Boosting the Role of Inductive Invariants in Model Checking

Gianpiero Cabodi and Sergio Nocco and Stefano Quer
Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

Abstract

This paper focuses on inductive invariants in unbounded model checking to improve efficiency and scalability.

First of all, it introduces optimized techniques to speed-up the computation of inductive invariants, considering both equivalences and implications between pairs of nodes in the logic network. Secondly, it presents a very efficient dynamic procedure, based on an incremental SAT approach, to reduce the set of checked invariants. Finally, it shows how to effectively integrate inductive invariant computations with state-of-the-art model checking procedures.

Experiments address different property verification aspects, and specifically consider cases where inductive invariants alone are not sufficient for the final proof.

1. Introduction

Given a transition system, an inductive invariant is a safety property p, written as $AG(p)$ in Computation Tree Logic (CTL), that has been proved to hold by *induction*. Given a depth k, a k-step inductive check verifies that:

- Starting from the initial state set of the system, p holds for k consecutive time steps.
- If p holds for k consecutive time steps, regardless the initial state, then it holds also at time step $k + 1$.

If both the above conditions (the *base case* and *proving step*, respectively) are true, then p is always true for the system. Obviously, if the base case is false, then p does not hold. If the base case is true but the proving step fails, then nothing can be said about p. At this point, k is increased and the two conditions are verified again.

The use of inductive invariants in formal verification was initially proposed by van Eijk [12] for sequential equivalence checking. His approach finds initial candidate equivalences with random simulation and carries out all computations by BDD-based manipulations. Moreover, the process is limited to 1-step induction, and it requires a quadratic number of iterations (with respect to the total number of nodes in the system) to reach the fix-point. Bjesse et al. [2] applied van Eijk's algorithm to property checking. The method relies on a SAT tool instead of BDDs, and it extends van Eijk's approach by strengthening the induction to k steps. In order to avoid the quadratic complexity, the authors also put forward an incomplete method to trade-off the quality of the result for the execution time. With the target of simultaneously checking a set of properties, Fraer et al. [7] proposed a linear complexity algorithm that, given the invariant set and an induction depth k, finds the maximum subset of invariants provable through k-step induction. More recently, invariants in the form of implications have been considered in [5]. Because of the huge amount of potential invariants, the authors adopted a divide-and-conquer strategy, which considers invariants only inside a window of the circuit. Once a fix-point has been reached, the window is enlarged and the process restarts.

In this paper, we combine equivalence and implication invariant management in a global effort to minimize the amount of invariants to manipulate. We improve over the previous approaches in the following directions:

- Equivalence classes are used as a (reduced) representation of all node equivalences. Given a set of equivalent nodes and a class representative, we avoid a quadratic number of equivalences by expressing (a linear number of) equivalences with the class representative.
- Implications are kept reduced by filtering out equivalences, representing implications between equivalence classes, and pruning out transitive implications.
- Equivalences and implications are proved under external care set conditions, which allow a tighter integration (and inter-twining) with other verification engines and/or strategies.
- We adopt state-of-the-art partitioning/windowing strategies, in view of a scalable approach, able to tackle large problem instances.
- Within the verification field, implications have been traditionally used as a pre-processing step able to poten-

978-3-9810801-2-4/DATE07 © 2007 EDAA

tially provide early solutions by pure induction [11]. We provide, instead, a dynamic and unified framework which tightly integrates invariant-based optimization and re-synthesis with various verification strategies. In this scheme, invariants are generally considered as a restriction of the state space.

Our experiments specifically address over-approximate reachability analysis and BDD/SAT-based property verification.

2. Background

The sequential systems we address are modeled as Finite State Machines (FSMs). We denote with S the set of states, with $S_0 \subseteq S$ the initial states, and with TR the transition relation of the system, TR: $S \times S \rightarrow \{0, 1\}$. We call L the set of potential invariants, to be proved inductively.

The high-level algorithm to inductively compute invariants is shown in Figure 1. In the algorithm, d is the maximum value for the induction depth, and Care represents a set of external constraints to restrict the space while checking for invariants. BDD-based approximate reachable states are an example of external care set. The INDUCTIVEINV function returns the set P of invariants that have been proved inductively.

```
1   INDUCTIVEINV (TR, S0, L, d, Care)
2     L = L ∪ RANDOMSIMINV (TR, S0)
3     P = ∅
4     for k = 0 to d
5       // Proving Step
6       P = P ∪ CHECKINV (TR, Care, L, k)
7       L = L \ P
8       // Base Case
9       L = CHECKINV (TR, S0, L, k)
10    return (P)
```

Figure 1. Top level algorithm.

The function first extends L from the pure set of properties under check, with more potential invariants obtained through simulation. Then, it loops through inductive proofs with increasing k depths, where the proving step and the base case are executed. Proved invariants are accumulated into the P set. Note that we start from induction depth 0, i.e., the set L is checked against the full set of states, without any assumption other than the Care set. All redundancies which hold "combinationally" are thus immediately captured.

A unique SAT-based proof function (CHECKINV) is used for the proving (line 6) as well as the base (line 9) case. For this function, we choose the implementation presented in [7] and reported in Figure 2. We use subscripts to indicate time-frames, i.e., TR_i means $TR(s_i, s_{i+1})$, where $s_0, s_1, \ldots, s_k$ indicate a sequence of states. Similarly, I_0 represents the (current) set of initial states, expressed in the first time frame, and L_i denotes the conjunction of all the candidate invariants in time frame i.

```
1   CHECKINV (TR, I, L, k)
2     do
3       f = I0 ∧ (⋀_{i=0}^{k-1} TRi ∧ Li) ∧ (TRk ∧¬ Lk)
4       cex = SOLVE (f)
5       if (cex ≠ ∅)
6         L = VALIDINV (Lk, cex)
7     while (cex ≠ ∅)
8     return (L)
```

Figure 2. Inductive invariant checking.

For each iteration of the main loop, the propositional formula f is built and verified with a SAT solver. The target of the SAT solver is to falsify at least one of the potential invariants in the final time frame ($\neg L_k$), when all the invariants are true in the previous ones. If such an invariant can be found, then the instance is satisfiable. The produced counter-example is used by function VALIDINV to refine the set of potential invariants for the next iteration, by keeping only the ones which have not yet been disproved.

As noticed in previous works, a key feature for efficiency is the use of incremental SAT. It essentially consists in reusing (partially or completely) the conflict clauses across different SAT calls. We also adopt this mechanism in our formulation using the *unit assumptions* strategy (see [6] for further details).

3. Equivalences and Implications

We propose in this section a specialized version of the algorithm presented in Section 2 to deal with both equivalences and implications. We start by considering only equivalences. Then we describe how we handle implications.

3.1. Equivalence classes

In principle, the algorithm of Section 2 may work with any kind of invariants, including equivalences. The drawback is that the number of candidate invariants to be checked may be too large, being potentially quadratic in the size of the network. In order to efficiently reduce the number of checks, we adopt the idea introduced in [3] for latch mapping in equivalence checking. We assume that the initial random simulation partitions all the nodes (or their negations) in a set of *equivalence classes*, characterized by the following properties:

- Given two distinct classes C_i and C_j, they do not overlap, i.e., $C_i \cap C_j = \emptyset$

- Given any two circuit nodes n_i and n_j and an equivalence class C, if both the nodes belong to C, they are equivalent, i.e., $n_i \in C \wedge n_j \in C \Rightarrow n_i = n_j$

With abuse of notation, we call L the set of equivalence classes. The key idea of [3] is to start with an initial set of equivalence classes, and to iteratively refine them by splitting, until a fix-point is reached. In our case, this is done in the following way.

First, we represent all equivalences by means of a sufficient, i.e., linear, number of equivalences. More specifically, for each class C, we randomly choose a class representative, or leader, l_C. We then explicitly represent equivalences between all class members and l_C. We call such equivalences atomic. All other equivalences can be transitively derived from the atomic relations. The union of all the atomic equivalences provides the set of potential invariants to be checked.

When a SAT counter-example is found in function CHECKINV, at least one equivalence has been falsified. We thus loop through all classes, refining them. For each class C, all nodes satisfying their atomic equivalence with l_C are actually kept in the class itself. All other nodes (if any) are removed from the class and collected into a new one, to be finally added back to L. The pseudo-code for such a "split" algorithm is given in Figure 3.

```
1   VALIDINV (L, cex)
2     for C ∈ L
3       N = ∅
4       for n ∈ C
5         if (cex(n) ≠ cex(l_C))
6           C = C \ n
7           N = N ∪ n
8       if (|N| > 0)
9         L = L ∪ N
10    return (L)
```

Figure 3. Refining the equivalence classes.

3.2. Managing implications

As previously mentioned, the authors of [5] handle sets of implications. Equivalences are possibly obtained from the subset of proved invariants.

We explicitly represent potential equivalences by means of equivalence classes. As a consequence, we may consider a smaller set of implication invariants, because implications between leaders are sufficient.

Hence, let us assume now that function CHECKINV receives a set of potential invariants made up of equivalence classes and implications among them. The inductive proof algorithm may proceed exactly as previously described, up to the SAT counter-example analysis. Function VALIDINV is now in charge for both refining the equivalence classes and for updating the set of candidate implications. This implies removing falsified implications from the potential list, but also adding new potential implications, arising as a result of the class splitting process. More in detail, whenever an equivalence class is split, the two newly generated partitions are still related by an implication (the one compatible with the current SAT counter-example). Furthermore, some more implications may be generated.

Example 1 *Assume that function* CHECKINV *is called with a single potential equivalence class X_0, made up of* 5 *nodes, namely (x_1, . . ., x_5). Let be x_1 the class representative.*

The initial problem we provide the SAT solver is: $(x_1 \neq x_2) \vee (x_1 \neq x_3) \vee (x_1 \neq x_4) \vee (x_1 \neq x_5)$. *Let us suppose that the solver returns the counter-example:* $x_1x_2x_3x_4x_5 = 10110$. *Hence the original class X_0 is partitioned into* $X_1 = \{x_1, x_3, x_4\}$ *and* $X_2 = \{x_2, x_5\}$, x_1 *and* x_2 *being the two leaders. Furthermore, the "implication"* $x_1 \vee \overline{x_2}$ *is generated and added to the candidate invariant set.*

The second call to the SAT solver is done to solve the problem: $(x_1 \neq x_3) \vee (x_1 \neq x_4) \vee (x_2 \neq x_5) \vee (\overline{x_1} \wedge x_2)$. *Assume that now the solver returns the counter-example* $x_1x_2x_3x_4x_5 = 01011$. *The equivalence class X_2 remains unchanged, whereas X_1 is split into* $X_3 = \{x_1, x_3\}$ *and* $X_4 = \{x_4\}$. *Such a split introduces also the new implication* $\overline{x_1} \vee x_4$. *The same counter-example also disproves the implication generated at the previous step, but a new implication,* $x_4 \vee \overline{x_2}$, *is created as a result of the splitting process.*

In general, let us assume that two equivalence classes P and Q are given, p and q being their leaders respectively. Assume that $p \vee q$ is one of the candidate invariants (we use hereon the disjunctive notation for implications, based on the known equivalence: $\overline{p} \Rightarrow q \equiv p \vee q$) and, finally, that the current counter-example has caused both P and Q to split, p' and q' being the two representatives of the newly created classes. Then, Table 1 schematically reports the set of generated implications, according to the value of p and q in the counter-example. Implications marked with an 'o' symbol are "alive" for the next SAT iteration.

p	q	$p \vee q$	$p \vee q'$	$p' \vee q$	$p' \vee q'$
0	0	x	o	o	o
0	1	o	x	o	o
1	0	o	o	x	o
1	1	o	o	o	x

Table 1. Implications and class splitting.

Note that no negation signs have been considered in Table 1, whereas the complete algorithm appropriately deals with the general case.

Albeit the previous procedure provides a dynamic approach for updating the candidate invariant set, the total number of possible implications in a circuit is still quadratic in the circuit size. In order to attack large problem instances, and to provide a scalable solution, we adopt two techniques introduced in [5]. The first one, consists in a transitive reduction of the graph representing the implications. The aim is to filter out candidate implications that can be derived transitively from the other ones. The second technique is the windowing mechanism, i.e., we iteratively consider invariants just on a subset of the circuit nodes.

4. Inductive Invariants and Model Checking

As an initial remark, let us observe that inductive invariants can be exploited for circuit optimization, as shown in [5]. In this section, however, we explore more dynamic approaches exploiting inductive invariants, by combining them with other symbolic model checking techniques, such as BDD-based reachability and verification.

4.1. Approximate Reachability

Invariants have been used in [5] to compute an over-approximation of the reachable states, with a more modular effort than BDD-based reachability. Here we argue that the two approaches can be intertwined in order to get further benefits, especially in terms of accuracy of the result. From the one hand, inductive invariants can be exploited as an external constraint in order to make BDD-based approximate reachability tighter. On the other hand, BDD-based approximate states can help finding more inductive invariants.

```
1   APPROXTRAVIMPROVED (TR, S0, d)
2     R+ = 1
3     do
4       L = INDUCTIVEINV (TR, L, ∅, d, R+)
5       TR = OPTIMIZE (TR, L)
6       R+ = R+ ∧ L
7       R+ = R+ ∧ APPROXTRAV (TR, S0, R+)
8     while (¬ (fixPoint ∨ timeLimit ∨ spaceLimit))
9     return (R+)
```

Figure 4. Improved Approximate Traversal.

The improved reachability procedure is shown in Figure 4. The function iterates through inductive invariant computations and approximate traversals. Both procedures accept the current R^+ expression as a "care" parameter, that is used to enforce invariant proofs and to tighten reachability. Function OPTIMIZE returns an optimized TR starting from the current one and the set of available invariants.

4.2. Circuit based quantification

In the field of circuit based cofactoring [1, 8, 4], inductive invariants and over-approximate reachable states can play a key role as care set for backward reachable state sets [10].

```
1   BACKVERIF (TR, S0, p, d)
2     Care = INDUCTIVEINV (TR, L, p, d, 1)
2     Reached = From = New = ¬ p
3     while (SOLVE (New))
4       if (SOLVE (From ∧ S0))
5         return (FAIL)
6       To = PREIMAGE (TR, From, Care)
7       New = (To \ Reached) ∧ Care
8       From = BESTAIG (To, ¬ Reached ∧ Care)
9       Reached = Reached ∨ To
10    return (PASS)
```

Figure 5. Backward traversal with care.

The approach, adapted to SAT manipulations, is shown in Figure 5, where p is the property to prove. In this case, the main optimization, related to the Care set, is embedded within the PREIMAGE procedure. Based on circuit composition and existential (circuit based) quantification of primary inputs, the procedure can exploit the Care set in circuit optimizations based on redundancy removal. In our experience the benefits can be strong, sometimes making the real difference for the applicability of the approach. Function BESTAIG returns the smallest AIG representing a set of states included between To and ¬ Reached ∧ Care.

5. Experimental Results

Our experiments ran on a Pentium Dual Core 3 GHz Workstation with 3 GByte of main memory, running Debian Linux. We present results on some standard benchmarks belonging to the ISCAS suites, to the VIS distribution, and to the IBM Formal Verification Benchmark Library. We also present some results on industrial circuits coming from STMicroelectronics.

5.1. Invariants Computation

Table 2 reports some evidence on invariants computation. In this case, only circuits coming from the ISCAS suites have been considered, and all the experiments run with an induction depth equal to 1.

The table is conceptually divided into three sections. The first one gives the original circuits details, in terms of number of primary inputs, latches and total combinational gates. In the second section, we performed the analysis enabling equivalences only: the number of proved (atomic) equivalences is then specified, together with the total time for the

computation and the size of the optimized network. The third section reports data for the case in which implications are enabled too, with a time limit of 5 minutes. More specifically, the number of proved equivalences (*beyond* the previous ones) and of proved (atomic) implications is provided, as well as the total computation time and the size of the optimized circuit. Notice that we performed only a combinational optimization step, i.e., implications are not used to "sequentially" simplify the circuit as done in [5].

Furthermore, Table 2 only reports the number of implications that could be proved at depth 1, but not at depth 0. The reason is that, unlike equivalences, we deem all the implications found with induction depth 0 as useless. For instance, for circuit s6669, a total number of 12981 implications was actually proved. However, removing all the non-atomic implications and those proved with $k = 0$, only 3 implications remain. For the largest circuits the process timed-out while it was still proving combinational implications (we reported 0 as proved invariants and "$-$" as optimized circuit size).

The data in the table show that the running time is usually quite small, though the method can lead to a significant reduction in terms of circuit size.

5.2. Approximate Reachability Analysis

In this section we present results on approximate reachability analysis, by following the algorithm of Figure 4. In other words, we use invariants (the ones computed in Section 5.1) to constrain over-approximate reachable state sets.

Table 3 presents our data. The table reports, after the global number of states (equal to 2^{FF}), the number of states reached with the original approximate reachability analysis, and the ones obtained considering equivalence and then implication invariants.

In the table, ovf means time out (after 1800 seconds). and "$-$" data not available (because of the time overflow or the data missing in Table 3). The times necessary to tight the original over-approximate reachability analysis are usually reasonable compared to the advantages (often several orders of magnitude) obtained. In other words, equivalence provide a very good constraint to improve the quality of over-approximate reachable state sets evaluated with BDDs. Implications restrict results even more, but can be more expensive to compute.

5.3. Property Verification

In this section we present results obtained with a tight integration of invariants computation, with forward and backward BDD-based reachability analysis, SAT-based interpolation and circuit-based quantification.

Table 4 reports our results. Column Method indicates the verification method adopted; CBQ stands for AIG circuit-based quantification [1, 8, 4], ITP for interpolant-based [9], and BDD for forward or backward BDD-based verification. Finally, IND means that inductive invariants are sufficient to complete the task. Columns No Inv and Inv report verification time (in seconds) without and with invariants.

Notice that for each experiment we ran all the above techniques with a time limit of 1800 seconds. After that, we present in the table only the verification methodology delivering the best result. We specifically target experiments where invariants alone are not sufficient to complete the verification, though a few runs of this kind are also reported.

Model	FF	Nodes	Method	No Inv	Inv
Ns2	67	2949	ITP	363	148
Ns3	103	4552	CBQ	ovf	565
Blackjack	103	3979	ITP	1221	46
Soap_1	140	2821	BDD	681	124
Soap_2	140	3572	IND	ovf	17
31_1_batch_1	122	1510	BDD	518	311
31_2_batch_1	122	1506	ITP	45	10
IndustrialA	234	1755	CBQ	ovf	389
IndustrialB	670	4040	IND	ovf	18
IndustrialC	1681	11635	IND	ovf	90

Table 4. Verification results.

Data show that there are cases in which no stand-alone techniques was able to complete the verification, but the integrated method solve them with reasonable memory and time resource. As far as memory is concerned, we do not report explicit data on it but all our experiments ran with less than 1 GByte of main memory.

6. Conclusions

This paper addresses two main topics. First of all, it introduces optimized techniques for speeding-up the computation of inductive invariants by reducing the number of checks performed. Secondly, it proposes a tighter integration of inductive invariants within state-of-the-art model checking procedures. Experimental results address different property verification aspects showing very promising results.

References

[1] P. A. Abdulla, P. Bjesse, and N. Een. Symbolic Reachability Analysis based on SAT-Solvers. In *TACAS 2000 - Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 1785, Springer Verlag*, pages 411–425, Upsala University, Prover Technology, Chalmers University, Sweden, Apr. 2000.

[2] P. Bjesse and K. Claessen. SAT–Based Verification without State Space Traversal. In *Proc. Formal Methods in*

Model	Circuit Stats			Equivalences			Implications			
	FF	PI	Nodes	# Eq.	Nodes	Time	# Eq.	# Impl.	Nodes	Time
s1423	74	17	537	3	533	0.5	8	411	525	93
s3384	183	43	1153	6	1141	1	0	49	1141	51
s4863	104	49	1782	470	1245	2	0	20	1245	6
s5378	164	35	1078	148	913	1	142	1293	742	242
s6669	239	83	2224	8	2216	1	0	3	2216	15
s9234.1	211	36	1956	95	1817	4	0	728	1817	300
s9234	228	19	1960	1368	567	2	0	261	567	117
s13207.1	638	62	3034	1361	1615	10	18	818	1597	300
s13207	669	31	3047	1805	1175	10	86	2117	1060	300
s15850.1	534	77	3798	420	3278	10	0	286	3278	300
s15850	597	14	3805	1946	1712	11	335	3108	1307	300
s35932	1728	35	11956	1056	9460	32	0	0	–	300
s38417	1636	28	10527	838	9650	105	0	0	–	300
s38584	1452	12	12631	5687	6659	234	0	0	–	300

Table 2. Optimization results.

Model	Total States	Over-approx Reachable States					
		Original	Time	Equiv.	Time	Impl.	Time
s1423	$1.89{\cdot}10^{22}$	$7.00{\cdot}10^{20}$	2	$7.00{\cdot}10^{20}$	3	$1.35{\cdot}10^{19}$	4
s3384	$1.23{\cdot}10^{55}$	$3.02{\cdot}10^{49}$	12	$3.02{\cdot}10^{49}$	306	$3.86{\cdot}10^{48}$	61
s4863	$2.03{\cdot}10^{31}$	$5.03{\cdot}10^{26}$	4	$5.03{\cdot}10^{26}$	1	$5.03{\cdot}10^{26}$	1
s5378	$2.34{\cdot}10^{49}$	$1.39{\cdot}10^{38}$	1	$1.13{\cdot}10^{36}$	2	$9.37{\cdot}10^{27}$	20
s6669	$8.83{\cdot}10^{71}$	$2.59{\cdot}10^{67}$	7	$2.59{\cdot}10^{67}$	23	$2.59{\cdot}10^{67}$	23
s9234	$4.31{\cdot}10^{68}$	$8.39{\cdot}10^{23}$	365	$2.34{\cdot}10^{20}$	14	$3.86{\cdot}10^{17}$	14
s9234.1	$3.29{\cdot}10^{63}$	$2.70{\cdot}10^{54}$	498	$2.70{\cdot}10^{54}$	632	–	ovf
s13207	$2.45{\cdot}10^{201}$	$7.29{\cdot}10^{78}$	8	$7.93{\cdot}10^{71}$	20	$1.20{\cdot}10^{68}$	284
s13207.1	$1.14{\cdot}10^{192}$	$4.71{\cdot}10^{127}$	755	$2.78{\cdot}10^{102}$	1434	–	ovf
s15805	$5.19{\cdot}10^{179}$	$1.24{\cdot}10^{100}$	11	$7.96{\cdot}10^{97}$	10	$3.64{\cdot}10^{68}$	23
s15805.1	$5.63{\cdot}10^{160}$	$8.64{\cdot}10^{135}$	9	$2.17{\cdot}10^{135}$	53	$7.99{\cdot}10^{133}$	58
s35932	$1.51{\cdot}10^{520}$	$1.51{\cdot}10^{520}$	13	$2.53{\cdot}10^{468}$	29	–	–
s38417	$3.06{\cdot}10^{492}$	$6.83{\cdot}10^{465}$	169	$2.24{\cdot}10^{438}$	231	–	–
s38584	$1.25{\cdot}10^{437}$	$1.26{\cdot}10^{385}$	212	$1.56{\cdot}10^{361}$	94	–	–

Table 3. Approximate reachability results.

Computer-Aided Design, volume 1954 of *LNCS*, Austin, TX, USA, 2000.

[3] J. R. Burch and V. Singhal. Robust Latch Mapping for Combinational Equivalence Checking. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 563–569, San Jose, California, Nov. 1998.

[4] G. Cabodi, S. Nocco, and S. Quer. Circuit Based Quantification: Back to State Set Manipulation within Unbounded Model Checking. In *Proc. Design Automation & Test in Europe Conf.*, Munich, Germany, Mar. 2005.

[5] M. L. Case, A. Mishchenko, and R. K. Brayton. Inductively Finding a Reachable State Space Over-Approximation. In *Proc. Int'l Workshop on Logic Synthesis*, Lake Tahoe, California, May 2006.

[6] N. Eén and N. Sörensson. Temporal induction by incremental sat solving. In *BMC'03: First International Workshop on Bounded Model Checking*, Boulder, Colorado, July 2003.

[7] R. Fraer, S. Ikram, G. Kamhi, T. Leonard, and A. Mokkedem. Accelerated Verification of RTL Assertions Based on Satisfiability Solvers. In *Proc. HLDVT*, 2002.

[8] M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring. In *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, Nov. 2004.

[9] K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer Aided Verification*, volume 2404 of *LNCS*, pages 250–264, Copenhagen, Denmark, 2002.

[10] I. Moon, J. Jang, G. D. Hachtel, F. Somenzi, J. Yuan, and C. Pixley. Approximate Reachability Don't Cares for CTL Model Checking. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 351–358, San Jose, California, Nov. 1998.

[11] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and SAT Solver. In W. A. Hunt and S. D. Johnson, editors, *Proc. Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 108–125. Springer-Verlag, Nov. 2000.

[12] C. A. J. van Eijk. Sequential Equivalence Checking Based on Structural Similarities. *IEEE Trans. on Computer-Aided Design*, 19:814–819, July 2000.