

# Dynamic Transition Relation Simplification for Bounded Property Checking

Andreas Kuehlmann  
Cadence Berkeley Labs, Berkeley, CA, USA

## Abstract

*Bounded Model Checking (BMC) is an incomplete property checking method that is based on a finite unfolding of the transition relation to disprove the correctness of a set of properties or to prove them for a limited execution lengths from the initial states. Current BMC techniques repeatedly concatenate the original transition relation to unfold the circuit with increasing depths. In this paper we present a new method that is based on a dual unfolding scheme. The first unfolding is non-initialized and progressively simplifies concatenated frames of the transition relation. The tail of the simplified frames are then applied in the second unfolding, which starts from the initial state and checks the properties. We use a circuit graph representation for all functions and performs simplification by merging vertices that are functionally equivalent under given input constraints. In the non-initialized unfolding, previous time frames progressively tighten these constraints thus leading to an asymptotic simplification of the transition relation. As a side benefit, our method can find inductive invariants constructively by detecting when vertices are functionally equivalent across time frames. This information is then used to further simplify the transition relation and, in some cases, prove unbounded correctness of properties. Our experiments using industrial property checking problems demonstrate that the presented method significantly improves the efficiency of BMC.*

## 1 Introduction

Bounded Model Checking [1] has gained significant acceptance in property verification due to its relative robustness in practical applications. For verifying safety properties BMC attempts to find a property violation within  $k$  time steps from the initial state(s). Although exhaustive up to the applied bound, BMC does not guarantee completeness, i.e., proving the correctness of a property for depths 0 through  $k$  does not necessarily imply that no violation will occur at depths greater than  $k$ . Nevertheless, for practical applications BMC has a significant value for *refuting* properties.

For checking whether a property  $P$  can be violated in  $k$  time steps, BMC performs a satisfiability (SAT) check for a formula composed of the predicate for the set of initial states  $I$  with  $k$  copies of the transition relation  $T$  (often referred to as *time frames* or *simply frames*), and the inverse of the property  $\neg P$ . The strength of BMC is mainly based on two facts: First, it avoids expensive variable quantification used in complete model checking [2] to “erase the paths history” required for a breath-first search termination condition. Second, modern SAT solvers [3, 4] are efficient in focusing on relevant parts of the problem and often manage to avoid large parts of the search space.

This paper focuses on improving the actual unfolding process of BMC and its interaction with the SAT engine. In the first part

we present an algorithm that dynamically simplifies the transition relation used for BMC unfolding. For this we show that the transition relation for a given time frame can be simplified using a non-initialized unfolding of the transition relation of all previous time frames as *don't care*. The resulting overall algorithm utilizes a dual unfolding scheme: the first unfolding successively simplifies the transition relation from a non-initialized state and the second unfolding performs the actual checks from the initial state.

In the second part of the paper we describe how the dynamic simplification scheme can be implemented efficiently based on AND/INVERTER graphs (AIG). Here the *don't care*'s are exploited in a particular manner aiming at removing functionally redundant AND vertices. For this purpose we present a refined implementation of a SAT-based equivalence detection algorithm, referred to as *SAT-sweeping*, which is tuned for the given BMC application. As a side benefit, SAT sweeping can automatically identify inductive invariants present in the transition relation by finding equivalences across time frames. These invariants are used to further simplify the transition relation and possibly prove correctness of properties.

This paper is structured as follows: After discussing previous work, Sections 3 and 4 introduce preliminary concepts and some theoretical background, respectively. The next section briefly revisits the use of AND/INVERTER graphs for Boolean reasoning and describes the SAT sweeping algorithm. Section 6 presents the BMC framework and gives implementation details of the algorithms for dynamic transition relation simplification. Sections 7 and 8 provide results and concluding remarks.

## 2 Previous Work

There is a rich set of publications addressing improvements of the original BMC technique. Multiple approaches aim at ensuring completeness, including diameter-based methods [1, 5, 6], abstraction-based techniques that combine BMC with classical symbolic model checking [7, 8], and inductive methods that can prove correctness of  $k$ -step inductive properties as part of the BMC unfolding [9]. Other works improve the SAT engine itself and address efficient encoding schemes [10].

AND/INVERTER graphs (AIGs), which are utilized in this paper, were first proposed in [11, 12]. In [13] they were also applied for BMC with the specific improvement to assert previously proved properties at past time frames in the BMC unfolding [9]. Note that this work and others (e.g. [12, 14]) apply simplification only for the BMC unfolding and do not simplify a non-initialized unfolding of the transition relation as presented in this paper. The main advantage of the latter method is that the simplification of the transition relation can be performed incrementally, meaning that the simplified result for frame  $i$  can be used as starting point for the simplification of frame  $i + 1$ . In contrast, due to the initialization, the time frames of a BMC unfolding are unique and their simplifi-

cation cannot be reused for later frames. This causes a significant amount of redundant work that must be performed repeatedly during each BMC step, e.g. to discover that two AND vertices are invariantly equivalent after a few transitions. Moreover, proving the same equivalence in subsequent BMC unfolding frames typically requires a growing reasoning effort due to the enlarging formula depth.

In [15] a method is presented that reuses clauses of a formula in Conjunctive Normal Form (CNF) which are learned during earlier BMC steps for future unfoldings. This contribution comes closest to our concept of dynamically improving the BMC process. However, the work in [15] is restricted to a CNF-based representation and simply copies learned clauses from previous frames. In contrast, our work applies a graph-based representation of the transition relation which is progressively simplified by restructuring. This approach has a significant advantage as it systematically reduces the size of the representation for the transition relation and avoids the addition of learned clauses which typically slows down Boolean Constraint Propagation (BCP) in SAT and often confuses its decision heuristic.

The concept of using SAT for finding equivalent circuit vertices has been proposed for multiple applications [16, 17, 18, 19, 20]. The SAT *sweeping* algorithm used in this paper provides a particularly efficient implementation for an equivalence-based simplification of AIGs. It is based on a successive equivalence class refinement procedure that combines simulation, carefully scheduled SAT queries, and graph restructuring. Vertex equivalences which have been proved by SAT queries are immediately processed by structurally merging the vertices and rehashing their fanout cone similar to the technique use in BDD sweeping [11]. The counter examples of SAT queries that disprove vertex equivalences are used to progressively build up simulation signatures that functionally discriminate all vertices in the currently processed frame and are also likely to be useful for future frames. In the given application, the performance of SAT sweeping is more robust than BDD sweeping [11] as it scales better for deep circuits occurring in BMC. Furthermore, SAT sweeping can deal with complex input constraints more efficiently, which is critically important in our application.

There are multiple approaches for exploiting inductive invariants in property checking. Previous methods typically require to “guess” such invariants (e.g. the property itself) and then use a SAT solver to validate them [21, 9]. In addition to checking inductiveness of the properties, the presented scheme also detects in a constructive manner whether any other nets are inductively constant. A special case for automatically detecting inductive invariants was developed for functional equivalence checking to find the register correspondences [22, 23, 24]. In [14] this concept was extended for the proving properties. However, these methods search for inductiveness across one time frame only or need to be repeatedly applied for probing for larger induction depths. Our method is orthogonal to these techniques. It is restricted to finding vertex equivalences across time frames of the form  $v[t - i] \leftrightarrow v[t], i > 0$ , however, it can detect them efficiently as a side benefit of the simplification procedure.

We would like to point out the relationship of the presented work with the application of retiming and resynthesis in property verification [25, 26]. Due to the blurring of the state boundaries the dynamic simplification of a non-initialized unfolding of the transition relation achieves similar effects as an interleaved application of retiming and resynthesis. However, the dynamic simplification is to some degree more general as it does not rely on temporary state

boundaries. Furthermore, its simplification across state boundaries is more general and can, for example, identify signals that are inductive invariants.

### 3 Preliminaries

In this section we briefly summarize the basic BMC concepts using the notation of [9]. Let  $s$  denote the set of state variables and  $T(s, s')$  be the predicate for the transition relation from the current state  $s$  to the next state  $s'$ . Furthermore, let  $I(s)$  denote the set of initial states and  $P(s)$  be the safety property to be checked, i.e., we attempt to verify the safety formula  $AG P$ . Bounded model checking is based on an explicit unfolding of the transition relation; we will use the notation  $s_i$  to refer to the copy of the state variables for time frame  $i$ . Then

$$T^k(s_0, s_k) = \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \quad (1)$$

with  $T^0 = 1$ , denotes the predicate for all paths of length  $k$  from state  $s_0$  to state  $s_k$ . Checking whether  $P$  can be violated in exactly  $k$  time steps starting from the initial state can be accomplished by a SAT check of the following formula:

$$BMC_k = I(s_0) \wedge T^k(s_0, s_k) \wedge \neg P(s_k) \quad (2)$$

BMC iteratively searches for violations of  $P$  by successively checking formula (2) for  $k = 0$  up to a given bound  $n$ . Note, that if the checks for  $0 \dots k - 1$  have passed in previous BMC iterations, one can safely modify the path predicate as follows:

$$TP^k(s_0, s_k) = \bigwedge_{0 \leq i < k} P(s_i) \wedge T(s_i, s_{i+1}) \quad (3)$$

with  $TP^0 = 1$ , leading to a modified check at step  $k$ :

$$BMC_k = I(s_0) \wedge TP^k(s_0, s_k) \wedge \neg P(s_k) \quad (4)$$

By additionally checking the formula

$$INV_k = TP^k(s_0, s_k) \wedge \neg P(s_k) \quad (5)$$

at unfolding depth  $k$ , one can detect whether  $P$  is a  $k$ -step inductive invariant. In other words, if the sequence of checks of  $BMC_k$  and  $INV_k$  for  $k = 0 \dots n$  yields unsatisfiability for both formulas at step  $n$ , then correctness of  $P$  is proved by induction.

For completeness, we would like to mention that conditions (4) and (5) can be further tightened by excluding non-simple paths from consideration, i.e., paths that are restricted to visiting each state at most once. The corresponding *simple path* condition is:

$$TP_{simple}^k(s_0, s_k) = \bigwedge_{0 \leq i < k} TP(s_i, s_{i+1}) \wedge \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j \quad (6)$$

and would replace  $TP^k$  in (4) and (5). Additional conditions for further tightening the paths are outlined in [9].

## 4 Dynamic Transition Relation Simplification

### 4.1 Basic Approach

In this section we show how the transition relation can be simplified as the unfolding of BMC progresses. For this we first introduce the *simplify* operator, denoted as  $A(s)|_{B(s)}$ , for two predicates  $A$  and  $B$  as follows:

$$A(s)|_{B(s)} = \begin{cases} A(s) & \text{if } B(s) = 1 \\ \text{don't care} & \text{otherwise} \end{cases} \quad (7)$$

*Don't care* means that a simplification procedure can pick any value for  $A(s)$ . The simplify operator is similar to the classical *constrain* or *restrict* operators [27, 28, 29], however, it does not require any particular choice for the *don't care*. In our implementation, we will use the *don't care* to maximally compact a circuit-based representation of the transition functions. As for the constrain and restrict operators, the following property holds also for the simplify operator:

$$A(s)|_{B(s)} \wedge B(s) = A(s) \wedge B(s) \quad (8)$$

We now introduce two new concepts needed for the dynamic simplification process. Let  $\tilde{T}_i$  denote the *simplified transition relation* at time  $i$  that is obtained by applying the transition relations from 0 to  $i-1$  for simplification. Furthermore, let  $\tilde{T}^i$  denote the predicate for the *simplified path* from 0 to  $i$ .  $\tilde{T}_i$  and  $\tilde{T}^i$  are recursively defined as follows:

$$\begin{aligned} \tilde{T}_1(s_0, s_1) &= T(s_0, s_1) \\ \tilde{T}_i(s_{i-1}, s_i) &= \tilde{T}_{i-1}(s_{i-1}, s_i)|_{\tilde{T}^{i-1}(s_0, s_{i-1})} \\ \tilde{T}^1(s_0, s_1) &= \tilde{T}_1(s_0, s_1) \\ \tilde{T}^i(s_0, s_i) &= \tilde{T}^{i-1}(s_0, s_{i-1}) \wedge \tilde{T}_i(s_{i-1}, s_i) \end{aligned} \quad (9)$$

Notice that  $\tilde{T}_i$  is simply constructed from  $\tilde{T}_{i-1}$  by renaming the variables followed by simplifying it with respect to  $\tilde{T}^{i-1}(s_0, s_{i-1})$ . Figure 1 illustrates the definition of simplified transition relation and simplified path.

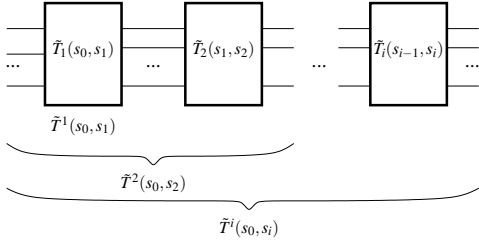


Figure 1: Illustration of the simplified transition relation  $\tilde{T}_i$  and simplified paths  $\tilde{T}^i$ .

The following theorem provides the key insight into our simplification approach.

**Theorem 1**

$$\tilde{T}^k(s_0, s_k) = T^k(s_0, s_k)$$

*Proof:* We show correctness by induction. The base case for  $k = 1$  is trivial. For the inductive step we demonstrate the equality for  $k = i + 1$  by applying definition (9):

$$\begin{aligned} \tilde{T}^{i+1}(s_0, s_{i+1}) &= T^{i+1}(s_0, s_{i+1}) \\ \tilde{T}^i(s_0, s_i) \wedge \tilde{T}_{i+1}(s_i, s_{i+1}) &= T^i(s_0, s_i) \wedge T(s_i, s_{i+1}) \\ \tilde{T}^i(s_0, s_i) \wedge \tilde{T}_i(s_i, s_{i+1})|_{\tilde{T}^i(s_0, s_i)} &= T^i(s_0, s_i) \wedge T(s_i, s_{i+1}) \end{aligned}$$

The last equality can be established by using the induction hypothesis for  $k = i$ , property (8), and noticing that:

$$\tilde{T}_i(s_i, s_{i+1}) = T(s_i, s_{i+1})|_{\tilde{T}^1(s_{i-2}, s_{i-1})|_{\tilde{T}^2(s_{i-3}, s_{i-1})} \cdots |_{\tilde{T}^{i-1}(s_0, s_{i-1})}}$$

and

$$\tilde{T}^j(s_{i-j-1}, s_{i-1}) \supseteq \tilde{T}^i(s_0, s_i) \quad \text{for } 1 \leq j < i$$

□

Note that the recursive definition of (9) corresponds to an iterative construction of the simplified transition relation. During each

unfolding step,  $\tilde{T}_i$  is copied from  $\tilde{T}_{i-1}$  and further simplified with respect to  $\tilde{T}^{i-1}$ . This establishes an accumulative simplification process for the transition relation that maximally avoids replication of effort. Algorithm 1 gives the pseudo-code of a BMC algorithm using dynamic simplification. It applies a dual unfolding —  $\tilde{T}$  refers to the unfolding that progressively simplifies the transition relation whereas the *BMC* unfolding performs the actual checks.

Intuitively, the simplification of  $\tilde{T}_i$  takes advantage of states that are not reachable in  $i-1$  transitions and uses them as *don't care*. Clearly, this process is asymptotic and will yield no further reduction once these states are exhausted. This suggests a practical implementation that unfolds  $\tilde{T}$  for a limited number of frames and then continues to use the last  $\tilde{T}_i$  for further *BMC* unfolding.

Figure 2 illustrates the dual unfolding process used on Algorithm 1. Notice that this procedure is particularly suited for a circuit-based representation of both unfoldings because it allows an efficient implementation of frame copying and their simplification by structural modifications. In Section 6 we outline a specific implementation of this procedure using AIGs.

**Algorithm 1** BMC USING DYNAMIC SIMPLIFICATION

```

1:  $\tilde{T}_1(s_0, s_1) := T(s_0, s_1)$ 
2:  $\tilde{T}^1(s_0, s_1) := \tilde{T}_1(s_0, s_1)$ 
3:  $BMC_0(s_0, s_0) := I(s_0)$ 
4: if SAT( $BMC_0(s_0, s_0) \wedge \neg P(s_0)$ ) then
5:   return FAIL
6: for  $1 \leq i \leq n$  do
7:    $BMC_i(s_0, s_i) := BMC_{i-1}(s_0, s_{i-1}) \wedge \tilde{T}_i(s_{i-1}, s_i)$ 
8:   if SAT( $BMC_i(s_0, s_i) \wedge \neg P(s_i)$ ) then
9:     return FAIL
10:   $\tilde{T}_{i+1}(s_i, s_{i+1}) := \tilde{T}_i(s_i, s_{i+1})|_{\tilde{T}^i(s_0, s_i)}$  {copy + simplify}
11:   $\tilde{T}^{i+1}(s_0, s_{i+1}) := \tilde{T}^i(s_0, s_i) \wedge \tilde{T}_{i+1}(s_i, s_{i+1})$ 
12: return SUCCESS

```

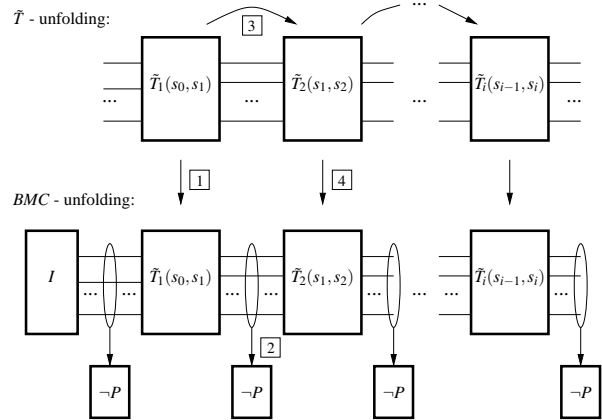


Figure 2: Illustration of the dual unfolding process for BMC with dynamic transition relation simplification. The boxes mark the following parts in Algorithm 1: [1] copy frame  $\tilde{T}_{i+1}$  from  $\tilde{T}$  to *BMC* unfolding (line 7); [2] perform property check (line 8); [3] copy  $\tilde{T}_i$  to  $\tilde{T}_{i+1}$  and simplify (line 10) to build  $\tilde{T}^{i+1}$  (line 11); [4] repeat.

**4.2 Simplification Using Proved Properties**

In the following we extend the concept of simplified transition relation and simplified paths to also consider properties  $P$  that are previously proved for  $0 \leq i < k$ . For this we modify the defini-

tion (9) and introduce the *asserted simplified transition relation*  $\tilde{TP}_i(s_{i-1}, s_i)$  and *asserted simplified paths*  $\tilde{TP}^i(s_0, s_i)$  as follows:

$$\begin{aligned} \tilde{TP}_1(s_0, s_1) &= P(s_0) \wedge T(s_0, s_1)|_{P(s_0)} \\ \tilde{TP}_i(s_{i-1}, s_i) &= P(s_{i-1}) \wedge \tilde{TP}_{i-1}(s_{i-1}, s_i)|_{\tilde{TP}^{i-1}(s_0, s_{i-1}) \wedge P(s_{i-1})} \\ \tilde{TP}^1(s_0, s_1) &= \tilde{TP}_1(s_0, s_1) \\ \tilde{TP}^i(s_0, s_i) &= \tilde{TP}^{i-1}(s_0, s_{i-1}) \wedge \tilde{TP}_i(s_{i-1}, s_i) \end{aligned} \quad (10)$$

The next theorem is a simple extension of Theorem 1 and can be proved in a similar manner:

**Theorem 2**

$$\tilde{TP}^k(s_0, s_k) = TP^k(s_0, s_k)$$

Algorithm 1 can be extended in a straightforward manner to accommodate the use of previously proved properties.

The modified definition (10) takes advantage of property frames previously proved correct and uses them to increase the *don't care* set for the simplification. Note that this can have a significant effect on the simplification potential, especially if there are many states violating the property (i.e., the offset of  $P$  is small). Furthermore, asserting  $P$  is of particular advantage when AIGs are applied for Boolean reasoning — the resulting constant propagation often simplifies the graph structure significantly.

**4.3 Simple Path Constraints**

The set of *don't care* can be further enlarged by simple paths constraints as given in formula (6). The simple paths constraint adds states to the *don't care* set of  $\tilde{T}_i$  which are not reachable in  $i - 1$  steps by a simple paths. In practice the simple path constraints are seldom beneficial as they add a large number of constraints to the SAT problem but increase the *don't care* set only marginally. In our experience this leads to an overall unpredictable performance of the algorithm.

**5 Basic Boolean Reasoning**

In the following sections we describe the implementation details for a bounded model checking approach that uses dynamic simplification of the transition relation. We utilize an AIG as proposed for equivalence checking in [11] and also for bounded model checking in [12, 13]. It is used to store and manipulate the transition relation and the unfolded BMC formula. The AIG construction includes simple on-the-fly simplifications such as structural hashing and constant folding which are of significant advantage as they eliminate redundancies during the unfolding process. The actual BMC checks require SAT queries on the AIG which are implemented by an interleaved application of SAT sweeping to simplify the graph structure and SAT checks for deciding the problem [12].

**5.1 AND/INVERTER Graph Representation**

For completeness, this section gives a short summary of the AIG representation and basic operations to manipulate it. Let  $C = (V, E)$  denote an AIG with the set of vertices  $V = \{v_0\} \cup X \cup G$  and set of directed edges  $E \subseteq V \times V$ . Vertex  $v_0$  represents the logical constant 0,  $X$  is the set of inputs, and  $G$  is the set of AND gates.  $v_0$  and all inputs  $x \in X$  have no predecessor, whereas the gates  $g \in G$  have exactly two incoming edges denoted by  $g \rightarrow right$  and  $g \rightarrow left$ . The attribute  $INVERTED(e)$  indicates whether the function referenced by edge  $e$  is complemented. For referring to functions represented

in  $C$ , we will utilize complemented and non-complemented reference pointers to graph vertices. The semantical interpretation of  $G$  is straightforward.

The AIG is built in topological order from the inputs toward the outputs using a set of construction operators and vertex references. Algorithm NEWVARIABLE creates a new variable and the NOT operator simply toggles the INVERTED attribute of a reference. We will use the functions Deref, TYPE, and INVERTED to access the vertex referenced by a pointer, the type  $TYPE \in \{Zero, Input, And\}$  of a vertex, and the complementation attribute of a reference, respectively. LEVEL( $v$ ) provides for vertex  $v$  the longest paths to any input.

The pseudo-code for the AND operator is shown in Algorithm 2. It takes two vertex reference as input arguments and applies first constant folding (lines 1-6) followed by a hash table lookup (line 9) for identifying simple redundancies. If it is not structurally redundant, a new AND vertex is created and added to the hash table. The meanings of functions RANK, HASHLOOKUP, INSERTHASH, and NEWANDVERTEX are self-explanatory.

---

**Algorithm 2** AND ( $l, r$ )

---

```

1: if  $l = v_0 \vee r = v_0 \vee l = \text{NOT}(r)$  then {Constant folding}
2:   return  $v_0$ 
3: if  $l = \text{NOT}(v_0)$  then
4:   return  $r$ 
5: if  $r = \text{NOT}(v_0) \vee l = r$  then
6:   return  $l$ 
7: if  $\text{RANK}(l) > \text{RANK}(r)$  then {Sort to catch commutativity}
8:    $temp := r; r := l; l := temp$ 
9:  $v := \text{HASHLOOKUP}(l, r)$ 
10: if  $v = \perp$  then {Hash miss}
11:    $v := \text{NEWANDVERTEX}$ 
12:    $v \rightarrow left := l$ 
13:    $v \rightarrow right := r$ 
14:    $\text{INSERTHASH}(l, r, v)$ 
15: return  $v$ 

```

---

A key transformation on the AIG is the MERGE operator. It is used to declare two vertices as functional equivalent (e.g. as a result of a SAT check) and performs the corresponding changes in the graph. The implementation of MERGE moves all outgoing edges from one vertex to the other and rehashes the fanout structure towards the outputs until all changes are propagated.

**5.2 SAT Sweeping**

Boolean reasoning can be performed efficiently on an AIG by interleaved applications of graph simplification from the inputs and SAT checks from the output [12]. In previous work [11], BDD sweeping was proposed for the simplification step which applies heap controlled BDD propagation to identify functionally identical vertices and merge them using the MERGE operator. In this section we describe *SAT sweeping*, which achieves the same goal by an interleaved application of a word-parallel simulator and CNF-based SAT solver [4] (though a circuit-based SAT solver would be equally applicable). In the given application, SAT sweeping is more efficient than BDD sweeping as it can deal robustly with deep circuit graphs and also handle complex input constraints

Algorithm 3 outlines the basic flow of SAT sweeping. The goal of the algorithm is to identify all pairs of functionally equivalent vertices and merge them in the graph. For this a set of simulation vectors is applied for refining the vertex equivalence classes

---

**Algorithm 3** SAT SWEEPING

---

```
1: {Given: Circuit graph  $C = (V, E)$  with inputs  $X$ }
2: randomly initialize simulation vectors for all  $x \in X$ 
3:  $Clusters := \{V\}$  {Initially all vertices in single cluster}
4: while (1) do
5:   simulate  $C$  to update all vertex signatures  $sig(v)$ 
6:    $\forall v \in V : phase(v) =$  first bit of simulation vector
7:   refine partitioning  $\{V_1, V_2, \dots, V_c\}$  s.t.
    $\forall u, v \in V_i : sig(u) \oplus phase(u) = sig(v) \oplus phase(v)$ 
8:   if  $\forall i : |V_i| = 1 \vee (UsedResource > ResourceLimit)$  then
9:     return
10:  else
11:    for all  $i : |V_i| > 1$  do {Once per cluster}
12:      select  $u, v \in V_i$  s.t.
         $(u, v) = argmin \max(LEVEL(u) \text{ and } LEVEL(v))$ 
13:       $res := SAT\_CHECK((u \oplus phase(u)) \oplus (v \oplus phase(v)))$ 
14:      if  $res = SAT$  then
15:        extend simulation vector for all  $x \in X$  by
        SAT counter-example
16:      else if  $res = UNSAT$  then { $u$  and  $v$  equivalent}
17:         $MERGE(u \oplus phase(u), v \oplus phase(v))$ 
18:        remove  $v$  from  $V_i$  { $u$  is representative for  $v$  in  $V_i$ }
```

---

$\{V_1, V_2, \dots, V_c\}$ . During each iteration of the algorithm a SAT solver is applied for cluster  $V_i$  to check equivalence of two of its vertices (line 13). If the SAT check returns “unsatisfiable”, equivalence is shown. Both vertices are then merged and replaced in  $V_i$  by only one representative (line 17, 18). Note that the MERGE operation reshapes the fanout structure in the graph, which often causes a “chain reaction” of identifying more equivalences which do not need to be proved by additional SAT queries.

If the SAT check returns “satisfiable” the counterexample expressed in terms of the inputs is appended to the set of simulation vectors (line 15). These vectors typically provide “interesting” corner cases which have a strong chance to break many clusters apart in the next refinement iteration (line 7). This feature combined with the use of vertex merging and rehashing borrowed from BDD sweeping [11] provides a unique advantage of the presented technique in the given application and is not used in previous work suggesting the application of simulation and SAT queries for identifying functional equivalent vertices [16, 17, 18, 19, 20].

In the following we provide a few more implementation details. In order to catch vertex equivalences modulo complementation, we use the variable *phase* which is simply initialized for each vertex by the value of its first simulation vector (line 6). The refinement condition on line 7 is designed such that this approach automatically handles complemented vertices as they have opposite values for *phase*.

The performance of the SAT sweeping algorithm is critically dependent on the scheduling of the SAT checks on line 13. It is important to postpone checks that are deep in the graph as long as possible as they are time consuming and may not finish. Therefore, during each iteration the two most “shallow” vertices are selected from each cluster  $V_i$  (line 12). A refinement of this strategy handles only the subset of the  $V_i$  in each iteration which is closest to the primary inputs. This refinement takes advantage of the forward rehashing during MERGE as it often covers SAT checks that are outstanding for vertices deeper in the graph. The shown pseudo-code does not include the resource control required for the SAT check on line 13. For efficiency, one would start with little effort

(e.g. controlled by the number of decisions) and increase it over time.

The algorithm terminates when all clusters contains only one representative vertex. This means that all vertices of the simplified AIG are functionally unique. Furthermore, the set of simulation vectors includes for each pair of vertices at least one vector that distinguishes them. The second, alternative termination condition checks for exceeding the *ResourceLimit* and thus provides a mean for the calling program to control the effort spent in SAT sweeping.

### 5.3 SAT Check for AIG Vertices

Similar to the use of BDD sweeping, SAT sweeping is most efficiently applied in an interleaved application with a direct SAT check for the property to be proved. Algorithm 4 shows a simplified flow adapted from [12].

---

**Algorithm 4** SAT ( $v$ )

---

```
1: while (1) do
2:   if  $v = NOT(v_0)$  then
3:     return  $SAT$ 
4:   if  $v = v_0$  then
5:     return  $UNSAT$ 
6:    $res := SAT\_CHECK(v)$  {Attempt to check vertex directly}
7:   if  $res = SAT$  then
8:     return  $SAT$ 
9:   else if  $res := UNSAT$  then
10:    return  $UNSAT$ 
11:    $SAT\_SWEEPING(ResourceLimit)$  {Simplify graph}
12:   increase  $ResourceLimit$ 
```

---

## 6 Bounded Model Checking

In this section we describe the application of the AIG reasoning presented in the previous section for bounded model checking and will specifically discuss the implementation of dynamically simplifying the transition relation in this framework.

### 6.1 Basic Algorithm

A core algorithms required for bounded model checking using AIGs is a vectored compose operation. Let  $s$  denote a sorted vector of vertex references of the AIG and  $s[i]$  refer to its  $i$ th component. Algorithm 5 gives the pseudo-code for the vectored compose operator. VECTORCOMPOSE builds a new function vector  $s_{res}$  from  $s_f$  by replacing all occurrences of  $s_h$  by  $s_g$ . The parameters  $l_{gh}$  and  $l_f$  represent the length of vectors  $s_g/s_h$  and  $s_f$ , respectively.

As shown in the pseudo-code, the algorithm VECTORCOMPOSE first resets a set of shadow variables for each vertex  $v \in V$  and then initializes the vertices of  $s_h$  with the corresponding entries of  $s_g$ . It then uses the function COPY to recursively build the composed vector  $s_{res}$  from  $s_f$ . The vectored application of compose is important from a practical point of view as it minimizes redundant graph traversals and vertex replications. Furthermore, the embedded application of the AND operation performs constant folding and simple restructuring on the fly.

Next, as an introduction for the more complex version with dynamic simplification, we describe the basic BMC algorithm using an AIG representation. As applied in Section 3, let  $s$  and  $s'$  denote the vectors of AIG references for the set of current and next state variables of the transition relation  $T(s, s')$ , respectively, and  $m = |s| = |s'|$  be the number of state bits.

Algorithm 7 outlines the basic BMC flow using an AIG. First, the graph representation of  $T$  is constructed by initializing the set

---

**Algorithm 5** VECTORCOMPOSE ( $s_g, s_h, l_{gh}, s_f, l_f$ )

---

```
1: for all  $v \in V$  do
2:    $shadow(v) := \perp$ 
3: for  $1 \leq j \leq l_{gh}$  do
4:    $shadow(s_h[j]) := s_g[j]$ 
5: for  $1 \leq j \leq l_f$  do
6:    $s_{res}[j] := COPY(s_f[j])$ 
7: return  $s_{res}$ 
```

---

---

**Algorithm 6** COPY ( $e$ )

---

```
1:  $v = DEREF(e)$ 
2: if  $shadow(v) = \perp$  then
3:   if  $TYPE(v) = Input$  then
4:      $shadow(v) := NEWVARIABLE$ 
5:   else
6:      $shadow(v) := AND(COPY(v \rightarrow left), COPY(v \rightarrow right))$ 
7: if  $INVERTED(e)$  then
8:   return  $NOT(shadow(v))$ 
9: else
10:  return  $shadow(v)$ 
```

---

$s$  with fresh variables and building the components of vector  $s'$  in topological order from  $s$  towards  $s'$  (lines 1 and 2).  $P$  and  $I$  denote the references of the property to be verified and initial state predicate, respectively. The AIG representations for them are built in a similar manner (lines 3-5). For the bounded unfolding of  $T$  we use  $s_i$  to denote the vector of references of the copy of state variables for time frame  $i$  and  $P_i$  for the copy of the property at that frame.

The BMC unfolding is performed by successively composing the current state variables of  $T$  with the tailing set of unfolded state variables  $s_i$  (line 8). Similarly, the copy  $P_i$  of the property is built by composing the variables  $s$  in  $P$  with the tailing state variables  $s_i$  (line 9). It is then checked for satisfiability (line 10). As noted before the use of an AIG provides an efficient implementation for attaching new time frames as it heavily utilizes structural hashing and constant folding.

## 6.2 BMC using Dynamic Simplification

After describing the necessary infrastructure, we can now present the implementation of the details of the dynamic simplification scheme as outlined in Algorithm 8. For clarity we omit the details for using proved properties to further increase the *don't care* set for simplification. However, the necessary updates are straightforward and can easily be derived from definition (10).

The algorithm utilizes a dual unfolding as illustrated in Figure 3. We apply two sets of state variable vectors.  $\tilde{s}_i$  denote the variables

---

**Algorithm 7** BASIC BMC

---

```
1:  $s := NEWVARIABLES$  { current state vars of  $T$  }
2:  $s' := BUILDT(s)$ 
3:  $P := BUILDP(s)$ 
4:  $s_0 := NEWVARIABLES$  { current state vars of BMC frame 1 }
5:  $I := BUILDI(s_0)$ 
6:  $MERGE(I, NOT(v_0))$  { Assert initial state attribute }
7: for  $0 \leq i \leq n$  do
8:    $s_{i+1} := VECTORCOMPOSE(s_i, s, m, s', m)$ 
9:    $P_i := VECTORCOMPOSE(s_i, s, m, P, 1)$ 
10:  if  $SAT(\neg P_i) = SAT$  then
11:    return  $FAIL$ 
12: return  $SUCCESS$ 
```

---

used for the  $\tilde{T}$  unfolding whereas the set of variable vectors  $s_i$  are applied in the  $BMC$  unfolding. Similarly, we use the notation  $\tilde{P}_i$  and  $P_i$  to denote the property in the  $\tilde{T}$  and  $BMC$  unfoldings, respectively. In the pseudo-code, the operator “ $\parallel$ ” denotes vector concatenation.

---

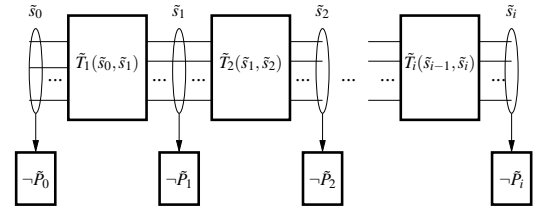
**Algorithm 8** BMC USING DYNAMIC SIMPLIFICATION

---

```
1:  $\tilde{s}_0 := NEWVARIABLES$  { current state vars of  $\tilde{T}_1$  }
2:  $\tilde{s}_1 := BUILDT(\tilde{s}_0)$ 
3:  $\tilde{P}_0 := BUILDP(\tilde{s}_0)$ 
4:  $SAT$  SWEEPING {Simplify  $\tilde{T}_1$  and  $\tilde{P}_0$ }
5:  $s_0 := NEWVARIABLES$  { current state vars of BMC frame 1 }
6:  $I := BUILDI(s_0)$ 
7:  $MERGE(I, NOT(v_0))$  { Assert initial state attribute }
8:  $s_1 := VECTORCOMPOSE(s_0, \tilde{s}_0, m, \tilde{s}_1, m)$ 
9:  $P_0 := VECTORCOMPOSE(s_0, \tilde{s}_0, m, \tilde{P}_0, 1)$ 
10: if  $SAT(\neg P_0) = SAT$  then
11:  return  $FAIL$ 
12: for  $1 \leq i \leq n$  do
13:   $\tilde{s}_{i+1} := VECTORCOMPOSE$ 
14:     $((\tilde{s}_1 \parallel \dots \parallel \tilde{s}_i), (\tilde{s}_0 \parallel \dots \parallel \tilde{s}_{i-1}), i \cdot m, \tilde{s}_i, m)$ 
15:   $\tilde{P}_i := VECTORCOMPOSE$ 
16:     $((\tilde{s}_1 \parallel \dots \parallel \tilde{s}_i), (\tilde{s}_0 \parallel \dots \parallel \tilde{s}_{i-1}), i \cdot m, \tilde{P}_{i-1}, m)$ 
17:   $SAT$  SWEEPING {Simplify  $\tilde{T}_{i+1}$  and  $\tilde{P}_i$ }
18:   $P_i := VECTORCOMPOSE$ 
19:     $((s_0 \parallel \dots \parallel s_i), (\tilde{s}_0 \parallel \dots \parallel \tilde{s}_i), (i+1) \cdot m, \tilde{P}_i, m)$ 
20:   $s_{i+1} := VECTORCOMPOSE$ 
21:     $((s_0 \parallel \dots \parallel s_i), (\tilde{s}_0 \parallel \dots \parallel \tilde{s}_i), (i+1) \cdot m, \tilde{s}_{i+1}, m)$ 
22:  if  $SAT(\neg P_i) = SAT$  then
23:    return  $FAIL$ 
24: return  $SUCCESS$ 
```

---

$\tilde{T}$  - unfolding:



$BMC$  - unfolding:

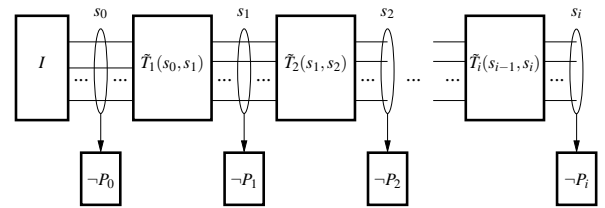


Figure 3: Illustration of the specific implementation of the dual unfolding process of Algorithm 1 using an AIG as outlined in Algorithm 8.

The overall flow of the algorithm is similar to the basic version BASIC BMC. First, the initial frames and property for the  $\tilde{T}$  (lines 1-4) and  $BMC$  unfoldings (lines 5, 8, and 9) are constructed and the predicate for the initial state is built and asserted (lines 6 and 7). After checking the property at the initial states (line 10), the main loop is executed for a given number of iterations. It repeatedly composes a new copy  $\tilde{T}_{i+1}(\tilde{s}_i, \tilde{s}_{i+1})$  from  $\tilde{T}_i(\tilde{s}_{i-1}, \tilde{s}_i)$  (line 13) and

simplifies it using SAT sweeping (line 15). Similarly, a new copy of  $\tilde{P}_i$  is built and simplified (lines 14 and 15). Next the  $\tilde{P}_i$  and  $\tilde{T}_{i+1}$  are attached to the BMC unfolding (lines 16 and 17) and  $\tilde{P}_i$  is checked for satisfiability (line 18).

Note that the interleaving of time frames during the composition of  $\tilde{s}$  and  $s$  during the steps on lines 13,14, 16, and 17. This is crucial for the correctness of the algorithm as vertices of different time frames in the  $\tilde{T}$  unfolding may get merged.

### 6.3 Identification of Inductive Invariants

The SAT sweeping step in Algorithm 8 is applied on the entire  $\tilde{T}$  unfolding. Whenever two AIG vertices of two different time frames that originated from the same vertex in  $T$  get merged an equivalence of the form  $\tilde{v}_{t-i} \leftrightarrow \tilde{v}_i$  is detected. This corresponds to an inductive invariant and can immediately be exploited for simplifying  $\tilde{T}$  by merging the vertex  $\tilde{v}_{t-i}$  in  $\tilde{T}$  with the corresponding vertex  $v_{t-i}$  in the BMC unfolding (which is either constant or some function of primary inputs only). Note that even if  $\tilde{v}$  is not the property, the merging step is beneficial for the simplification of  $\tilde{T}$ . However, if the merging of  $\tilde{v}$  results in a constant 1 for the property  $\tilde{P}_i$  its correctness if proved inductively and the BMC process can be terminated.

## 7 Results

In this section we provide results that validate the efficiency of the overall BMC implementation employing an AIG representation, SAT sweeping, dynamic transition relation simplification, and simplification through induction. We implemented the described scheme using a state-of-the-art SAT solver comparable to [4] for SAT-CHECK in the SAT sweeping algorithm given in Algorithm 3 and the global check outlined in Algorithm 4.

For gaining insight into the actual reduction power of the dynamic simplification scheme we used a set of industrial property checking problems. These problems were produced straight from the language frontend and included a large amount of redundancy. In order to establish a meaningful comparison point, we first compacted these problems using SAT sweeping. We then continued to simplify the transition relation using the  $\tilde{T}$  unrolling mechanism during the actual BMC run. Figure 4 shows how the transition relation  $\tilde{T}_i$  gets reduced with increasing unfolding depth. The plot demonstrates a significant reduction potential of the dynamic simplification scheme. After only a few unfolding steps, the transition relation can be reduced to an average of 62% of the size of the simplified transition relation of the first time frame. Such substantial decrease of the AIG size greatly helps the SAT search during the BMC checks and often results in an exponential reduction of its runtime. This outcome is encouraging and fully justifies the use of dynamic transition relation simplification especially for longer BMC unfoldings where the accumulative gain is significant.

Clearly, the additional effort for the transition relation simplification is not always offset by the gain during the BMC checks. Especially for shorter BMC sequences, the overhead can easily eliminate any performance gain. This observation led to a tuned implementation of the simplification scheme that controls the depth and effort of the transition relation simplification based on the depth of the BMC run, the size of the AIG, and the measured simplification complexity.

We compared such an implementation with a state-of-the-art BMC implementation that is based on a plain CNF translation of the unfolded formulas [1]. Both implementations utilize the same

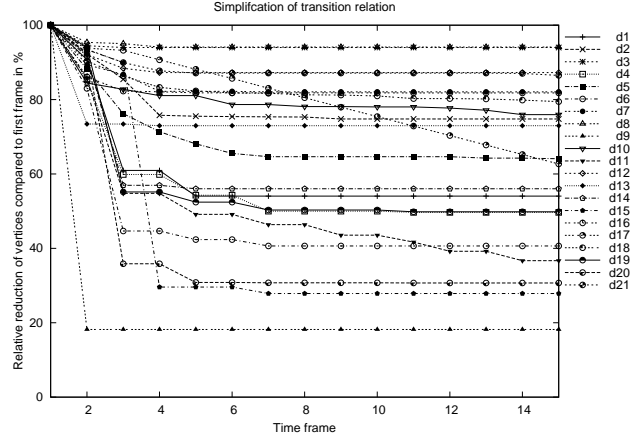


Figure 4: Simplification of the transition relation for the industrial benchmarks used in Table 1. At each time frame the size of the transition relation in terms of AND vertices is compared with the size of the first time frame  $\tilde{T}_1$ .

Design	# State Variables	CEX Length	CNF-based BMC [sec]	AIG + Simpl BMC [sec]
d1	1115	7	133	3.2
d2	165	12	72	2.9
d3	121	4	8.03	0.7
d4	68	8	2.4	0.03
d5	144	21	131	5.8
d6	130	20	47.5	6.2
d7	82	17	16.3	0.6
d8	125	17	27.4	1.4
d9	177	5	38.9	0.1
d10	117	22	30.3	0.9
d11	471	17	80.5	6.06
d12	90	13	46.2	9.2
d13	217	15	7312	169
d14	83	72	3080	467
d15	378	54	440	136
d16	647	41	5584	203
d17	243	42	46	0.1
d18	77	316	2881	12.1
d19	221	55	4699	51
d20	155	1152	63441	56
d21	192	34	52.1	1.01

Table 1: Runtime results comparing state-of-the-art CNF-based BMC with a tuned BMC implementation based on AIG reasoning, SAT sweeping, dynamic simplification, and simplification through induction.

core SAT solver [4] which makes them to some degree comparable. Table 1 provides an overview of the results on the set of industrial property checking benchmarks. The table lists the number of state variables and the lengths of the shortest counter-example in columns 2 and 3. The fourth column reports the runtime for the CNF-based reasoning scheme and Column 5 gives the runtime for the presented methods based on a tuned combination of the methods presented in this paper.

The reported results clearly show that the presented method is superior when compared with a classical CNF-based implementation.

## 8 Conclusions

In this paper we presented two new contributions for improving bounded model checking. First, we described a BMC method that applies dynamic simplification of the transition relation using a dual unfolding process. Applying previous time frames as constraints, the transition relation is progressively compacted as the BMC unfolding depth increases. This results in an asymptotic compaction of the transition relation, making the BMC checks of future time frame more efficient. In the first part of the paper we provided

a general description of this technique which is independent of the underlying data structures and reasoning mechanisms. This is followed by outlining a specific implementation using a previously proposed AIG representation which is particularly suited for the dynamic simplification method.

Second, we presented SAT sweeping, a simplification procedure that is explicitly tuned for AIGs. SAT sweeping compacts the graph by identifying functional equivalent vertices using a combination of simulation, SAT queries, and structural hashing. The particular advantage of SAT sweeping is its robustness for deep AIGs which are typical for applications in bounded model checking. Furthermore, this technique can handle arbitrary input constraints which makes it generally applicable for the simplification technique described above. The application of dynamic simplification with SAT sweeping can also identify functional equivalences across time frames. This can be applied for inductive reasoning, making BMC complete for special cases.

Our results indicate that a BMC implementation based on AIGs and dynamic simplification can speed up verification runs by several orders of magnitude when compared with a state-of-the-art CNF-based BMC implementation.

Our future work focuses on three parts. First, one can further strengthen the power of inductive reasoning by not only exploiting vertex equivalences of the form  $v_{t-1} \leftrightarrow v_t$  but also utilize implications, i.e.,  $v_{t-1} \rightarrow v_t$ . Here the trade-off between the more expensive analysis [18, 30, 14] and corresponding gain of reasoning power requires special attention. A second important aspect of our future work aims at using the described technique in abstraction-based model checking [8]. Here the challenge is to extract the unsatisfiable core of a BMC check which is significantly more complex in the presence of dynamic graph restructuring. Our third effort explores other applications of the dynamic transition relation simplification, e.g. in logic synthesis. We were surprised by how much the transition relation can be simplified after a few time frames and would like to investigate whether this can lead to better circuit implementations.

## Acknowledgments

The author would like to thank Shuo Sheng and Vic Du for their support to integrate the work into a product setting and the help to perform detailed benchmarks.

## References

- [1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, (Amsterdam, The Netherlands), pp. 193–207, March 1999.
- [2] K. L. McMillan, *Symbolic Model Checking*. Boston, MA: Kluwer Academic Publishers, 1993.
- [3] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th ACM/IEEE Design Automation Conference*, (Las Vegas, Nevada), pp. 530–535, June 2001.
- [4] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Design Automation and Test in Europe*, (Paris, France), pp. 142–149, March 2002.
- [5] J. Baumgartner, A. Kuehlmann, and J. Abraham, "Property checking via structural analysis," in *Computer-Aided Verification (CAV'02)*, (Copenhagen, Denmark), pp. 151–165, July 2002.
- [6] D. Kroening and O. Strichman, "Efficient computation of recurrence diameters," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, January 2003.
- [7] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Formal Methods in Computer Aided Design (FMCAD'02)*, pp. 33–51, Springer-Verlag, November 2002.
- [8] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, (Warsaw, Poland), pp. 2–17, April 2003. LNCS 2619.
- [9] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*, (Austin, TX), pp. 108–125, Springer-Verlag, November 2000.
- [10] A. Gupta, A. Gupta, Z. Yang, and P. Ashar, "Dynamic detection and removal of inactive clauses in SAT with applications in image computation," in *38th Design Automation Conference Proceedings*, (Las Vegas, NV), 2001.
- [11] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th ACM/IEEE Design Automation Conference*, (Anaheim, CA), pp. 263–268, June 1997.
- [12] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design*, vol. 21, pp. 1377–1394, December 2002.
- [13] M. K. Ganai and A. Aziz, "Improved SAT-based bounded reachability analysis," in *Proceedings of the 7th ASPDAC/15th International Conference on VLSI Design*, pp. 729–734, January 2002.
- [14] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design (FMCAD'00)*, (Austin, TX), pp. 372–389, Springer-Verlag, November 2000.
- [15] O. Strichman, "Pruning techniques for the SAT-based bounded model checking problem," in *Correct Hardware Design and Verification Methods (CHARME'01)*, (Livingston, Scotland), pp. 58–70, Springer-Verlag, September 2001.
- [16] D. Brand, "Verification of large synthesized designs," in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 534–537, November 1993.
- [17] W. Kunz, "HANNIBAL: An efficient tool for logic verification based on recursive learning," in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 538–543, November 1993.
- [18] W. Kunz, D. Stoffel, and P. Menon, "Logic optimization and equivalence checking by implication analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 266–281, March 1997.
- [19] E. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT in combinational equivalence checking," in *Design Automation and Test in Europe*, (Munich, Germany), pp. 114–121, March 2001.
- [20] F. Lu, L.-C. Wang, K.-T. Cheng, and R. C.-Y. Huang, "A circuit SAT solver with signal correlation guided learning," in *Design Automation and Test in Europe*, (Munich, Germany), pp. 892–897, March 2003.
- [21] D. Déharbe and A. M. Moreira, "Using induction and BDDs to model check invariants," in *Correct Hardware Design and Verification Methods (CHARME'97)*, (Montréal, Québec, Canada), pp. 203–213, Springer-Verlag, October 1997.
- [22] T. Filkorn, "A method for symbolic verification of synchronous circuits," in *Proceedings of 10th IFIP Symposium on Computer Hardware Description Languages (CHDL'91)*, 1991.
- [23] C. van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Transactions on Computer-Aided Design*, vol. 19, July 2000.
- [24] D. Stoffel and W. Kunz, "A structural fixpoint iteration for sequential logic equivalence checking based on retiming," in *International Workshop on Logic Synthesis*, (Tahoe City, CA), May 1997.
- [25] G. Cabodi, S. Quer, and F. Somenzi, "Optimizing sequential verification by retiming transformations," in *Proceedings of the 37th ACM/IEEE Design Automation Conference*, (Los Angeles), pp. 601–606, June 2000.
- [26] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer Aided Verification*, (Paris, France), pp. 104–117, July 2001.
- [27] O. Coudert, C. Berthet, and J. C. Madre, "Verification of sequential machines using Boolean functional vectors," in *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 111–128, November 1989.
- [28] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *International Workshop on Automatic Verification Methods for Finite State Systems*, (Grenoble, France), Springer-Verlag, June 1989.
- [29] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit state enumeration for finite state machines using BDD's," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 130–133, November 1990.
- [30] D. Stoffel and W. Kunz, "Record and play: a structural fixed point iteration for sequential circuit verification," in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (San Jose, California), pp. 394–399, November 1997.