



# Supporting SAT based BMC on Finite Path Models

Daniel Geist, Mark Ginzburg, Yoad Lustig  
Ishai Rabinovitz, Ohad Shacham and Rachel Tzoref  
[geist@il.ibm.com](mailto:geist@il.ibm.com)

*IBM, Haifa Research Labs*

---

## Abstract

The standard translation of a Bounded Model Checking (BMC) instance into a satisfiability problem, (a.k.a SAT), might produce misleading results in the case when the model under verification contains finite paths. Models with finite paths might be produced unknowingly when using modern verification languages such as PSL-Sugar [1]. Specifically, the use of language constructs such as *restrict*, *assume* etc. might lead to such models. Thus the user may receive misleading results from SAT based tools.

In this paper we describe in what circumstances the finite path problem occurs and present an improved translation of the BMC problem into a SAT instance. The new translation does not suffer from the discussed shortcoming. Our translation is only slightly longer than the usual one introducing one extra Boolean variable in the model.

We also show that this translation may improve the SAT solver runtime even for models without finite paths.

*Keywords:* BMC, PSL, Finite paths

---

## 1 Introduction

Since its introduction in the seminal paper [5], SAT-based Bounded Model Checking (BMC) has become an important tool in the verification engineer toolbox. However, traditional translation of a Bounded Model Checking instance into a satisfiability problem (a.k.a SAT) is not perfect. In particular it might produce misleading results when the model under verification contains a finite path that violates the specification.

In this paper we describe this problem and in what circumstances it might occur. Models with finite paths might be produced unknowingly when using modern verification languages such as PSL-Sugar [1]. Specifically, the use of language constructs such as *restrict*, *assume* etc. might lead to such models. Thus the user may receive misleading results from SAT based tools. We also present an improved translation of the BMC problem into a SAT instance that does not suffer from the discussed shortcoming. Our translation is only slightly longer than the usual one introducing one extra Boolean variable in the model.

Our improved translation not only fixes the problem when finite paths exist, it may also improve the performance of the SAT solver even for models with no finite paths.

The rest of the paper is divided as follows: Section 2 overviews the standard translation of the BMC problem to a SAT problem. Section 3 presents examples of models with finite paths. Section 4 presents our solution. Section 5 details run time results that show that the new translation can assist in runtime and Section 6 presents our conclusions.

## 2 Translating a BMC Problem to a SAT Problem

The usual way a BMC problem is translated to a SAT instance is quite simple. Before describing the translation itself, we introduce several notations:

### 2.1 Notations

Denote by  $s_i$  a vector of propositional variables encoding the state of the model in cycle  $i$ . Denote by  $INIT(o)$  the propositional formula translation the initial set, i.e.  $INIT(s_0)$  encodes "The state  $s_0$  is in the set of initial states". Denote by  $TR(o, o)$  the function encoding the transition relation, i.e.  $TR(s_{i-1}, s_i)$  encodes "There is a transition from state  $s_{i-1}$  to state  $s_i$ ". A *computation path* is a sequence of states  $s_0, s_1, \dots, s_n$  such that  $s_0$  is in the initial set and for each two consecutive states  $s_{i-1}, s_i$  there is a transition from state  $s_{i-1}$  to state  $s_i$  (i.e.  $TR(s_{i-1}, s_i) = 1$ ).

Finally we introduce the notion of a bad state. Recall that BMC can be applied to formulas of the form *always p*. Since the specification is of the type *always p*<sup>1</sup>, the specification can be seen as an invariant, a bad state is a state

<sup>1</sup> In fact, many other formulas can be transformed to formulas of the type always p, possibly adding to small monitoring automata to the model. SAT based BMC methods can then handle all safety formulas [3], and even liveness formulas (although in the latter case, the complexity price is significant).

that violates the specification invariant. For example, if the specification is *always p*, a bad state will be any state not satisfying *p*. Denote by  $BAD(o)$  the formula translation of the bad states, i.e.,  $BAD(s_i)$  encodes "The state  $s_i$  is a bad state".

A *bug* is a bad state that is reachable by a computation path.

## 2.2 The translation

The standard translation of BMC into a satisfiability problem is to find a satisfying assignment to the following equation:

$$(1) \quad INIT(s_0) \wedge \left( \bigwedge_{i=1}^k TR(s_{i-1}, s_i) \right) \wedge \left( \bigvee_{i=0}^k BAD(s_i) \right)$$

A bug can be reached within  $k$  cycles iff the traditional BMC formula is satisfiable. Furthermore, a satisfying assignment to the BMC formula can be translated in a straightforward manner to a counter-example trace leading to the bug.

In practice, the verification engineer fixes the length parameter  $k$ , for example  $k = 10$ . The tool produces the formula and feeds it into a SAT solver. If the formula is satisfiable then a bug is found, otherwise, the result is seen as "k-passed", i.e., the model does not contain a bug in paths up to length  $k$ . Some modern verification tools (such as IBM's RuleBase [4]) also provide automatic modes in which the bound  $k$  is automatically increased until a bug is found or the system runs out of resources.

```

VAR xx : 0..7;

ASSIGN
  init (xx) := 0;
  next(xx) := case
    xx = 7 : 0;
    else   : {xx, xx + 1};
  esac;

assume always((xx != 7) | (next(xx) = 2));

```

Fig. 1. Simple model with a finite path

### 3 Models with Finite Paths

Model checking is usually performed on infinite paths models [6]. A finite computation path may result from a computation ending in a state for which no other state satisfies the transition relation. For example, the model depicted in Figure 1 has finite paths: a finite path may occur if  $xx$  reaches 7. In that case,  $xx$  must both become 0 to satisfy the  $next(xx)$  assignment, and become 2 to satisfy the  $assume$  statement. Clearly no value can satisfy both constraints therefore the path has no continuation and is finite.

In modern verification languages such as PSL-sugar, finite paths might occur from constructs such as  $assume$  and  $restrict$  statements, using next variables on the right hand side of an assignment statement, and from assume or restrict verification directives. We would like to stress that even though the example in Figure 1 is contrived, models with finite paths often occur in practice because engineers find assumptions to be a very useful convenience and that was our motivation in doing this work.

When dealing with a model that has finite computation paths, it is customary to define a bug as valid only if it is part of an infinite path. Still, it is also reasonable and desirable to define it as valid even it has no infinite extension for several reasons:

- (i) In many cases the verification engineer is interested in bugs occurring on finite paths. In fact, the verification engineer often introduces verification directives (such as  $restrict$ ) that might turn many or even all paths to finite ones. This is done in order to reduce the state space significantly and to "concentrate" the verification effort on parts of the state space that the verification engineer considers sensitive. For example, a verification engineer may use such directives in order to ignore paths with known bugs, so the verification engineer may turn an assert into an assume, however, any bug that occurs prior to the violation of the assume is relevant and should be reported.
- (ii) Some of the main verification techniques used by modern tools, do not guarantee that a bug found is on an infinite path. The most notable are BDD-based On The Fly verification (e.g. IBMs Discovery engine inside RuleBase), and SAT based Bounded Model Checking. While both techniques can be adapted to ensure that a bug found can be reached on an infinite path, the cost (in terms of time and memory consumption) of this adaptation is significant, many times much bigger than the cost of finding the bug in the first place.

To see that SAT based BMC does not ensure that the bug found is on

an infinite path, simply note that the BMC formula refers only to the first  $k$  cycles and there is no guarantee as to what happens after cycle  $k$ . There are SAT based algorithms that solve the unbounded MC problem but they do not scale to large designs as BMC.

Therefore, the approach today is to find bugs on finite as well as infinite path. While this approach does not follow the strict temporal logic definitions it enables the verification engineer to enjoy stronger tools (such as BDD based On The Fly algorithm, or SAT based BMC), as well as easier use of verification directives statements (such as *restrict*). See for example [7] that discusses the temporal logic semantics on finite paths and considers validity of bugs which appear only on finite paths.

Verification engineers that use BMC to find bugs expect BMC to be *monotonic*: the expectation is that if BMC does not find a bug on a run with a bound of  $k$ , then it can not find bugs on any run with bound  $k' < k$ . The monotonicity of BMC is a very important attribute, since it enables the verification engineer to increase  $k$  in increments greater than one without the risk of missing a bug.

The traditional BMC formula does not treat finite paths well enough. For example, the model in Figure 2 is deterministic, and therefore contains only one path. This single path is finite and of length 5 since at cycle 5 (when the first cycle being cycle 0) it holds that  $xx = 5$ . If  $xx = 5$  the path has no continuation since no state can satisfy both the *next*( $xx$ ) assignment statement and the *assume* statement.

```

VAR xx : 0..7;

ASSIGN
  init (xx) := 0;
  next(xx) := case
    xx = 7 : 0;
    else   : xx + 1;
  esac;

assume always((xx != 5) | (next(xx) != 6));

```

Fig. 2. An example of a model with all paths being finite

### 3.1 The Bounded Paths Problem

The equation Eq. 1 encodes the following statement: "There is a computation path of length  $k$ , and somewhere on this path a bad state is encountered". Note however, that in a model in which all paths that violate the specification<sup>2</sup> are finite and are of length  $k - 1$  or less the formula is unsatisfiable although a bug might be encountered before cycle  $k - 1$ . For example, look at the model in Figure 2 with the specification *always*( $xx < 3$ ). The model violates the specification in the fifth cycle (the first cycle being cycle 0). However, if the verification engineer sets the bound  $k$  to 10, then Eq. 1 will be unsatisfiable because there are no paths of length 10. Thus the verification engineers seeing that the formula is unsatisfiable will classify the model as "10-passed", which is clearly wrong. Increasing the bound will not help and decreasing the bound is counter-intuitive for the engineer and it is impractical to ask him to consider it. So the end result will be a bug miss which is a very **severe** outcome. In other words, using SAT on a model which contains PSL assumptions absolutely requires handling this case.

By this example we can see that the presence of finite paths causes BMC to lose its monotonicity attribute, this is problematic even if the verification engineer is not interested in bugs that occur on finite paths, the reason is that the verification engineer cannot tell that a bug is on a finite path (the path may be long enough), and therefore can suffer from the following scenario:

- (i) A BMC run with a bound of  $k$  passes since it ignores a violation in a finite  $k'$ -length path ( $k' < k$ ).
- (ii) As a result of a change in the design, the verification engineer runs BMC again and by chance uses a bound of  $k'' < k'$ . A bug is reported and the verification engineer assumes that this is a new bug, entered by the change in the design.

Such a scenario is obviously problematic.

## 4 Solution to finite path problem

A simple solution is to start with  $k = 1$  and increment  $k$  by 1 on each iteration of the verification tool, then we are sure to catch the bug on the first  $k$  it appears. The problem with this solution is of course that it is extremely time consuming since many invocations of the verification tool are needed. A second solution is to encode into a propositional formula the statement

---

<sup>2</sup> We neglect paths in which the bug is cycle larger than  $k$ , and cannot be found in this run of BMC in any case.

”Either there is a path of length 1 leading to a bad state, or there is a path of length 2 leading to a bad state or... there is a path of length  $k$  leading to a bad state”. This solution, while only invokes the SAT solver once, invokes it on a significantly longer formula (in fact of quadratic length compared to the original formula), which is extremely costly.

#### 4.1 The Improved Translation

A better solution to the finite path problem can be achieved by changing slightly the traditional BMC formula. We introduce one extra Boolean variable to the model called **AlreadyFailed**. This variable records whether a bug has been hit on particular path. Mathematically:

- For an initial state  $s_0$   
 $AlreadyFailed(s_0) \leftrightarrow BAD(s_0)$
- For any other state  $s_i$ ,  
 $AlreadyFailed(s_i) \leftrightarrow (AlreadyFailed(s_{i-1}) \vee BAD(s_i))$

The BMC equation now becomes:

$$(2) \quad INIT(s_0) \wedge \left( \bigwedge_{i=1}^k (TR(s_{i-1}, s_i) \vee AlreadyFailed(s_{i-1})) \right) \wedge \left( \bigvee_{i=0}^k BAD(s_i) \right)$$

Thus by adding this extra variable and changing the translation we are ensured of identifying a bug even if it is sitting on a finite path and the bound  $k$  we choose to submit to the model checker is greater than the length of that path.

In fact it is possible to simplify this translation in two ways:

- (i) we can replace the definition of **AlreadyFailed** by  
 $AlreadyFailed(s_0) \rightarrow BAD(s_0)$   
 $AlreadyFailed(s_i) \rightarrow (AlreadyFailed(s_{i-1}) \vee BAD(s_i))$
- (ii) we can replace the term  $\bigvee_{i=0}^k BAD(s_i)$  in Eq. 2 with  $AlreadyFailed(s_k)$ .

However, these replacements may not necessarily provide a better runtime.

#### 4.2 Implementation details

The new translation presented in Eq. 2 can in theory be applied always, even if the model under verification does not contains finite paths. In practice, It is not clear the effect of this translation to the performance of the SAT solver. It can slow the SAT solver since some optimizations cannot be performed when this translations is used, and since we added a new variable to the formula. On

the other hand it may improve the SAT solver performance especially when the formula is satisfiable. The intuition is that the SAT solver attempts to find an assignment for all the variable replications until cycle  $k$  even if the bug is on cycle  $k' < k$ . In the new translation the SAT solver takes advantage of the *AlreadyFailed* variable replications to assign arbitrary values to variable replications belonging to cycles greater than  $k'$ , and therefore may find the assignments faster.

For models with finite paths this is a necessity. Therefore the actual implementation has the following details:

The translation checks if the model contains PSL constructs that cause finite paths and chooses the translation according to that:

- (i) When there can be finite paths there are several options:
  - (a) It is recommended to use the new translation (Eq. 2).
  - (b) The user can use the traditional translation, while advancing  $k$  by 1 each run. this way no bug is missed. However, this seems to be a very slow technique.
  - (c) The user can force the use of the traditional translation (Eq. 1) using larger steps. However, the risk of missing a bug is taken after a conscientious decision.
- (ii) When there can be no finite paths, the only issue in choosing the translation is the SAT solver performance. There are several options:
  - (a) Use the new translation (Eq. 2). This is recommended if it the user think that the SAT solver will find a satisfying assignment.
  - (b) Use the traditional translation. This is recommended when the user think that the SAT solver will not find a satisfying assignment.
  - (c) Use both translations and run two SAT solvers in concurrent, killing the slowest after the quickest gives a response. This is recommended for users that have the hardware resources.

The trace that is generated may contain states after the cycles that the bug occurred. In case the translation in Eq. 2 was used, those states can violate the constraints of  $TR(o, o)$  and hence confuse the engineer. A postprocessing program has to remove those states from the trace before presenting it to the user.

## 5 Experimental results

As mentioned in Section 3 models with finite paths require the translation of Eq. 2 in order to identify an error so regardless of runtime improvement it is necessary to use the new translation. However, the new translation can also

improve runtime. Table 1 details a comparison of the new and old translation in runs when the bound  $k = 100$ . This is a typical situation when a verification engineer starts verifying a new design. In this case, the engineer assumes, as is usually the case, that a bug exists in the first 100 cycles. All the SAT problems in Table 1 are satisfiable. They are taken from some proprietary industrial designs and from the IBM benchmarks [2]. Table 1 shows that in most of the cases, except for *D1*, the new translation significantly reduces the runtime. A reasonable explanation for such results is the fact that when a bug exists in a relatively small cycle then the new translation makes it much easier for the SAT solver to find a satisfying assignment for the formula variables in the higher cycles.

Table 2 details another realistic usage methodology. In practice, users run successive SAT in bound increments of 10 or 20 until a bug is found or a desirable cycle limit is reached. When the model contains finite paths the traditional translation can only be used with increments of 1. In order to use larger increments the new translation has to be used. All the models in the table contain finite paths. The first column in Table 2 specifies if the SAT problem is satisfiable or not. The second column specifies in which cycle the bug is found or the desirable limit was reached. The fourth column details the runtime using the old translation with 1 cycle increments. The fifth and last columns detail the runtime with the new translation using 10 and 20 cycle increments. In all of the examples we tried, incrementing the bound by 20 or by 10 was faster than incrementing by 1. Note that there is no reason to use the new translation with increments of 1 since its advantage is when the bug is not on the last cycle.

## 6 Conclusions

In this paper we presented a new encoding to SAT based BMC that enables BMC to effectively handle models with finite paths. This encoding is necessary for preserving the monotonicity of BMC on a model with finite paths without serious performance degradation. In a certain cases, we have shown that the use of the new encoding increases the SAT solvers performance.

In the future, we plan to implement this new encoding in conjunction with incremental SAT based BMC [8][9].

	New trans. k=100	Old trans. k=100
batch_1_11	<b>140</b>	3201
D1	23	13
batch_29	<b>142</b>	192
batch_20	<b>826</b>	4039
batch_22	<b>3203</b>	13383
batch_18	<b>490</b>	3293
D2	<b>237</b>	250

Table 1

New translation Vs. Old translation. The runtime is displayed in seconds.

	SAT/UnSAT	No. Cyc.	Old Trans. inc. 1	New Trans. inc. 10	New Trans inc. 20
D1	SAT	40	18773	<b>948</b>	3221
D3	UnSAT	60	37199	3112	<b>2115</b>
D4	UnSAT	60	18818	884	<b>411</b>
D5	UnSAT	10	1176	<b>196</b>	—
D6	UnSAT	15	8265	<b>4787</b>	—
D7	UnSAT	100	14712	685	<b>394</b>
D8	SAT	31	991	556	<b>159</b>
D9	UnSAT	60	>145000	1916	<b>1763</b>
D10	UnSAT	100	40012	5622	<b>3059</b>

Table 2

comparing various increments of the new translation versus increments of 1 in the old translation. The runtime is displayed in seconds.

## References

- [1] Property Specification Language: *Reference Manual*. Version 1.1, Accellera, June 2004.
- [2] The IBM Formal Verification Benchmarks,  
[http://www.haifa.il.ibm.com/projects/verification/RB\\_Homepage/bmcbenchmarks.html](http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html),  
 2004

- [3] Ilan Beer, Shoham Ben-David, Avner Landver: *On-the-Fly Model Checking of RCTL Formulas*. CAV 1998.
- [4] Shoham Ben-David, Cindy Eisner, Daniel Geist, Yaron Wolfsthal: *Model Checking at IBM*. Formal Methods in System Design 22(2): 101-108 (2003)
- [5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: *Symbolic Model Checking without BDDs*. TACAS 1999.
- [6] Edmund Clarke, Orna Grumberg, Doron Peled, *Model Checking*, MIT Press, 2000.
- [7] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac and David Van Campenhout, *Reasoning with Temporal Logic on Truncated Paths*, CAV04.
- [8] H. Jin, F. Somenzi. *An incremental algorithm to check satisfiability for bounded model checking*. BMC04
- [9] Ofer Strichman: *Pruning Techniques for the SAT-Based Bounded Model Checking Problem*. CHARME 2001.