

Comparing SAT Encodings for Model Checking

Alan Frisch, Daniel Sheridan, and Toby Walsh

Artificial Intelligence Group,
Department of Computer Science,
University of York,
York, England
{frisch, djs,tw}@cs.york.ac.uk

Abstract. Model checking as a tool is widely used in both academia and industry. The traditional approach, computing using binary decision diagrams, has been researched in much depth. We will look at the newer approach of encoding the problem into SAT, and the resulting question of determining the efficiency of different encodings.

1 Model Checking: a brief introduction

Model checking is a method of verifying the behaviour of a dynamic system, comparing an implementation or *model* against a temporal logic *specification*. A typical application is in the field of hardware verification, where a model checking tool can assist in finding bugs in hardware designs long before a product enters the testing phase.

Bounded model checking (BMC) [1] was proposed as a solution to some of the problems of conventional BDD-based symbolic model checking such as space explosion by introducing a temporal bound. The problem can then be encoded as a Boolean formula and used as input to a SAT checker; the output of a BMC tool is a conjunction of state transition functions and state verification functions. Certain modal operators can be handled by including a check for loops in the state transitions.

While there has been some analysis of the run time of BMC tools, the available tool is limited in the specifications that it accepts, so there are few readily available benchmarks that run on BMC. Solving this problem brings to light the need for an improved encoding: the pathological case is omitted from the original tool, so does not show up in the original analysis. This approach is in contrast to the work of Shtrichman [5], who investigated the problem of improving the available SAT checkers to better deal with the problems produced by the present encodings.

With a variety of encodings available, it becomes necessary to perform a comparison between them. A first approach to this work was discussed in [4], where the number of clauses produced by the encoding is discussed. We will be extending this analysis here, but also looking at other methods for predicting the difficulty of a SAT problem.

2 Linear Time Logic

The work concentrates on the encoding of the specification, and to this end we include a brief summary of the temporal logic LTL. LTL is a convenient logic to use to write down the encoding, but typically the specification is given in a branching time logic such as CTL. A limitation of BMC is that it performs *existential* model checking; as the specification is negated before encoding (since success in solving a model checking problem is equivalent to finding a bug) this means that only *universal* specifications may be given. After negation, the existential quantifiers may be removed from the logic, leaving an LTL specification.

LTL specifies an event occurring in the future. Five operators are available in addition to the usual propositional ones, and are given various symbols depending on the context. We will use the symbols derived from modal logic here. The model of time used is one of sequential states indexed from zero. The operators are defined relative to the current position in time, as follows:

- $\bigcirc f$ holds if f holds in the next moment
- $\diamond f$ holds if f holds at some time in the future
- $\square f$ holds if f holds in all future moments
- $f \mathcal{U} g$ holds if f holds now and in all future moments until g holds.
- $f \mathcal{W} g$ holds if f holds now and in all future moment, or until g holds. Equivalent to $(f \mathcal{U} g) \vee \square f$

2.1 Computational Tree Logic

CTL extends LTL by introducing the universal and existential quantifiers **A** and **E** which appear directly before a temporal operator.

- **A** x holds if x holds in all possible future paths
- **E** x holds if there exists a future path in which x holds

3 Encodings

Introducing a bound in the number of states to be considered causes an apparent conflict between the finite nature of the checking procedure and the infinite nature of the possible specifications.

The model is given as, or converted to, a finite state transition diagram. We know from the pumping lemma that if any infinite path of states must contain a loop. By repeating the model checking for each possible loop, infinite properties such as \diamond can be refuted. This does not eliminate the problem altogether, as if a refutation is not found within the bound, then the system may falsely report a bug-free model. Choosing an appropriate bound is a difficult problem: heuristics which produce an overestimate exist, and are detailed in [1].

We use the notation $\llbracket f \rrbracket_k^i$ to denote the encoding of an LTL formula, f , with bound k at time i ; if a loop is to be checked, the notation ${}_i \llbracket f \rrbracket_k^i$ denotes the encoding with a loop from state k to state l . The notation $\llbracket M, f \rrbracket_k$ denotes the encoding of model M with specification f and bound k .

3.1 Biere's Encoding

Biere et al. [1] introduces the notion of bounded model checking and loops, and gives an encoding for each operator and a general encoding which we will use as a framework for later work. The general encoding, based on the definitions above is

$$\llbracket M, f \rrbracket_k := \llbracket M \rrbracket_k \wedge \left((\neg L_k \wedge \llbracket f \rrbracket_k^0) \vee \bigvee_{l=0}^k (L_k \wedge_l \llbracket f \rrbracket_k^l) \right) \quad (*)$$

The encoding for the specification is defined inductively on the operators. For example,

$$\begin{aligned} \llbracket \diamond f \rrbracket_k^i &:= \bigvee_{j=i}^k \llbracket f \rrbracket_k^j \\ \llbracket f \mathcal{W} g \rrbracket_k^i &:= \bigvee_{j=i}^k (\llbracket g \rrbracket_k^j \wedge \bigwedge_{n=i}^j \llbracket f \rrbracket_k^n) \\ \llbracket f \vee g \rrbracket_k^i &:= \llbracket f \rrbracket_k^i \vee \llbracket g \rrbracket_k^i \end{aligned}$$

3.2 Encoding via SNF_C

Separated normal form is a normal form for temporal logics, and SNF_C [2] is the normal form for CTL. A formula in SNF_C takes the form $\mathbf{A}\square \bigwedge_i (P_i \Rightarrow F_i)$, where each $P_i \Rightarrow F_i$ is of the form (slightly simplified from the paper)

$$\begin{aligned} \mathbf{start} &\Rightarrow \bigvee_{j=1}^k \beta_j && \text{an } \textit{initial} \text{ rule} \\ \bigvee_{i=1}^l \alpha_i &\Rightarrow \mathbf{A} \bigcirc \bigvee_{j=1}^k \beta_j && \text{a } \textit{global} \text{ } \mathbf{A}\square\text{-rule} \\ \bigvee_{i=1}^l \alpha_i &\Rightarrow \mathbf{E} \bigcirc \bigvee_{j=1}^k \beta_j \langle \text{ind} \rangle && \text{a } \textit{global} \text{ } \mathbf{E}\square\text{-rule} \\ \bigvee_{i=1}^l \alpha_i &\Rightarrow \mathbf{A} \diamond \gamma && \text{a } \textit{global} \text{ } \mathbf{A}\diamond\text{-rule} \\ \bigvee_{i=1}^l \alpha_i &\Rightarrow \mathbf{E} \diamond \gamma \langle \text{ind} \rangle && \text{a } \textit{global} \text{ } \mathbf{E}\diamond\text{-rule} \end{aligned}$$

α, β, γ are literals, which means that a formula in SNF_C has temporal operators nested to a depth of at most two. This is important, given the predominance of large conjunctions and disjunctions in the original encoding. The angle brackets indicate an index used to tie multiple existentials to the same path.

Where operators are nested, the nested formula is replaced with a new variable, and a new rule is introduced. For example,

$$P \Rightarrow \mathbf{E}\square (\mathbf{A}\diamond f) \langle i \rangle \rightarrow \begin{cases} P \Rightarrow \mathbf{E}\square x \langle i \rangle \\ x \Rightarrow \mathbf{A}\diamond f \end{cases}$$

The conversion of a general CTL expression to SNF_C is achieved by a series of rewrite rules; initially, a CTL expression in negation normal form c is written

ϕ	$p(\phi)$	$\bar{p}(\phi)$
$\neg\phi_1$	$\bar{p}(\phi_1)$	$p(\phi_1)$
$\phi_1 \wedge \phi_2$	$p(\phi_1) + p(\phi_2)$	$\bar{p}(\phi_1)\bar{p}(\phi_2)$
$\bigwedge_{i=a}^b \phi_1$	$\sum_{i=a}^b p(\phi_1)$	$\prod_{i=a}^b \bar{p}(\phi_1)$
$\phi_1 \vee \phi_2$	$p(\phi_1)p(\phi_2)$	$\bar{p}(\phi_1) + \bar{p}(\phi_2)$
$\bigvee_{i=a}^b \phi_1$	$\prod_{i=a}^b p(\phi_1)$	$\sum_{i=a}^b \bar{p}(\phi_1)$
$\phi_1 \Rightarrow \phi_2$	$\bar{p}(\phi_1)p(\phi_2)$	$p(\phi_1) + \bar{p}(\phi_2)$
$\phi_1 \Leftrightarrow \phi_2$	$p(\phi_1)\bar{p}(\phi_2) + \bar{p}(\phi_1)p(\phi_2)$	$p(\phi_1)p(\phi_2) + \bar{p}(\phi_1)\bar{p}(\phi_2)$

Table 1. $p(\phi)$: the number of clauses produced

$\mathbf{A}\square \bigwedge_i (\mathbf{start} \Rightarrow c)$, and transformations based on the fixed point characterisations are applied (see [2] for more details).

To encode a specification using SNF_C , we negate the formula and convert using the usual transformation rules. The outer $\mathbf{A}\square$ may be distributed across the conjunction, giving a conjunction of rules of the form $\mathbf{A}\square(P \Rightarrow F)$. We can give an encoding for each of these rules as below. The indication of the starting state is omitted as it is implicitly zero, there being no possibility of nested rules. The encodings are given in terms of the original encodings for purely propositional formulæ.

$$\begin{aligned}
\llbracket \mathbf{A}\square(\mathbf{start} \Rightarrow \mathbf{E}\bigcirc f \langle L \rangle) \rrbracket_k &:= \llbracket f \langle L \rangle \rrbracket_k^1 \\
\llbracket \mathbf{A}\square(\mathbf{start} \Rightarrow \mathbf{E}\diamond f \langle L \rangle) \rrbracket_k &:= \bigvee_{j=0}^k \llbracket f \langle L \rangle \rrbracket_k^j \\
\llbracket \mathbf{A}\square(f \Rightarrow \mathbf{E}\bigcirc g \langle L \rangle) \rrbracket_k &:= \bigwedge_{i=0}^{k-1} (\llbracket f \langle L \rangle \rrbracket_k^i \Rightarrow \llbracket g \langle L \rangle \rrbracket_k^{i+1}) \\
\llbracket \mathbf{A}\square(f \Rightarrow \mathbf{E}\diamond g \langle L \rangle) \rrbracket_k &:= \bigwedge_{i=0}^k \left(\llbracket f \langle L \rangle \rrbracket_k^i \Rightarrow \bigvee_{j=i}^k \llbracket g \langle L \rangle \rrbracket_k^j \right)
\end{aligned}$$

4 Determining Difficulty

4.1 Number of Clauses

We discuss in [4] a method for determining the number of clauses produced by an encoding. For propositional formulæ, this is summarised in Table 1. Applying the function to the general translation in (*) gives us (where T is the transition

function, and I is the initialisation function)

$$p(\llbracket M, f \rrbracket_k) = p(I) + kp(T) + \left(((k+1)\bar{p}(T) + p(\llbracket f \rrbracket_k^0)) \times \prod_{l=0}^k (p(T) + p(l\llbracket f \rrbracket_k^0)) \right)$$

This approach can be extended to CTL formulæ in general, and the value of the function obviously depends on the encoding used. In [4] this is examined for the restricted CTL case — where nested temporal operators are not permitted. While in many cases this restriction hides the differences between the encodings, certain differences still show up, for example with the unless operator (below), which is the pathological case for the original encoding.

	Original encoding	Encoding via SNF _C
$p(\llbracket f_1 \mathcal{W} f_2 \rrbracket_k^i)$	$\prod_{j=0}^k (p(f_1) + j(p(f_2)))$	$(1+k)p(f_2)(p(f_1) + 1)$
$p(l\llbracket f_1 \mathcal{W} f_2 \rrbracket_k^i)$	$k p(f_2) \prod_{j=0}^k (p(f_1) + j(p(f_2)))$	$(2+k)p(f_2)(p(f_1) + 1)$

4.2 Shortcomings of Size as a Difficulty Indicator

The number of clauses indication assumes a very simplistic CNF conversion procedure. Duplicate clauses and trivially true clauses are counted, where a real conversion procedure would eliminate them. Through the methods detailed in [3] the number of clauses may be reduced automatically using renaming, and a conversion procedure may also use this type of method.

In [4] we see that the number of clauses can be reduced drastically in certain restricted cases. An example which has just two variables modelled over two states produces 12 156 726 clauses; a simple set of rearrangements and one renaming gives a formula which produces just 41 clauses. Since there are six variables required in this encoding, even if all possible clauses were included in the output, this still gives an upper bound of $3^6 = 729$ non-trivial clauses. We can deduce that there must be repetition in the original clause set. (Note that with renaming, the number of variables is increased, so the upper bound on the number of clauses is increased.)

The observation to be made from this is that a SAT solver which performs a repetition check and a check for trivial clauses on its input may succeed more quickly than one which tries to apply a decision procedure to the whole clause set. Similarly, a solver which implements some sort of structure-sharing data structure will quickly eliminate a large number of clauses (for example SATO [6], which uses tries to share clause prefixes). It is this type of issue that makes computed size an inaccurate indicator, but also complicates the usefulness of SAT run time as an indicator.

4.3 Alternative Comparison Methods

We can envisage a number of alternative comparison methods, summarised below. Other methods will undoubtedly come to light as these are investigated.

- *Compute the number of useful clauses.* We define “useful” as non-repetitious and non-trivial. A deeper analysis of the formulas involved in bounded model checking may reveal indications to the origins of repeated clauses; this could lead to a more intelligent size function or heuristic, while still avoiding the need to actually carry out the clause form conversion (the main advantage of the p function).
- *Compare the runtime of SAT solvers.* While different SAT solvers differ enormously in their behaviour, even on similar problems, the results from many different solvers may reveal trends. Similarly, the difference in behaviour between DP and local search procedures will be instructive.
- *Analyse the nature of the clauses.* The number of variables per clause, the distribution of variables throughout the clauses, and the constrainedness of the clause set may indicate a trend in difficulty when compared with run times.

5 Summary

The problem of determining in advance the difficulty of a set of clauses is a very difficult one, but one which has applications in selecting the method by which the clauses are produced. We hope to present conclusive results to indicate preferred encoding methods, and to indicate how such results can lead to the development of better encodings.

References

1. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., July 1999.
2. Alexander Bolotov and Michael Fisher. A resolution method for CLT branching-time temporal logic. In *Proceedings of the Fourth International Workshop on Temporal Representation and Reasoning (TIME)*. IEEE Press, 1997.
3. Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In Claude Kirchner and Hélène Kirchner, editors, *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer-Verlag, 1998.
4. Daniel Sheridan and Toby Walsh. Clause forms generated by bounded model checking. In Andrei Voronkov, editor, *Eighth Workshop on Automated Reasoning*, 2001.
5. Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., 2000.
6. Hantao Zhang and Mark E. Stickel. *Implementing the Davis-Putman Method*.