Methods of Handling State Space Explosion in Model Checking

Jannik Hiller
jphiller@ucdavis.edu
University of California, Davis
Davis, California, USA
jannik.hiller@rwth-aachen.de
RWTH Aachen University
Aachen, Germany

ACM Reference Format:
Jannik Hiller. 2018. Methods of Handling State Space Explosion in Model Checking. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Initially, model checking algorithms used explicit representations (e.g., adjacency lists) for their transition systems[2]. This approach quickly became problematic due to state space explosion in many real-world problems. For example, Figure 1 shows a program where 685 distinct states are reachable in 50 steps. Whereas with the code in Figure 2, there are already 3535 reachable states. In this case, the size of the transition system is exponential in the size of the program.

To address this problem, symbolic representations of the transition system are used. Symbolic representations of the transition system, such as SAT/SMT approaches, use variables to represent possible states and constrain them with a transition relation formula T. The size of this formula is not exponential because each line of the program only needs a single conjunction in T. This conjunction groups all transitions (with all possible variable values) from one code location to the next.

In this project, I explore two ways of creating such a transition relation. It is based on the following papers:

(1) *Symbolic model checking without BDDs* [1]. This paper uses propositional logic to represent transition systems, which can then be checked with SAT solvers.
(2) *SMT-Based Bounded Model Checking for Embedded ANSI-C Software* [3]. This paper uses a subset of first-order logic to represent transition systems and utilizes SMT solvers.

1.1 Overview of Project Files

The project consists of the following files:

- **while_parsing.py**: Implements a parser and a runtime for the WHILE programming language. The parser translates WHILE code into GOTO-like code to simplify the SAT/SMT encoding.
- **transition_system.py**: Directly unrolls WHILE programs, storing the entire (exponential) transition system. This is used for testing and investigating the growth of these systems.
- **transition_relation.py**: Creates a Z3 formula for the transition relation, irrespective of the chosen encoding. The SAT/SMT specific parts are factored out into separate files.
- **smt.py**: Contains SMT-specific code, primarily a wrapper for z3.Int and mappings from WHILE operators to Z3 functions.
- **sat.py**: Contains SAT-specific code, which is more complex. It uses a tuple of z3.Bools as the base type and implements

```
1  INPUT x
2  INPUT y
3  WHILE a < x DO
4    a := a + 1
5    WHILE b < y DO
6      b := b + 1
7    END WHILE
8  END WHILE
```

Figure 1: WHILE program with two nested loops. 685 distinct states are reachable in 50 steps.

```
1  INPUT x
2  INPUT y
3  INPUT z
4  WHILE a < x DO
5    a := a + 1
6    WHILE b < y DO
7      b := b + 1
8      WHILE c < z DO
9        c := c + 1
10     END WHILE
11   END WHILE
12 END WHILE
```

Figure 2: WHILE program with three nested loops. 3535 distinct states are reachable in 50 steps.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

```
var = (* any string without whitespace that does not
        start with a number, keyword, or operator *);

expression = op, " ", {var | int}
           | (var | int), " ", infix_op, " ",
             (var | int);

comment = "\n" | "//", {char - "\n"}, "\n";

statement = var, " := ", expression, "\n"
          | "IF ", expression, " THEN\n",
            while_program,
            ["ELSE\n", while_program],
            "END IF\n"
          | "WHILE ", expression, " DO\n",
            while_program, "END WHILE\n"
          | "INPUT ", var
          | "OUTPUT ", expression
          | comment;

while_program = {statement};
```

Figure 3: EBNF Grammar of WHILE

encodings for "+" and binary comparison operators using bit operations ("bitblasting").

- **compare_encodings.py**: Driver code to display (and optionally save) encodings of WHILE programs.

2 WHILE PROGRAMMING LANGUAGE

The WHILE programming language used in this project is a simple language similar to other WHILE languages. It operates exclusively on integers and supports all basic arithmetic operations. Figure 3 shows its EBNF grammar, and Figure 4 shows an example program.

The WHILE language translates into a GOTO-like language with the following instructions:

`"SET_VAR"`, `"JUMP_IF_NOT"`, `"JUMP"`, `"INPUT"`, `"OUTPUT"`

2.1 Interactive Shell Mode

The language also supports an interactive shell mode. This is achieved by making everything lazy evaluated using Python's generator functions.

2.2 Variable Naming Restrictions

For SMT/SAT encoding, only lower-case letter variable names are supported.

3 SMT ENCODING

The transition relation T is created by iterating over all locations in the GOTO code and constraining two state variables to conform to the transition of the code location. For example, the statement `5: a := a + 5` with the state variables x and y becomes:

$$x_{\text{loc}} = 5 \wedge y_{\text{loc}} = 6 \wedge x_a + 5 = y_a \wedge x_b = y_b \wedge \dots$$

```
1  INPUT l
2  a := 0
3  b := 1
4  WHILE a < l DO
5    OUTPUT a
6    t := b
7    b := a + b
8    a := t
9  END WHILE
```

Figure 4: Example WHILE program that outputs the Fibonacci sequence up to a limit l.

If a variable is set by user input, its value is unknown. This is encoded using an additional boolean value. To correctly handle cases with an unknown variable in other expressions, multiple conjunctions are returned per location in the code.

To obtain the complete formula, the disjunction of all the returned conjunctions is taken.

4 SAT ENCODING

The SAT encoding works on the same principle as the SMT encoding, but instead of using integer variables, it uses a vector of bits to represent the value of a variable. This approach significantly increases the complexity, as constraints on the value of a variable must be expressed at the bit level. Additionally, each operator must be implemented individually, as opposed to using pre-defined Z3 functions. To manage the scope of the project, only addition and binary comparisons were implemented.

4.1 Addition

For addition, an additional bitvector holding the carry bits is created. Each result bit is constrained to be the XOR of the bits of the summands and the corresponding bit of the carry bitvector. This ensures the correct propagation of carry bits throughout the addition.

4.2 Less-than Comparison

For the less-than comparison, a disjunction of the following conditions is taken: for each bit i, all bits larger than i are equal, and the i-th bit of the left-hand side (lhs) is smaller than the i-th bit of the right-hand side (rhs). The other comparisons are based on the less-than operator.

5 FINDINGS

I evaluated the encodings on the following metrics:

- Implementation difficulty, measured in Lines of Code.
- Encoding size, measured in AST nodes.

For this, I created encodings of the following programs:

I 3 nested while loops (Figure 2)
II The Fibonacci sequence (Figure 4).
III An absolute value function (Figure 5)

Table 1: Comparison of Direct Unrolling, SMT, and SAT Encodings for Various Programs

	Direct unrolling (50 steps)	SMT	SAT
Lines of code	186	214 + 56 SMT specific	214 + 110 SAT specific
AST nodes / states I	3535	4048	43942
AST nodes / states II	59	2840	29724
AST nodes / states III	7	1847	not supported

```
INPUT x
IF x >= 0 THEN
    y := x
ELSE
    y := -1 * x
END IF
OUTPUT y
```

Figure 5: Absolute value program.

Table 1 shows the results. Out of interest, the number of reachable transitions with direct unrolling for 50 steps is added. It should be noted that this is not comparable, as the other encodings represent the whole transition system. For some of the WHILE programs, if we unrolled transition systems directly until, for example, all states with 16-bit variable values are reached, all the memory in the world would not be enough.

5.1 Discussion

The SAT encoding is roughly a factor of 10 larger than the SMT encoding. I expected this factor to be at least 16 because for every integer variable in the SMT encoding, there are 16 boolean variables in the SAT encoding. The smaller factor can likely be attributed to there being more nodes than expected higher up in the AST.

In terms of implementation difficulty, SAT is considerably more challenging. It should also be noted that the lines of code do not reflect that only a few operators have been implemented for SAT, whereas almost all WHILE operators work in SMT.

It is noteworthy that the encodings for program I are much larger than for program II despite their similar size. My explanation for this is that because of the nested WHILE loops, program I contains more `JUMP_IF_NOT` instructions. These result in four conjunctions when creating the encoding.

For program III, direct unrolling found all possible states (which are limited, as x is unknown).

REFERENCES

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without bdds. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer, 193–207.

[2] Edmund M Clarke and Qinsi Wang. 2014. 25 years of model checking. In *Ershov Memorial Conference*, 26–40.

[3] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. 2011. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38, 4, 957–974.