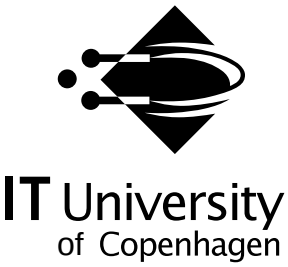

Greedy Model Checking

Poul Frederick Williams
Antoine Rauzy

IT University Technical Report Series **TR-2000-2**
ISSN 1600–6100 **October 2000**

ISSN 1600–6100

ISBN 87–7949–002–6

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 3816 8888
Telefax: +45 3816 8899
Web www.it-c.dk

Greedy Model Checking

Poul Frederick Williams[1] Antoine Rauzy[2]

Abstract

We present a model checking method which greedily explores the state space. Using ideas similar to greedy satisfiability checking, our method tries to fit a path to match a temporal specification. The advantages of this method is that we do not need any quantifications, we do not calculate a reachable (neither forward nor backward) set of states, and the memory requirements are quite small.

1 Introduction

Greedy model checking (GMC) is a new technique for determining whether a Kripke structure is a model for a temporal logic specification. GMC is inspired by greedy satisfiability checking (GSAT) [11] of CNF formulae. GSAT works as follows: We generate a random bit vector and measure how close it is to satisfy a given formula — a typical measure is the number of clauses satisfied by the vector. Then we repeatedly flip bits in the vector in such a way that our satisfiability measure always increases. We stop when either the vector satisfies the formula or a maximum number of flips has been reached. In the latter case we may generate a new, random bit vector and start over again.

In GMC we use the same basic idea of GSAT on Kripke structures. Instead of fitting a bit vector to satisfy a formula, we fit a sequence of states to become a path in the Kripke structure. In the rest of this paper we will fit the sequence of states to become a path with a fixed set of initial states, a fixed length, and a fixed set of end states. This corresponds to a witness for an **EF** (reachability) property or a counterexample for an **AG** (invariant) property.

Like GSAT, GMC is not complete. If we find a witness we know the specification holds. If we are unable to find a witness, we cannot conclude anything. Likewise for counterexamples. If we find a counterexample we know the specification does not hold. Lack of a counterexample does not allow us to conclude anything.

2 Related Work

Model checking was invented by Clarke, Emerson, and Sistla in the 1980s [9]. Their model checking method required an explicit enumeration of states which limited the size of the systems they could handle. Burch *et al.* [7] showed how to do model checking without enumerating the states. They called this symbolic model checking (SMC). Using fixed point iterations, they compute characteristic functions for sets of states which can then be used to determine whether the specification holds. The key to the success is the canonical Binary Decision Diagram (BDD) [6] data structure for representing Boolean functions. However, such a representation explodes in size for certain functions.

Recently people have studied alternatives to BDDs in model checking. Biere, Clarke *et al.* have proposed Bounded Model Checking (BMC) as an alternative [3, 4, 5]. They unfold the transition relation and look for repeatedly longer and longer counterexamples, and then use SAT-solvers instead of BDDs for the verification. BMC is good at finding errors with short counterexamples. The diameter of the system determines the number of unfoldings of the transition relation that are necessary in order to prove the correctness of the circuit. Unfortunately, for many examples the diameter cannot be calculated and the estimates are too rough. In such cases BMC reduces to a partial verification method in practice.

Abdulla *et al.* [1] and Williams *et al.* [14] have both proposed the use of SAT-solvers in fixed point based model checking. Abdulla *et al.* use a Reduced Boolean Circuit data structure for representing Boolean formula and then Stålmarck's patented method [12] for satisfiability checking. Williams *et al.* use the

[1] Email `pfw@it-c.dk`

[2] With the CNRS Université Bordeaux I, Computer Science Lab. (LaBRI), 351, cours de la Libération, Talence, F-33405, France. Email `rauzy@labri.u-bordeaux.fr`

Boolean Expression Diagram (BED) [2] data structure and combine SAT-solver and BDD techniques in the verification.

3 Definitions

A Kripke structure M is a tuple (S, I, T, ℓ), with a finite set of states S, a set of initial states $I \subset S$, a transition relation $T \subset S \times S$, and a labeling of the states $\ell : S \to \mathcal{P}(\mathcal{A})$ with *atomic propositions* $\mathcal{A}$. We assume the presents of a set of goal states G.

Definition 1 (Path) *Given a Kripke structure $M = (S, I, T, \ell)$ and a set of goal states G, a* path *of length n from I to G is a sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ (all $s_i \in S$) such that:*

- $s_0 \in I$
- $s_n \in G$
- $T(s_i, s_{s+1})\,, \qquad i = 0, \ldots, n-1$

Definition 2 (Quasi path) *Given a Kripke structure $M = (S, I, T, \ell)$ with set of goal states G, a sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ (all $s_i \in S$) is called a* quasi path *of length n if there exists a j such that:*

- $s_0 \in I$
- $s_n \in G$
- $T(s_i, s_{s+1})\,, \qquad i = 0, \ldots, j-1, j+1, \ldots, n-1$

We call the place, if any, in the quasi path where the transition relation does not hold for a *gap*.
We will use $T(I, \cdot)$ and $T(\cdot, G)$ to mean the following relations:

$$\begin{aligned} T(I, \cdot) &= \{\ (s,t) \in S \times S \mid T(s,t) \text{ and } s \in I\ \} \\ T(\cdot, G) &= \{\ (s,t) \in S \times S \mid T(s,t) \text{ and } t \in G\ \} \end{aligned}$$

We use the standard tricks for working with Kripke structures as Boolean formulae: We use a vector of Boolean variables to encode states, and characteristic functions over these variables to represent sets of states. As atomic propositions we use the Boolean variables and implicitly label each state with the Boolean variables which are true in that state. To represent the transition relation we need two sets of Boolean variables: a set of unprimed variables for the current state and a set of primed variables for the next state. Thus we represent the transition relation as a characteristic function of primed and unprimed variables.

The *Hamming distance* H between two states $s_i, s_j \in S$ is the number of places their bit vector representations differ. If $H(s_i, s_j)$ is zero then s_i is equal to s_j. Note that the Hamming distance depends on the chosen method for encoding states as bit vectors.

4 Algorithm

Given a Kripke structure $M = (S, I, T, \ell)$, a set of goal states G, and a length n, we look for a path $\langle s_0, \ldots, s_n \rangle$ in M.

To begin with we create a quasi path of length n from I to G. Let $\langle s_0, \ldots, s_n \rangle$ be that quasi path. If we assume that the transition relation T is total, then the quasi path can be constructed by a random walk of length $n-1$ in M starting from a random state s_0 in I, and finally pick a random state s_n in G.

Figure 1 shows a quasi path of length 4 from I to G. The dashed line from s_2 to s_3 indicates that there is no corresponding transition in the transition relation.

If the quasi path $s = \langle s_0, \ldots, s_n \rangle$ is a path, then we are done. Otherwise let j indicate a gap between s_j and s_{j+1}. For the quasi path in Figure 1, j equals 2.

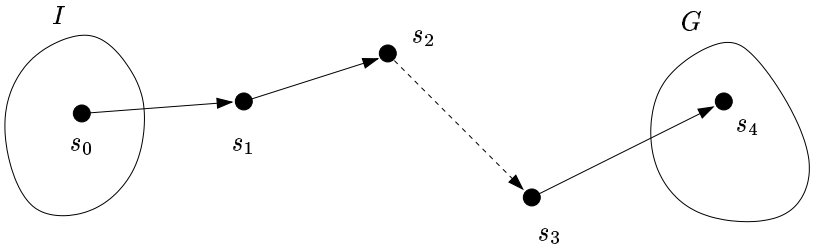

Figure 1: A quasi path from I to G. The dashed line from s_2 to s_3 indicates that there is no corresponding transition in the transition relation.

Name: gmc(I, G, T, n)
1: *tries* $\leftarrow$ max number of tries
2: **while** *tries* > 0 **do**
3: *tries* $\leftarrow$ *tries* $- 1$
4: $j \leftarrow$ create-quasi-path(I, G, T, n)
5: **if** flip-states(I, G, T, n, j) **then**
6: **return** "$\langle s_0, s_1, \ldots, s_n \rangle$ is a path of length n"
7: **return** "no path found"

Algorithm 1: The gmc algorithm. The create-quasi-path function picks a random quasi path, assigns the global variables $s_0, s_1, \ldots, s_n$ the states along this quasi path, and returns the index j for the gap between s_j and s_{j+1}.

We now try to patch the quasi path such that it becomes a path. The quasi path is not a path because there is no transition from s_j to s_{j+1}. We compute two new candidate states s'_j and s'_{j+1} for patching the quasi path. The new states must satisfy:

$$T(s'_j, s_{j+1}) \qquad \text{and} \qquad T(s_j, s'_{j+1}) \tag{1}$$

By using either s'_j instead of s_j or s'_{j+1} instead of s_{j+1} there is no longer a problem at position j in our quasi path. However, we may have created a problem at position $j-1$ (if we chose s'_j) or at position $j+1$ (if we chose s'_{j+1}).

To chose whether to pick s'_j or s'_{j+1} we use the following procedure: If one of them turns the quasi path into a path, then select that one. Otherwise, select s'_j if $H(s_j, s'_j)$ is less than $H(s_{j+1}, s'_{j+1})$, select s'_{j+1} if $H(s_j, s'_j)$ if greater than $H(s_{j+1}, s'_{j+1})$, and select either one at random if $H(s_j, s'_j)$ equals $H(s_{j+1}, s'_{j+1})$. This way we select the one which makes the least changes as measured by the Hamming distance between bit vectors. The intuition behind this heuristic is to minimize changes in the hope that if there is a transition between two states then there is also a transition even though we make a small modification in one of the state bit vectors.

This still leaves the question of how to calculate s'_j and s'_{j+1}. Ideally we want the Hamming distance between the new states and the old ones to be minimal. However, it is computationally expensive to find the exact solution so we use an approximation: We run through all state variables. For each variable we assign it the same value as it had in the bit vector for the old state. If an assignment does not lead to a solution in which (1) holds, we backtrack and try the opposite assignment. It is possible that there exists no s'_j or s'_{j+1}. If one does not exists, we simply chose the other one. If both do not exist, we have reached a dead-end and possible courses of action are giving up or generating a new, random quasi path.

In the special cases where $j = 0$ and $j = n - 1$ we make sure that the new s_0 is in I and the new s_n is in G. One way to do this is to use a modified version of the transition relation which is restricted to transitions starting in I or ending in G, respectively.

We run the above procedure *tries* times, where in each try we perform *flips* changes to the quasi path. The variables *tries* and *flips* decide how much of the state space we explore. Algorithms 1 and 2 show the pseudo code.

Name: `flip-states`(I, G, T, n, j)

1: $flips \leftarrow$ max number of flips
2: **while** $flips > 0$ **do**
3: $\quad flips \leftarrow flips - 1$
4: $\quad T_{new} \leftarrow T$
5: $\quad$ **if** $j = 0$ **then**
6: $\quad\quad T_{new} \leftarrow T \cap T(I, \cdot)$
7: $\quad$ **if** $j = n - 1$ **then**
8: $\quad\quad T_{new} \leftarrow T \cap T(\cdot, G)$
9: $\quad (s'_j, s'_{j+1}) \leftarrow$ `calculate-new-states`(j, T_{new})
10: $\quad$ **if** both s'_j and s'_{j+1} are invalid **then**
11: $\quad\quad$ **return** false
12: $\quad$ **if** s'_j is better than s'_{j+1} or 50% chance if they are equally good **then**
13: $\quad\quad s_j \leftarrow s'_j$
14: $\quad\quad$ **if** $T(s_{j-1}, s_j)$ **then**
15: $\quad\quad\quad$ **return** true
16: $\quad\quad j \leftarrow j - 1$
17: $\quad$ **else**
18: $\quad\quad s_{j+1} \leftarrow s'_{j+1}$
19: $\quad\quad$ **if** $T(s_j, s_{j+1})$ **then**
20: $\quad\quad\quad$ **return** true
21: $\quad\quad j \leftarrow j + 1$
22: **return** false

Algorithm 2: The `flip-states` algorithm. The algorithm returns true if the quasi path $\langle s_0, s_1, \ldots, s_n \rangle$ becomes a path. The function `create-new-states` returns two states s'_j and s'_{j+1} such that (1) holds (using T_{new} as the transition relation), or "invalid" if such states do not exist. The "better than" in line 12 means that we pick the state which turns the quasi path into a path, or if neither do that then we pick the state which makes the least changes measured by the Hamming distance.

4.1 Early Stop

We are looking for a path of length n from I to G. Let the current quasi path be $\langle s_0, s_1, \ldots, s_n \rangle$ with j indicating the place of the gap. If any of the states $s_0, s_1, \ldots, s_j$ belong to G, say $s_i \in G$, then $\langle s_0, s_1, \ldots, s_i \rangle$ is a path of length i from I to G. We were looking for a path of length n, but found one of length i. In many cases this is fine and we can stop. In fact, a shorter path (remember that $i \leq j < n$) is often preferable, for example when the path is a counterexample showing a bug in a system.

When replacing s_{j+1} with s'_{j+1}, the sequence $\langle s_0, s_1, \ldots, s'_{j+1} \rangle$ is a path from I to G if and only if $s'_{j+1} \in G$ (since s'_{j+1} was constructed such that (1) holds). Instead of first constructing s'_{j+1} and then check whether it is a member of G we could try to construct s'_{j+1} such that $s'_{j+1} \in G$. An easy way to do this is to restrict the transition relation to $T(\cdot, G)$. Any s'_{j+1} we find in this manner will indicate a path of length $j+1$ from I to G. If, on the other hand, we cannot find such a $s'_{j+1} \in G$ we simply remove the restriction on the transition relation and look for a state s'_{j+1} which satisfies (1) but is not necessarily a member of G.

4.2 Combining GMC and SMC

It is possible to combine GMC and regular symbolic model checking (SMC). In GMC, instead of looking for a path from the set of initial states I to a goal state in G, we could look for a path from I' to G' where I' and G' are:

$$\begin{aligned} I' &= I \cup T(I) \cup T^2(I) \cup \ldots \cup T^x(I) \\ G' &= G \cup T^{-1}(G) \cup T^{-2}(G) \cup \ldots \cup T^{-y}(G) \end{aligned}$$

and x and y are small, positive integers. The notation $T^i(I)$ denotes the set of states reachable in exactly i steps from I. Likewise, $T^{-i}(G)$ denotes the set of states from which G can be reached in exactly i steps. In SMC we compute I' and/or G' for x and y equal to infinity. We now break off the computation after a few steps and apply GMC.

It is, of course, also possible to use $I' = T^x(I)$ and $G' = T^{-y}(g)$. In this case we know that if there is a path of length n from I' to G', then there is also a path of length $n + x + y$ from I to G.

4.3 Implementation

We have implemented a prototype version of GMC. It uses BDDs as the underlying data structure — we use the BED package[3] from the Technical University of Denmark and the IT University of Copenhagen, but run it as a BDD package. We use BDDs to represent characteristic functions for states, sets and relations.

Instead of working with a monolithic transition relation, we use a conjunctive partitioning of T. This has the advantage that restricting T to $T(I, \cdot)$ or $T(\cdot, G)$ can be done by just adding the characteristic functions for I or G, respectively, as a new conjunct to the transition relation. (The characteristic function for G must be expressed in primed variables.)

No computations require expensive BDD operations. All we need to do is to traverse BDDs and construct BDDs for states.

4.4 Comparison with BMC

BMC looks for paths of a certain length. Let $s = \langle s_0, s_1, \ldots, s_n \rangle$ be a sequence of states. Then BMC constructs a Boolean formulae for the requirements s must satisfy to become a path with the appropriate properties. For example, s_0 must be a member of the initial set of states, s_n must be a member of the set of goal states, and each pair (s_i, s_{i+1}) must belong to the transition relation.

In GMC we have a quasi path and try to close the single remaining gap to make it a real path. BMC, however, starts with nothing but gaps and tries to close them all at once. This results in a need for BMC to use a large number of Boolean variables — one set of variables for each state in the sequence s. In GMC we need only two sets of variables — one set for s_j and one for s_{j+1}, where j is the position of the gap.

[3] See [13] or `http://www.it-c.dk/research/bed` for more information

In our implementation of GMC we use BDDs to represent (a conjunctive partitioning of) the transition relation. Since means that we are limited to model checking systems for which we can construct such BDDs. The limitation is a consequence of our choice of BDDs as a data structure for GMC; not a property of GMC itself. BMC does not have this limitation as no BDDs are involved.

5 Experimental Results

To evaluate the performance of GMC, we test it on industrial model checking problems. We chose the SMV-version of the `motor` example from [10]. It is motor control system with 287 reachability properties in the specification. We pick six of these properties (number 0, 50, 100, 150, 200, and 250) and look for witness paths of length 1, 5, 10, 15, 20, 25, 50, and 100. For comparison we use BMC [3, 4, 5] and NuSMV [8]. As a SAT-solver for BMC we use SATO [15]. All results were run on a 450 MHz Pentium III computer under Linux. The GMC method uses *tries* equal to 50 and *flips* equal to 100, and early stop has been enabled.

The memory requirements for the three methods were quite different. GMC ran in 5 MB for all examples. NuSMV used up to 16 MB. And BMC used up to 100 MB for generation of the formula and 80 MB for SAT-solving with SATO — since they run in sequence, the memory requirement is the max of the two, that is 100 MB.

	Path	Goal					
Method	length	0	50	100	150	200	250
GMC	1	0.2	-	-	-	0.2	-
BMC	1	0.5	0.5	0.5	0.5	0.5	0.5
GMC	5	0.1	-	1.6	-	0.2	-
BMC	5	1.5	2.0	1.6	1.8	1.6	2.2
GMC	10	0.1	-	0.7	-	0.3	-
BMC	10	3.2	4.0	3.1	3.2	3.0	4.1
GMC	15	0.2	-	0.8	-	0.4	1.2
BMC	15	4.8	6.0	5.2	5.3	5.1	7.1
GMC	20	0.2	-	1.4	-	0.5	0.4
BMC	20	6.5	9.0	6.6	7.2	6.7	9.3
GMC	25	0.4	-	2.7	-	0.7	-
BMC	25	8.1	11.4	8.4	8.4	8.2	11.8
GMC	50	1.9	-	1.3	-	0.2	0.8
BMC	50	19.2	25.2	19.7	19.8	19.1	26.7
GMC	100	-	3.1	-	7.9	1.2	5.0
BMC	100	53.8	66.1	55.1	54.8	53.4	67.5
NuSMV		10.9	6.0	7.1	3.6	0.5	0.6
Minimal length		0	5	3	5	0	4

Table 1: Runtimes in seconds for GMC, BMC, and NuSMV on six of the 287 specifications from the `motor` example. A dash indicates that no path was found by GMC. Since GMC is a semi-decision procedure, a dash either means that no path exists or that a path exists but GMC could not find it. Since the other methods (BMC and NuSMV) are decision procedures, they are always able to give a yes/no result and therefore there are no dashes among their results. The last row gives the length of the shortest path for each of goal.

Table 1 shows the runtimes in seconds for the different methods. When looking for path lengths of 1, GMC is only able to find two (for goals 0 and 200). The other goals do not have paths of length 1 and since GMC is a semi-decision procedue, it cannot say no. BMC handle all the examples with length 1 — a path exists for goals 0 and 200, but for the other four goals, no path exists. This fits nicely with the GMC results.

As the path lengths grow, BMC takes longer and longer to finish the computations and it uses more and more memory. GMC uses an almost constant amount of memory (5 MB $\pm$ a few hundred kB). The

GMC runtimes increase only slightly with the path length. Note that allowing for a longer path, GMC is sometimes able to find results which it was unable to find with a shorter path length. Examples are path lengths of 100 for goals 50 and 150. Since looking for a longer path is not a penalized performancewise, it is a good idea to try longer paths.

6 Conclusion

We have presented a model checking method which resembles greedy satisfiability checking by greedily fitting a sequence of states to become a path in a Kripke structure. While not complete, the method has the advantage of not calculating sets of reachable states like most standard model checking methods do. Furthermore, our method uses very little memory. This makes our method an interesting alternative in situations where the representations of the reachable sets of states become too large.

In this paper we have dealt with paths between two sets of states corresponding to witnesses for **EF** properties or counterexamples for **AG** properties. We believe our method could easily be expanded to handle other topologies, for example lassos which could be witnesses for **EG** properties or counterexamples for **AF** properties.

It is possible to extend GMC to adapt the length of the path it is looking for. For example, when we reach a dead-end, we could add an intermediate state and thereby increasing the length of the path by one. In this way GMC decides by itself the length of the path.

References

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000.

[2] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[5] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[8] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] J. B. Lind-Nielsen. *Verification of Large State/Event Systems*. PhD thesis, Dept. of Information Technology, Technical University of Denmark, Lyngby, Denmark, April 2000. ISBN 87-89112-57-1.

[11] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proc. Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press.

[12] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. J. Windley, editors, *Proc. Formal Methods in Computer-Aided Design, Second International Conference, FMCAD'98, Palo Alto/CA, USA*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99, November 1998.

[13] P. F. Williams. *Formal Verification Based on Boolean Expression Diagrams.* PhD thesis, Dept. of Information Technology, Technical University of Denmark, Lyngby, Denmark, August 2000. ISBN 87-89112-59-8.

[14] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138, Chicago, U.S.A., July 2000. Springer-Verlag.

[15] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275, Berlin, July 1997. Springer-Verlag.