpret the output reported.

6.1 Installation and Requirements

SmacC requires a C compiler, preferably `gcc` and the build system `make`. Additionally the solver Boolector in its library version is required.

1. extract Boolector and SmacC archives: `tar xvfz smacc_vxx.tar.gz` and `tar xvfz boolector_vxx.tar.gz`, creating a boolector and a smacc directory

2. copy the Boolector library and header to the smacc directory `cp boolector/libboolector.a boolector/include/boolector.h smacc/`

3. change to smacc directory and compile SmacC: `cd smacc; make`

SmacC should now be ready to use, check by typing `./smacc -h`

6.2 Usage

Usage help for SmacC can be output by supplying the `-h` or `--help` argument to SmacC. A more detailed description of the command line arguments, when and how to use them, is presented in the tutorial section.

```
usage: smacc [<option> ...]

where <option> is one or more of the following:

  -h|--help      print this command line option summary
  -v|--verbosity <vl>   set verbosity level (0, 1, 2)
        default: vl = 1
  -if|--in-file <if>    input file to check
        default: if = stdin
  -er|--error-report <er> detail of memory check reports (0, 1)
        default: er = 1
  -bk|--bound-k <k>   bound for loop unrolling
        default: k = 4
  -si|--sequential-if   unroll loops to sequential if statements
        default: nested if statements
  -mc|--mem-check <cl>    memory analysis method (0, 1, 2, 3)
        default: cl = 3 (full, 2:deep, 1:basic, 0:none)
  -rv|--return-value <rv> expected return value
        default: rv = 0
  -fr|--force-return    use check
        'return value of program is always equal to rv'
        instead of
        'return value of program can be equal to rv'
  -fp|--first-path    stop after generation of first path
        default: generate all paths
  -no|--no-overflow   don't check overflow on arithmetic operations
        default: check overflow on arithmetic operations
  -na|--no-assertions   don't verify assertion statements
        default: verify assertion statements
  -nr|--no-return   don't check return statements
        default: check return statements
  -nd|--no-divzero    don't check for division by zero
        default: check division for division by zero
  -cb|--create-benchmarks boolector_dump instead of boolector_sat
        default: boolector_sat
  -ds|--dump-smt    dump in SMT format instead of BTOR format
        default: BTOR format
```

- `-v <vl>`: controls verbosity of SmacC. Level 2 prints the full code-list, the flattened code list and all paths executed. Level 1 omits the full and the flat code-list. Level 3 omits all output.

- `-if <if>`: specify the input file to check, if none is supplied standard input is used as input file.

- `-er <er>`: control detail of error messages.

- `-bk <k>`: the bound for loop unrolling

- `-si`: if supplied, loops are not unrolled to nested `if` statements but to a sequence of `if` statements.

- `-mc <cl>`: selects memory analysis method: 0 for no memory checks, 1 for abf memory check and 2 for deep memory check. When supplying 3, both memory analysis methods are used.

- `-rv <rv>`: specify a value against which return statements are checked.

- `-fr`: when a return statement is checked, the default check executed verifies that the return value can be equal to supplied return value. When supplying `-fr` a return statement check fails if a return value different from the supplied one is possible.

- `-fp`: only the path that is extracted as the first path through the program is symbolically executed.

- `-no`: if supplied, arithmetic operations are not checked for arithmetic overflow.

- `-na`: if supplied, assertion statements are not verified.

- `-nr`: is supplied, return statements are not checked for their return value

- `-nd`: if supplied, division operators are not checked for division by zero.

- `-cb`: benchmarking mode, instead of checking satisfiability of check formulas the formulas are conjuncted with all memory assumptions and path conditions and then dumped to a file.

- `-ds`: dump formulas in SMT format instead of BTOR format.

6.3 Creating a simple Example

Consider the following simple snippet of C code in a file named `test.c`:

```
int
main (void)
{
  unsigned x;

  if (x == 0 || x == 1)
    return 1;

  if (x >= 9)
    return 9;

  return 0;
}
```

Listing 6.1: File test.c

SmacC is run on the example by entering `./smacc -v 2 -if test.c`, and outputs:

```
original codelist:
CSENTER @ (2,6)
  CSENTER @ (3,0)
   CDECLL @ (4,12)
   CIF @ (7,4)
    CBBEG @ (7,4)
     CRET @ (7,12)
    CBEND @ (9,2)
   CIF @ (10,4)
    CBBEG @ (10,4)
     CRET @ (10,12)
    CBEND @ (12,2)
   CRET @ (12,10)
  CSEXIT @ (13,0)
CSEXIT @ (13,0)

flattened codelist:
CSENTER @ (2,6)
  CSENTER @ (3,0)
   CDECLL @ (4,12)
   CIF @ (7,4)
    CBBEG @ (7,4)
     CRET @ (7,12)
    CBEND @ (9,2)
   CIF @ (10,4)
```

```
      CBBEG @ (10,4)
       CRET @ (10,12)
      CBEND @ (12,2)
    CRET @ (12,10)
   CSEXIT @ (13,0)
CSEXIT @ (13,0)
```

As there are no loops in the example, flattened code-list is similar to the original full code-list. They are printed because verbosity level 2 was specified by `-v 2`. The first and second path are executed:

```
path:
CSENTER @ (2,6)
  CSENTER @ (3,0)
    CDECLL @ (4,12)
    CIF @ (7,4)
      CBBEG @ (7,4)
       CRET @ (7,12)
      CBEND @ (9,2)
  CBEND @ (12,2)
CSEXIT @ (13,0)
CSEXIT @ (13,0)

[IF] @ (7,4)

[CHECK] return statement @ (7,12) ERROR

path:
CSENTER @ (2,6)
  CSENTER @ (3,0)
    CDECLL @ (4,12)
    CELSE @ (9,2)
    CIF @ (10,4)
      CBBEG @ (10,4)
       CRET @ (10,12)
      CBEND @ (12,2)
  CSEXIT @ (13,0)
CSEXIT @ (13,0)

[ELSE] @ (9,2)

[IF] @ (10,4)

[CHECK] return statement @ (10,12) ERROR
```

The first path assumes the `if` condition in line 10 to be true and then checks the return statement `return 1`, which fails, because it is not satisfiable that the program returns 0 on this path. The coordinates in a path always consider the last character before the next statement, hence coordinates for the `if` statement are $(10, 4)$.
The second path assumes the first `if` statement to evaluate to false, but the second one to be true. Again, the return statement of this path cannot evaluate to 0, hence an error is output for the return statement in line 12.

```
path:
CSENTER @ (2,6)
  CSENTER @ (3,0)
    CDECLL @ (4,12)
    CELSE @ (9,2)
    CELSE @ (12,2)
    CRET @ (12,10)
  CSEXIT @ (13,0)
CSEXIT @ (13,0)

[ELSE] @ (9,2)

[ELSE] @ (12,2)

[CHECK] return statement @ (12,10) OK


btor refs before deletion: 0
sat calls: 8
time: 0.020000
time sat: 0.020000
% sat: 1.000000
```

The third path assumes the first two if statements to be unsatisfiable and returns 0, hence the return statement check for this path succeeds.
Because `-v 2` was supplied, statistics about the run are printed. The first number represents the number of `boolector_sat` calls, time represents the time passed since starting the program. The third number measures the time that passes when calling `boolector_sat`, and the fourth number outputs the percentage of time spent in `boolector_sat`.

Change `test.c` to contain the following code:

```
int
main (void)
{
  unsigned x;

  if (x == 0 || x == 1)
    return 1;

  if (x >= 9)
    return 9;

  assert (x > 1 && x < 9);
  return (x > 1 && x < 9);
}
```

Listing 6.2: Assertion statement added and return statement updated

The first two paths do not change, but the third path contains an additional assertion check and a different return statement expression.
Call SmacC on the example by entering `./smacc -if test.c -rv 1 -fr`. The last two parameters change the behaviour of SmacC when checking return statements: `-rv 1` does not use the default value 0 for checking return statements but value 1. `-fr` requires return statements to return the specified value in all cases, it must be unsatisfiable that the return statement returns a value different from the supplied one.
Consider the output for the third path:

```
path:
CSENTER @ (2,6)
  CSENTER @ (3,0)
    CDECLL @ (4,12)
    CELSE @ (9,2)
    CELSE @ (12,2)
    CASSERT @ (12,25)
    CRET @ (13,25)
  CSEXIT @ (14,0)
CSEXIT @ (14,0)

[ELSE] @ (9,2)

[ELSE] @ (12,2)

[CHECK] assertion violation @ (12,25) OK

[CHECK] return statement @ (13,25) OK
```

Again, both `if` conditions are assumed to evaluate to false. The assertion statement is checked. Checking an assertion requires SmacC to check if the value resulting from the expression asserted can evaluate to 0, if so, then the assertion might fail. In the example the assertion cannot fail because the two if statements that were not executed constrain the value of `x` to be between 2 and 8 (inclusive).
The same expression that was used in the assertion statement is used as return value. It can never evaluate to zero, as was proven when checking the assertion statement, but will always evaluate to 1, hence the return statement check succeeds.
Change `test.c` again:

```
int
main (void)
{
  int * p;
  unsigned x;

  if (x == 0 || x == 1)
    return 1;

  if (x >= 9)
    return 9;

  assert (x > 1 && x < 9);

  p = (int *) malloc (x);
  *p = 0;
  return 0;
}
```

Listing 6.3: Allocate and write to memory

Symbolically execute it: `./smacc -if test.c`, again considering only the third path.

```
path:
CSENTER @ (6,6)
  CSENTER @ (7,0)
    CDECLL @ (8,9)
    CDECLL @ (9,12)
    CELSE @ (14,2)
    CELSE @ (17,2)
    CASSERT @ (17,25)
    CCODE @ (19,24)
    CCODE @ (20,8)
    CRET @ (21,10)
  CSEXIT @ (33,0)
CSEXIT @ (33,0)
```

```
[ELSE] @ (14,2)

[ELSE] @ (17,2)

[CHECK] assertion violation @ (17,25) OK

[CHECK] legal assignment, abf @ (20,8) ERROR
  00000000000000000000000000000000  result
  00000000000000000000000000000000  const
  01000100000111111111111111111000  p
  01000010111111111111111111111100   *

[CHECK] return statement @ (21,10) OK
```

Again, the assertion statement holds, but the assignment statement `*p = 0` results in a failing memory model check. The assignment could target an invalid address because writing an `int` value requires 4 bytes and the call to `malloc` could allocate only 2 or 3 bytes.

Changing `test.c` to

```
int
main (void)
{
  int * p;
  unsigned x;

  if (x == 0 || x == 1)
    return 1;

  if (x >= 9)
    return 9;

  assert (x > 1 && x < 9);

  p = (int *) malloc (x * sizeof (int));
  *p = 0;
  return 0;
}
```

Listing 6.4: Fixed memory allocation

corrects the error.

```
path:
CSENTER @ (6,6)
  CSENTER @ (7,0)
   CDECLL @ (8,9)
   CDECLL @ (9,12)
   CELSE @ (14,2)
   CELSE @ (17,2)
   CASSERT @ (17,25)
   CCODE @ (19,39)
   CCODE @ (20,8)
   CRET @ (21,10)
  CSEXIT @ (33,0)
CSEXIT @ (33,0)

[ELSE] @ (14,2)

[ELSE] @ (17,2)

[CHECK] assertion violation @ (17,25) OK

[CHECK] signed multiplication overflow @ (19,39) OK

[CHECK] legal assignment, abf @ (20,8) OK

[CHECK] legal assignment, oob @ (20,8)  OK

[CHECK] return statement @ (21,10) OK
```

If no error is found for an assignment or indirection then also a deep memory check is issued. This behaviour can be changed by supplying the `-mc` argument. A value of 0 disables all memory checks, 1 selects basic memory check, 2 selects deep memory check and a value of 3 (the default behaviour) was used in the example above.

It is also possible to disable different checks, consider again path 3 of the program executed with arguments `./smacc -if test.c -na -no -nr`:

```
CSENTER @ (6,6)
  CSENTER @ (7,0)
   CDECLL @ (8,9)
   CDECLL @ (9,12)
   CELSE @ (14,2)
   CELSE @ (17,2)
   CASSERT @ (17,25)
   CCODE @ (19,39)
   CCODE @ (20,8)
   CRET @ (21,10)
```

```
   CSEXIT @ (33,0)
CSEXIT @ (33,0)

[ELSE] @ (14,2)

[ELSE] @ (17,2)

[CHECK] legal assignment, abf @ (20,8) OK

[CHECK] legal assignment, oob @ (20,8)  OK
```

Path condition checks can not be disabled but they are not executed when the benchmark creation argument is supplied: `./smacc -if test.c -cb`. More paths will be generated because unsatisfiable path conditions are not detected when using `-bc`. The output is not listed because the BTOR instances for the checks are between 50 and 400 lines per instance.

6.4 Analyzing Output

As final example consider again a variation of the example from [7].

```
main (void)
{
  unsigned a[4];
  unsigned i, t;
  char * p;
  a[0] = 1;
  a[1] = 3;
  a[2] = 5;
  a[3] = 2;

  if (i >= 4)
    return;

  p = (char *) a + i * 4;
  *p = *p - 1;

  t = a[*p];

  t = t / a[i];

  if (t == 2)
    assert (i == 1);
  else
    assert (i == 3);
}
```

Listing 6.5: Variation of the EXE example

Consider path 2 through the program where the `if` condition in line 12 of the program does not hold:

```
path:
CSENTER @ (2,6)
  CSENTER @ (3,0)
   CDECLL @ (4,15)
   CDECLL @ (5,12)
   CDECLL @ (5,15)
   CDECLL @ (6,10)
   CCODE @ (7,10)
   CCODE @ (8,10)
   CCODE @ (9,10)
   CCODE @ (10,10)
   CELSE @ (15,2)
```

```
    CCODE @ (15,24)
    CCODE @ (16,13)
    CCODE @ (18,11)
    CCODE @ (20,14)
     CIF @ (23,4)
      CBBEG @ (23,4)
       CASSERT @ (23,19)
      CBEND @ (24,2)
   CSEXIT @ (26,0)
CSEXIT @ (26,0)
```

`CELSE @ (15,2)` constrains `i` to be less than 4 because the assumption `!(i >= 4)` is added to the list of path conditions. The following `CCODE` entries represent the assignment statements in lines 15 to 20 in the program.
Omitting the rest of the checks, consider the following two errors reported by SmacC:

```
[CHECK] legal indirection, abf @ (18,11)  ERROR
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  result
  11011101011001111111000000000000  a
  11011101011001111111011111110100  p
  11011101011001111111000000001000   *
  00000100    *
  00000000000000000000000000000100  const
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx   *

[CHECK] unsigned division by zero @ (20,14) ERROR
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  result
  01110000000010101111111111111000  a
  01110000000010101111111111110100  i
  00000000000000000000000000000000   *
  00000000000000000000000000000100  const
  00000000000000000000000000000000   *
  01110000000010101111111111110000  t
  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx   *
```

The memory error reported at position `18,11` represents the assignment in line 18 in the program. At this state in the program `i` has a value between 0 and 3 therefor `p` points to value 1, 3, 5 or 2 which gets decremented by 1. The problem with the assignment is that `p` might point to the value 4, then the array access on the right hand side of the statement is invalid. The output lists addresses of variables, their values and constants if they occur in the expression. `11011101011001111111011111110100` is the address of `p`, `11011101011001111111000000001000` below `p`, marked with `*`, represents the value stored at `p` (`*` represents indirection in C, hence it is also used to mark values of variables in the output). `00000100  *` represents the character value `p` points to, in this case 4. Because `i` is not used in the statement its value is not shown in the output

but one can deduce that `i` had the value 2.
The next error reported represents an error on an arithmetic operation. If `p` points to a byte of the first element of array `a` value 1 is changed to 0 in line 16, resulting in division by zero in line 20. Because `i` is used in the expression is value is listed in the output.
The next error

```
[CHECK] assertion violation @ (23,19) ERROR
  00000000000000000000000000000000  result
  00000000000000000000000000000000  const
  00000000000000000000000000000001  const
  00000000000000000000000000000001  const
  11000000000000000000000000001011  i
  00000000000000000000000000000010   *
```

represents a failing assertion statement but one recognizes in the output that i has the value 4 which already caused the memory error in line 18.
After modifying the if condition in line 22, prohibiting value 2 for `i`,

```
if (t == 2 && i != 2)
```

the assertion in line 23 holds:

```
[CHECK] assertion violation @ (23,19) OK
```

This concludes the tutorial. After reading both the theoretic parts about the memory model, checks and limitations, one should be able to use and understand SmacC.

Appendix A

Supported C Subset

Translation Units

In the C language described by the draft submitted to ANSI on 31 October 1988, a program consists of one or more translation units stored in files [16].
SmacC does not support more than one translation unit, only one input file is supplied and it must not refer to code in other translation units.

Comments

In addition to the comment characters `/*` and `*/` supported by ANSI C the `//` comment characters are supported in translation units valid as SmacC input.

Constants

In ANSI C there exist several kinds of constants, all belonging to a certain type discussed in the type section.

Support for different forms of integer and character constants would require the front-end to be updated but would not need additional type support in the back-end.
Enumeration constants would need changes to support the declaration of enumeration types in the front-end. Identifiers declared as enumerators are constants of type integer and would require no changes in the back-end. Floating constants needs a decision

ANSI-C constant	**form**	**Supported in SmacC**
integer-constants	-	supported
	`0` octal	unsupported
	`0x`, `0X` hexadecimal	unsupported
	`u`, `U` unsigned	unsupported
	`l`, `L` long	unsupported
character-constants	”	unsupported
	””	unsupported
	decimal representation	supported
floating-constant	-	unsupported
enumeration-constants	-	unsupported

Table A.1: Supported constants

procedure that supports floating types which Boolector does not. Additionally changes would be necessary both in the front-end and back-end.

Storage Classes

An object denotes a named region of storage in memory, an lvalue is an expression referring to an object [16].

The storage class of an object is specified by several keywords together with the context of its declaration. Automatic objects are local to a block and invalidated when the block is exited.
Declarations within a block create automatic objects. Static objects can be local to a block or external to all blocks but retain their values throughout the program.
Objects declared outside all blocks are always static. These basic rules for objects do not always apply when declarations contain additional keywords:

- `register`: Automatic, should be stored in registers of the CPU if possible.
- `auto`: Automatic, force declaration to be automatic storage.
- `static`: Internal linkage, local to a particular translation unit.
- `external`: External linkage, global to the entire program.

In SmacC storage class specifiers are unsupported.

Basic Types

The type system supported by SmacC inludes not all basic types supported by the ANSI C standard.
The table lists keywords for the basic types in C and their representation in SmacC.

Basic Type	**Supported in SmacC**	**Size** (default)
`char`, `short`	supported	8 Bit
`unsigned char`, `unsigned short`	supported	8 Bit
`int`, `long`	supported	32 Bit (signed)
`unsigned int`, `unsigned`, `unsigned long`	supported	32 Bit
`float`	unsupported	-
`double`	unsupported	-
`enumerations`	unsupported	-
`void`	supported	0 Bit

Table A.2: Supported and unsupported types

As for the corresponding constants, `enumeration` types could be implemented very easy, support for floating types would require changes in the SMT solver or heavy changes in the back-end to model floating point types with types supported by the back-end.

Derived Types

Derived types can be constructed in addition to the basic types by using the following methods:

Method	**Description**	**Supported in SmacC**
`arrays`	Arrays of objects of a given type	supported
`functions`	Return objects of a given type	partly supported
`pointers`	Pointers to objects of a given type	partly supported
`structures`	Contain a sequence of objects of types	unsupported
`unions`	Contain any one of several objects of types	unsupported

Table A.3: Supported and unsupported methods for deriving types

Function pointers are not supported.

`unions` and `structures` are not supported but could be implemented with changes in both front- and back-end.

Function Calls

Functions are only partly supported, a translation unit may only contain one function declaration but not more.
Some function calls are supported as they are implemented in SmacC:

- `assert`: Asserts an expression.
- `malloc`: Allocates memory.
- `free`: Deallocates memory.

Type Qualifiers

Both type qualifiers `volatile` and `const` are not supported by SmacC.

Integral Promotion

Integral types may be used in expression wherever an integer may be used. If an `int` can represent all the values of the type, the value is converted to `int`. Otherwise it is converted to `unsigned`.

Primary Expressions

Primary Expression	**SmacC**
identifier	supported
constant	partly supported check table A.1
string	unsupported
(expression)	supported

Table A.4: Supported primary expressions

Postfix Expression	**Description**	**SmacC**
primary-expression	check table A.4	partly supported
postfix-expression `[` expression `]`	array reference	supported
postfix-expression `(` argument-expression-list `)`	function call	partly supported
postfix-expression `.` identifier	structure reference	unsupported
postfix-expression `->` identifier	structure reference	unsupported
postfix-expression `++`	postfix increment	unsupported
postfix-expression `--`	postfix decrement	unsupported

Table A.5: Supported postfix expressions

Postfix Expressions

Unary Expressions

Unary Expression	**Description**	**SmacC**
postfix-expression	check Table A.5	partly supported
`++` unary-expression	prefix increment	unsupported
`--` unary-expression	prefix decrement	unsupported
unary-operator cast-expression	cast, type conversion	supported
`sizeof` unary-expression	sizeof expression in bytes	supported
`sizeof (` type-name `)`	sizeof type in bytes	supported

Table A.6: Supported unary expressions

Unary-operators are one of the following:

Unary Operator	**Description**	**SmacC**
`&`	address operator	supported
`*`	indirection operator	supported
`+`	unary plus operator	supported
`-`	unary minus operator	supported
`~`	one's complement operator	supported
`!`	logical negation operator	supported

Table A.7: Supported unary operators

Multiplicative and Additive Operators

Multiplicative and Additive operators with their usual semantics are supported by SmacC.

Shift Operators

Shift operators are supported.

Relational and Equality Operators

The operators evaluate to 0 if the specified relation is false and to 1 if it is true.

Operator	**Description**	**SmacC**
`<`	less	supported
`>`	greater	supported
`<=`	less or equal	supported
`>=`	greater or equal	supported
`==`	equal to	supported
`!=`	not equal to	supported

Table A.8: Supported relational and equality operators

Bitwise and Logical Operators

Operator	**Description**	**SmacC**
`&`	bitwise and function	supported
`^`	bitwise exclusive or function	supported
`|`	bitwise inclusive or function	supported
`&&`	logical and operator	supported
`||`	logical or operator	supported
`?:`	conditional operator	supported

Table A.9: Supported bitwise and logical operators

Assignment Expressions

Only the non-augmented assignment operator is supported by SmacC, augmented assignment is not possible.

Operator	Description	SmacC
`=`	non-augmented assignment	supported
`*=`, `/=`,`%=`,`+=`,`-=`,`<<=`,`>>=`, `&=`, `^=`	augmented assignments	unsupported

Table A.10: Assignment operators

Comma Operator

The comma operator is supported for variable declarations, initializations and in `for` statements.

Statements

The following statements are described in the ANSI-C reference manual in [16].

Statement	Description	SmacC
labeled-statement	a statement may carry a label	unsupported
expression-statement	assignments, function calls	partly supported
compound-statement	sequence of statements, block	supported
selection-statement	`if`, `switch`	only `if` supported
iteration-statement	`while`, `do-while`, `for`	only `for` supported
jump-statement	`goto`, `continue`, `break`, `return`	only `return` supported

Table A.11: Statement types

Labeled statements and jump statements that jump back in the code require changes in the back-end, especially the `goto` jump statement.
Selection statement `switch` could be realized by transforming them into if statements, as could the `while` and `do-while` iteration statement.
The condition expression of `for` statements is mandatory and must not be omitted.

Preprocessor and Macros

The only preprocessor directive that is supported is the `#include` directive.
`#include` has no effect on the SMT formula generated, it is only supported to be able to run examples that compile with gcc with SmacC, most important is the `#include <assert.h>` directive.

Bibliography

[1] Jean-Michel Berge, Alain Fonkoua, Serge Maginot, and Jacques Rouillard. *VHDL Designer's Reference*. Kluwer Academic Publishers, Dordrecht, Boston, London, 1992.

[2] Armin Biere. The aiger and-inverter graph (aig) format. Available at http://fmv.jku.at/aiger/FORMAT.aiger, 2006-2007.

[3] Robert Brummayer. Boolector documentation. Available with the solver at http://fmv.jku.at/boolector/, 2009.

[4] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. 2008.

[5] 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.

[6] Robert Brummayer, Armin Biere, and Florian Lonsing. Btor: Bit-precise modelling of word-level problems for model checking. In *BTOR: Bit-precise mod-elling of word-level problems for model checking*, 2008.

[7] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death, 2006.

[8] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. Carnegie Mellon University, 2004.

[9] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. 2003.

[10] CProver. The cprover user manual. Available at http://www.cprover.org/cbmc/doc/manual.pdf.

[11] CVC3. Cvc3 user's manual. Available at www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html, July, 30, 2009.

[12] Niklas Een and Niklas Soerensson. Temporal induction by incremental sat solving. Chalmers University of Technology.

[13] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, 1995.

[14] Jr. Henry S. Warren. *Hacker's Delight*. Addision-Wesley, 7th Printing, New York, September 2007.

[15] Holger H. Hoos and Thomas Stuetzle. Satlib: An online resource for research on sat. Available at http://www.cs.ubc.ca/ hoos/Publ/sat2000-satlib.pdf, 2000.

[16] Brian W. Kerninghan and Dennis M. Ritchie. *The ANSI C Programming Language, 2nd Edition*. Prentice Hall Software Series, 41st Printing, Murray Hill, New Jersey, December 2006.

[17] Daniel Kroening. Bounded model checking for ansi-c. Available at http://www.cprover.org/cbmc/.

[18] Daniel Kroening and Ofer Strichman. *Decision Procedures, An Algorithmic Point of View*. Springer-Verlag, Berlin-Heidelberg, 2008.

[19] Microsoft. Smt 2008 6th international workshop on satisfiability modulo theories. Princeton, USA, July 2008.

[20] Silvio Ranise and Cesare Tinelli. The smt-lib standard: Version 1.2. Available at http://combination.cs.uiowa.edu/smtlib/, August, 30, 2006.

[21] John C. Reynolds. Separation logic: A logic for shared mutable data structures. Carnegie Mellon University, 2003.

[22] Niklas Soerensson. Efficient sat solving. Chalmers University of Technology and University of Gothenburg, 2008.

[23] William Stallings. *Operating Systems - Internals and Design Principles, 6th Edition*. Pearson Prentice Hall, 2008.

[24] Andrew S. Tannenbaum. *Modern Operating Systems, 3rd Edition.* Pearson, Prentice Hall, Upper Saddle River, New Jersey 07458, 2007.

[25] Jakob Zwirchmayr. Cvc2baf - converting cvc format to boolector ascii format, 2007.