Interleaved Invariant Checking with Dynamic Abstraction

Liang Zhang[1], Mukul R. Prasad[2], and Michael S. Hsiao[3]

[1] Cadence Design Systems, San Jose, CA
[2] Fujitsu Laboratories of America, Sunnyvale, CA
[3] Dept. of Electrical & Computer Engineering, Virginia Tech., Blacksburg, VA

Abstract. The notion of dynamic abstraction was recently introduced as a means of abstracting a model during the process of model checking. In this paper we show, theoretically and practically, how dynamic abstraction can be used with different algorithms for invariant checking, namely forward, backward and interleaved state-space traversal. Further, we formalize the correctness guarantees that can be made under different invariant checking algorithms operating on a dynamically abstracted model. We report experimental results on industrial strength benchmarks to further demonstrate the power and versatility of this abstraction mechanism in conjuction with interleaved state-space traversal.

1 Introduction

The application of formal verification techniques, such as model checking, to real-life industrial designs, has traditionally been hampered by what is commonly known as the *state explosion* problem. Dramatic increases in the size of digital systems and the corresponding exponential increases in the size of their state space have kept industrial designs well beyond the capacity of current model checkers. Abstraction refinement has recently emerged as a promising technology that has the potential to bridge this verification gap.

The basic idea behind *abstraction refinement* [13] is to verify the property at hand on a simplified version of the given design. This simplified version, or *abstraction*, is generated by removing elements from the original design that are not relevant to the proof of the given property. If the property passes on the abstract model, it is guaranteed to be true on the original design as well. However, if the property fails, counter-examples produced on the abstract model must be checked to see that they are true counter-examples on the original design. If however they are false counter-examples, the model checking process is iterated with another abstract model which approximates the original model more closely. The new abstract model can be obtained either by refinement, which embellishes the current abstraction with more details from the original design [5,22,23] or by re-generating a more detailed abstract model from the original design [11,17]. Usually the challenge in abstraction refinement is to construct as small an abstract model as possible so that the model checker can handle it easily. At the same time, the abstract model should retain sufficient details so that the model

checker can prove the property. Thus, the ideal technique for abstraction refinement is one which achieves a good balance between the *size* of the abstract model and its *accuracy*, with respect to being able to prove the given property.

Most previous work on abstraction refinement-based model checking has used *statically abstracted* models in that the abstract model produced by the abstraction step is never modified by the downstream model checker. The notion of *dynamic abstraction* was introduced in [26] whereby the initial abstract model of the design-under-verification is *further* abstracted during successive image computation steps of the model checking phase. Thus, dynamic abstraction provides for a more aggressive yet potentially more accurate abstraction methodology, effectively allowing the core model checking algorithm to work on smaller abstract models. The idea of dynamic abstraction is premised on the key observation that there may be state elements in the concrete model that are *partially abstractable*, *i.e.*, while a state element is necessary in the proof of the property, it may actually be required only in certain time-frames in the proof. For example, some latches in the design are solely present for initialization purposes and may effectively become *redundant* after a few initial time-frames, with respect to the given property.

The treatment in [26] implemented dynamic abstraction within an invariant checking algorithm using BDD-based forward state space traversal. However, as observed in several previous works such as [3,18], for many problem instances a backward state space traversal or a combination of forward and backward traversals (called *interleaved traversal*) may be significantly faster than a simple forward traversal. Such interleaved traversals can provide a more stable and balanced method of state space exploration and can be especially beneficial for failing instances where the failing trace may be constructed in part through forward and backward traversals respectively [18]. These facts are especially significant in the case of an iterative abstraction refinement framework, since a) abstraction can dramatically alter the state space of the model under verification, and b) *all* but the last iterations in iterative abstraction refinement produce failing models.

In the light of these arguments the main contributions of this paper are:

- We develop algorithms to implement dynamic abstraction within different state traversal techniques namely forward, backward and interleaved traversal, for invariant checking, in an abstraction refinement framework.
- We formalize the correctness guarantees that can be made under different invariant checking algorithms operating on a dynamically abstracted model.
- We present several optimizations to improve the performance of the basic techniques.

This paper is organized as follows. Section 2 surveys related work on abstraction refinement and state space traversal algorithms, followed by some background material in Section 3. In Section 4 we review the notion of dynamic abstraction as proposed in [26] and extend it to integrate different algorithms for state space traversal. We also present several simple but powerful optimizations to improve the performance of the basic algorithms. In Section 5 we present experimental results for the proposed algorithms and conclude the paper with directions for future work in Section 6.

2 Related Work

Abstraction refinement: Abstraction refinement was first introduced by Kurshan [13] for verifying linear time properties. The last few years have seen a lot of research activities on this topic. Abstraction refinement methods can be broadly classified into two categories: 1) counter-example driven and 2) counter-example independent. Counter-example driven methods for abstraction refinement [2,5,7,15,23] typically work by iteratively refining the current abstraction so as to block *a particular* (false) counter-example encountered in model checking the previous abstract model. The refinement algorithm could use a combination of structural heuristics or functional analysis based on SAT or BDDs or some combination of these. A recent paper [8] enlarges the scope of the refinement by using multiple counter-examples from the previous abstract model. This notion is further generalized by Wang *et al.* in [22]. The GRAB tool described there uses a BDD representation for the entire set of shortest counter-examples in the previous abstract model, called *synchronous onion rings (SORs)*. Each iteration (referred to as a *generation*) performs a series of micro-refinements to eliminate all counter-examples represented in the SORs.

Counter-example independent abstraction refinement was introduced in [17] by Amla and McMillan and was also independently discovered by Gupta *et al.* [11]. The basic idea is to perform a SAT-based BMC [6] for the property, upto some depth k, on the original design and then generate the abstract model based on an analysis of the *proof of unsatisfiability* [9,25] of the BMC problem. Essentially, the abstraction excludes latches and/or gates that are not included in the proof of unsatisfiability of the BMC problem and thereby guarantees that the abstract model also does not have any counter-examples upto depth k. Successive abstract models are similarly generated by solving BMC problems of increasing depth. The main contribution of [11], compared to [17], lies in the use of BMC on successively smaller abstract models within an iterative framework so that the unbounded verification methods can have better chance to complete. As comparison, our proposed dynamic abstraction can be applied on top of [11,17] to further reduce the size of the abstract model. Recently, a hybrid scheme [1] has also been proposed to combine the strength of counter-example independent and counter-example driven abstraction refinement. Our current implementation uses counter-example independent abstraction refinement, although our ideas could help in counter-example driven frameworks as well.

Recent papers have also proposed improvements to other aspects of abstraction refinement-based model checking, most notably the concretization test and the granularity of abstraction. The use of BMC to concretize abstract counter-examples was first proposed in [23] and most of the current abstraction refinement frameworks use some variant of SAT or ATPG-based BMC for this task. Bjesse and Kukula [2] proposed an enhancement in that the concrete error trace need not have the same length as the abstract counter-example. Information from the abstract counter-example is used as a guide for the concretization algorithm. Other works [8,21] have generalized the granularity of the abstraction. Cut-points are inserted at selected points in the fanin cones of latches, and are used as abstraction points in addition to latch outputs. Li *et al.* [14] proposed a

new search strategy for the SAT solver so that the proof of unsatisfiability will generate smaller abstract models.

State space traversal algorithms: The second body of work that this paper draws upon, is the combination of forward and backward state space traversals to perform symbolic invariant checking. Several works such as [3,4,10,12] use a combination of forward and backward traversal whereby one sweep of traversal is used to approximate or prune the search space and subsequently the other sweep is used to perform the actual verification. This process may potentially be iterated. However, the work closest to the approach of this paper is that of [18] where forward and backward traversal are used simultaneously in one single pass. The motivation here is that the state space of some problems maybe best suited to backward traversal while others may have a propensity towards forward traversal. The algorithm tries to dynamically make this decision, with minimum overhead, and in the case of a buggy design it may construct the error trace partly through a forward traversal and partly through backward traversal.

Recently, the idea of *dynamic abstraction* was introduced in [26]. Previous work on abstraction refinement differs from dynamic abstraction in two key aspects. In previous work, 1) the abstraction step is algorithmically distinct from the model checking phase, *i.e.*, the abstraction is performed outside the model checker, and 2) the abstraction is purely structural in nature and has no temporal component. For example, the same static abstraction is used for each image computation step in BDD-based model checking. In contrast, dynamic abstraction first analyzes the temporal behavior of various latches. Then, based on the analysis it dynamically and progressively abstracts away a set of latches *during* different steps of model checking. For example, in the case of a BDD-based model checker, progressively more abstracted versions of the transition relation are used for successive image computation steps.

This paper further develops the theory and implementation of dynamic abstraction based invariant checking. In [26] dynamic abstraction was introduced in the context of a basic forward state traversal algorithm. In this work we develop algorithms to implement dynamic abstraction within different state traversal techniques namely forward, backward and interleaved traversal and formalize the correctness guarantees that can be made under different traversal techniques, operating on a dynamically abstracted model. Further, we present several optimizations to improve the performance of the basic techniques. As demonstrated by the experimental results in Section 5, on several industrial benchmarks the integration of dynamic abstraction and interleaved traversal is necessary to complete the verification. In other instances the combination and the proposed optimizations provide a significant performance improvement.

3 Background

For the purpose of this paper we will only consider model checking of invariants *i.e.*, CTL properties of the form $\mathbf{AG}p$ where p is a Boolean expression on the variables of the given circuit model. The circuit under verification can be modeled as a sequential circuit M with primary inputs $W = \{w_1, w_2, \ldots, w_n\}$, present state variables $X = \{x_1, x_2, \ldots, x_m\}$ and the corresponding next state variables $Y =$

$\{y_1, y_2, \ldots, y_m\}$. Thus, M can be represented as $M = \langle T(X, Y, W), I(X)\rangle$, where $T(X, Y, W)$ is the *transition relation (TR)* and $I(X)$ is the set of initial states (more precisely the characteristic function of the set of initial states). M has a set of latches (state elements) $L = \{l_1, l_2, \ldots, l_m\}$. Thus, x_i and y_i would be the present state and next state variables corresponding to latch l_i. The transition relation T is a conjunction of the transition-relations of the individual latches. Thus,

$$T(X, Y, W) = \bigwedge_{i \in \{1 \ldots m\}} T_i(X, y_i, W)$$

Here, $T_i(X, y_i, W) = y_i \leftrightarrow \Delta_i(X, W)$ is the transition relation of latch l_i and $\Delta_i(X, W)$ is its transition function in terms of primary inputs and present state variables.

Given a subset of latches $L_{abs} = \{l_1, l_2, \ldots, l_q\}$, $L_{abs} \subseteq L$, that we would like to abstract away from the design, the abstract model can be constructed by cutting open the feedback loop of latches L_{abs} at their present-state variables X_{abs}, *i.e.* making the variables X_{abs} primary inputs and removing the logic cones of the transition functions of latches L_{abs} from the circuit model. Functionally, the abstracted transition relation $\widehat{T}$ can be defined as

$$\widehat{T}(\widehat{X}, \widehat{Y}, \widehat{W}) = \bigwedge_{i : l_i \in \widehat{L}} \widehat{T}_i(\widehat{X}, y_i, \widehat{W}) \qquad (1)$$

where $\widehat{W} = W \cup X_{abs}$, $\widehat{X} = X - X_{abs}$, $\widehat{Y} = \{y_i : x_i \in \widehat{X}\}$, and for all i such that $y_i \in \widehat{Y}$, $\widehat{T}_i(\widehat{X}, y_i, \widehat{W}) = T(X, y_i, W)$.

A concept that will be frequently used in the sequel is that of the *proof of unsatisfiability (POU)* of a CNF SAT formula. As reported in [9,14,25], modern SAT solvers such as `zchaff` [24] can be modified to produce a proof of unsatisfiability when the CNF formula being solved is found to be unsatisfiable. The proof of unsatisfiability (denoted by $\mathcal{P}$ in the sequel) of an unsatisfiable SAT CNF formula is a sequence of resolution steps which derives the empty clause from the original clauses of the formula. It can be represented as a directed acyclic graph, the nodes of which are clauses and each node (other than the leaves) has precisely two children. The root of this graph is the empty clause and the leaves are clauses from the original CNF. All other nodes (including the root) such that they can be derived through a resolution operation on their two child clauses.

The basic framework for abstraction refinement in our current implementation is similar to the one developed in [17] and [11]. A simplified version of the algorithm used in [17] is shown in Algorithm 3.1. The basic algorithm used in [11] is similar to this except that the abstraction (line 5 and 6) is performed multiple times in an inner loop.

Let v be a variable in the representation of the transition relation T. A k-step BMC problem is generated by unrolling and replicating T, k times. Let $v^1, v^2, \ldots, v^k$ denote the k instantiations of v in the unrolled BMC problem. The idea of the abstraction is to solve a k-step SAT-BMC [6] problem formulated on the original design and to analyze the POU returned by the SAT solver to generate the abstraction. The POU is analyzed to identify a set of latches L_{abs}

```
 1: k = InitValue
 2: if SAT-BMC(M, p, k) is SAT then
 3:    return "found error trace"
 4: else
 5:    Extract proof of unsatisfiability, P of SAT-BMC
 6:    M' = ABSTRACT(M, P)
 7: end if
 8: if MODEL_CHECK(M', p) returns PASS then
 9:    return "passing property"
10: else
11:    Increase bound k
12:    goto Step 2
13: end if
```

Algorithm 3.1: Abstraction Refinement Using SAT-BMC [17]

such that for each $l \in L_{abs}$ the variables $l^1, l^2, \ldots, l^k$ do not appear in any of the clauses of the POU. These latches can then be abstracted away using Equation 1 (in the ABSTRACT operation in Step 6 of Algorithm 3.1). The rationale is that since these latches provably do not contribute to the property check in the first k time-frames, they might be irrelevant from the point of view of deciding this property for unbounded behaviors as well.

The model checking algorithm employed in Algorithm 3.1 (Step 8) may use a variety of methods. For the purpose of this paper we will assume a symbolic model checker using BDDs [16]. However, the ideas can easily be applied to other model checking methods such as SAT-based state-space traversal or even SAT-BMC. Algorithm 3.2 shows the pseudo-code for a generalized symbolic invariant checking algorithm that could use BDDs. The algorithm implements a combination of forward and backward state space traversal, henceforth referred to as *interleaved traversal* (after [18]). The traversal can be made a purely forward traversal, a purely backward one, or any interleaved combination of the two by an appropriate implementation of the *Choose_Direction* function (line 7 of Algorithm 3.2). Here B denotes the "bad states" *i.e.*, states that violate p and S_N^f and S_N^b denote the set of states currently reached through the forward and backward traversals respectively. Basically, the algorithm iteratively furthers the traversal in a direction governed by *Choose_Direction* till either the two traversals intersect (line 4) or either of the sets S_N^f or S_N^b reaches a fixpoint (line 3). A possible greedy heuristic for *Choose_Direction* would be to select the easier direction for image computation, which may be gauged by the final and/or peak BDD size of current image step, and other factors. The core steps implementing the traversal are the image (*Img*) and pre-image (*PreImg*) operations, defined in Equations 2 and 3 respectively. The *Img* operation on a set of states S, computes the states reachable from the states in S, in one step of computation, via the transition relation T. The *PreImg* operation simply performs the inverse computation.

$$Img(S) \equiv \exists X, W.\ S(X) \wedge T(X, Y, W) \tag{2}$$

$$PreImg(S) \equiv \exists Y, W.\ S(Y) \wedge T(X, Y, W) \tag{3}$$

```
Invariant Check(M⟨T, I⟩, B)
 1: S_N^f = I; S_N^b = B; S_C^f = ∅; S_C^b = ∅;
 2: k_fwd = k_bwd = 0; j = 0;
 3: while S_C^f ≠ S_N^f and S_C^b ≠ S_N^b do
 4:    if S_N^f ∩ S_N^b ≠ ∅ then
 5:       return "found error trace";
 6:    end if
 7:    if Choose_Direction() = "forward" then
 8:       S_C^f = S_N^f;
 9:       S_N^f = S_C^f ∪ Img(S_C^f);
10:       k_fwd++;
11:    else {Backward direction chosen}
12:       S_C^b = S_N^b;
13:       S_N^b = S_C^b ∪ PreImg(S_C^b);
14:       k_bwd++;
15:    end if
16:    j++;
17: end while
18: return "no bad state reachable";
```

Algorithm 3.2: Interleaved Symbolic Invariant Checking

4 Dynamic Abstraction

In order to make the paper self-contained, we begin by reviewing the notion of dynamic abstraction as developed in [26]. Given the original circuit M and the property $P = \mathbf{AG}p$ let us assume that a SAT-BMC problem on M of depth k has been solved and there is no counter-example. Further, suppose that the SAT solver generates a proof of unsatisfiability $\mathcal{P}$ for this problem as described in [9,25]. Figure 1, taken from [26], is a graphical representation of the POU from a 40-step SAT-BMC problem on a real circuit example. For each latch (plotted for 40 representative latches on the y-axis) the plot shows the time-frames for which the corresponding instantiation of the latch variable appears in the POU of the SAT-BMC problem. The latches have been sorted on the y-axis for better readability of the data. Given a latch variable $l \in L$, we can define the *redundancy index*, $\rho(l)$ of l, with respect to the proof $\mathcal{P}$, as follows:

Definition 1 (Redundancy Index (RI)). *The redundancy index $\rho(l)$ of latch l with respect to the proof of unsatisfiability $\mathcal{P}$ is the smallest time-frame index such that for all time-frames j, $\rho(l) \leq j \leq k$, there does not exist a clause with variable l^j in $\mathcal{P}$.*

For example, in Figure 1 the points marked **A** and **B** show that latch number 15 has a redundancy index of 15 and latch 32 has a redundancy index of 28. Simply put, the redundancy index is the earliest time-frame at which the given latch stops participating in the POU of the *current* BMC problem. The situation depicted in Figure 1 is quite typical of a large variety of benchmarks we have experimented with. Most latches are not used in all time-frames of the POU. Moreover, there are several latches that are *only* used in the first few time-frames.

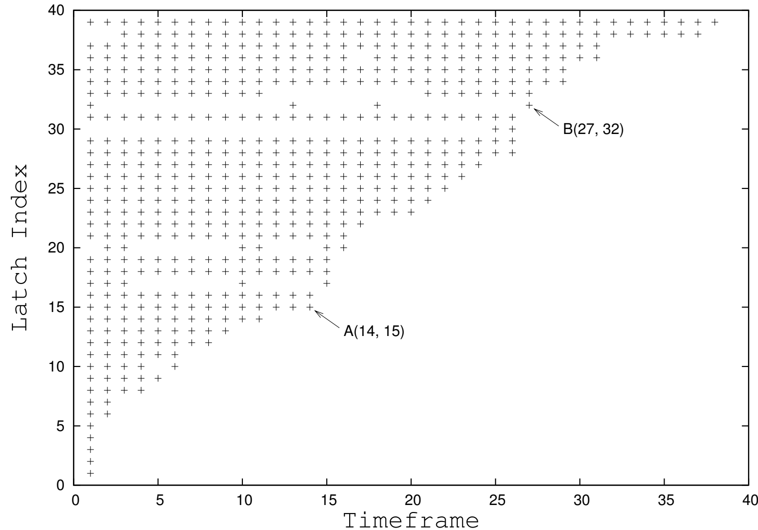

Fig. 1. Latch-based Unsatisfiability Analysis

Note that the redundancy index analysis is typically done with respect to a particular SAT-BMC problem with a certain depth of unrolling. However, we conjecture that, by and large, the redundancy index calculated from a single SAT-BMC run can be a fairly good predictor of the unbounded behavior of a latch with respect to the given property. This intuition is supported by the data shown in Figure 2. The graph plots the redundancy indices for 4 randomly chosen latches from one of our benchmarks, generated from different BMC problems with varying depth k. Apart from minor fluctuations, the RI values of each latch are remarkably consistent, despite having been derived from *independent* BMC runs of *different* depth.

In the next section we develop an algorithm which uses the information about the redundancy indices of various latches to implement dynamic abstraction within state space traversal.

4.1 Interleaved Traversal with Dynamic Abstraction

At each step of image computation (measured by the iteration count variable j in Algorithm 3.2) we can define a *candidate set* of latches available to be abstracted through dynamic abstraction.

Definition 2 (Candidate set). *The candidate set of latches for iteration j of image computation in Algorithm 3.2 is denoted $\mathcal{C}_j$ and is defined as* $\mathcal{C}_j = \{l_i : l_i \in L, \rho(l_i) \leq k_{fwd}\}$

For example, consider the instance in Figure 1 , and suppose $k_{fwd} = 15$, *i.e.*, the *Choose_Direction* heuristic chooses forward traversal for the first 15 steps,

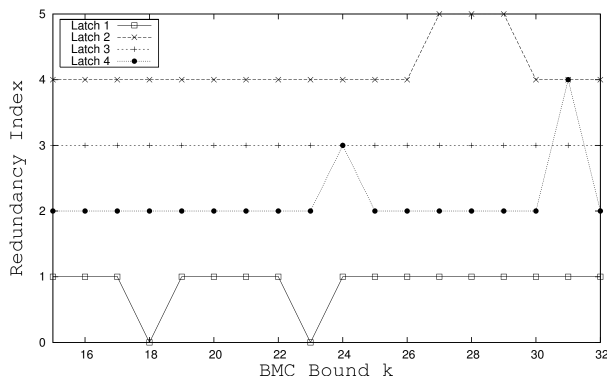

Fig. 2. Redundancy Index as a function of BMC-depth

then for time-frame $j = 15$, the candidate set consists of the first 16 latches, *i.e.*, $\mathcal{C}_{15} = \{l_1, l_2, \ldots, l_{16}\}$, since the RIs of these latches are no greater than 15. A modified version of Algorithm 3.2, incorporating dynamic abstraction, is shown in Algorithm 4.1.

As in Algorithm 3.2, the state space traversal in Algorithm 4.1 can be implemented as purely forward, purely backward or any interleaved combination of these by an appropriate definition of the *Choose_Direction* function (line 11 of Algorithm 4.1). For example, if *Choose_Direction()* always returns "backward" the result will be a purely backward state space traversal. We maintain that an abstraction refinement algorithm employing dynamic abstraction in combination with an interleaved state-space traversal can yield a very powerful framework for invariant checking and this is the focal point of this paper. There are several reasons for this belief. First, as demonstrated in [26] the notion of dynamic abstraction provides a convenient mechanism to abstract away latches that become "redundant" after a few steps of forward state space traversal, thereby focusing the invariant check on a smaller, but not necessarily less accurate, abstract model. On the other hand, [18] and several other works have noted that for certain state spaces, backward state traversal may be far more efficient than forward traversal. Further, in some failing instances, the optimum means of constructing an error trace may partly though a forward traversal and partly through a backward one. These two optimization mechanisms appear to be fairly complementary, especially since dynamic abstraction can simplify the transition relations for both forward and backward traversals. Thus, a combination of dynamic abstraction and interleaved traversal appears to be a reasonable means of harnessing the power of both. Second, as noted in [18] and also confirmed by our own experiments, interleaved traversal can work especially well in the case of failing properties since the forward and backward traversals can co-operate in the task of discovering the error trace. Given the fact that *all* but the last model produced during iterative abstraction refinement are failing instances, it seems to be an ideal candidate to benefit from the use of interleaved traversal.

```
Interleaved_Invariant_Check_DynamicAbstract (M⟨T, I⟩, B)
 1: S_N^f = I; S_N^b = B; S_C^f = ∅; S_C^b = ∅;
 2: k_fwd = k_bwd = 0; j = 0;
 3: while S_C^f ≠ S_N^f and S_C^b ≠ S_N^b do
 4:    L_abs = CHOOSE_ABSTRACTION_LATCHES(L);
 5:    T = ABSTRACT_TR(L_abs, T);
 6:    S_N^f = ∃X_abs . S_N^f;
 7:    S_N^b = ∃X_abs . S_N^b;
 8:    if S_N^f ∩ S_N^b ≠ ∅ then
 9:       return "found error trace";
10:    end if
11:    if Choose_Direction() = "forward" then
12:       S_C^f = S_N^f;
13:       S_N^f = S_C^f ∪ Img(S_C^f);
14:       k_fwd++;
15:    else {Backward direction chosen}
16:       S_C^b = S_N^b;
17:       S_N^b = S_C^b ∪ PreImg(S_C^b);
18:       k_bwd++;
19:    end if
20:    j++;
21: end while
22: return "no bad state reachable";
```

Algorithm 4.1: Interleaved Symb. Invar. Checking with Dynamic Abstraction

Our specific implementation of the interleaved traversal is discussed below, after a discussion of the CHOOSE_ABSTRACTION_LATCHES() function (line 11 in Algorithm 4.1).

As discussed in [26] a key determinant of the performance of dynamic abstraction is the latch selection heuristic CHOOSE_ABSTRACTION_LATCHES, that decides which latches, out of the current candidate set, should be abstracted at a given image computation step. The following definition of the latch selection heuristic allows us to place some correctness guarantees (Theorem 1 below) on the dynamically abstracted model.

Definition 3 (Choose_Abstraction_Latches()). *This heuristic chooses a subset of latches in $\mathcal{C}_j$ that have not already been abstracted away in previous iterations, for abstraction in the current iteration j of image computation in Algorithm 4.1.*

Our interleaving heuristic *i.e.*, the *Choose_Direction* function in Algorithm 4.1 maintains a cost for each direction of traversal (*i.e.*, forward and backward) based on peak BDD size and the increment of BDD size in the last image computation in that direction. In each iteration, the direction with the lower previous cost is chosen for traversal, with the exception that the the first few steps of traversal are always in the forward direction. This is because as per Definition 2 the candidate set open for abstraction, for both forward *and* backward traversal grows monotonically with the number of *forward* image computations performed. Thus, after

a few initial forward images, the transition relation is usually reduced through dynamic abstraction and this same reduced transition relation (*i.e.*, its inverse) is used for potential backward traversal, thereby enhancing its efficiency.

Theorem 1. *Algorithm 4.1 will not find any counter-example to the given property in the first k steps of image computation, where $k = k_{fwd} + k_{bwd}$.*

Proof outline: [4] The proof of this result is along the lines of the main result in [17]. It is based on the observation that the above dynamic abstraction scheme can be equivalently formulated as a SAT-BMC by unrolling T for k time-frames and then cutting open the the unrolled latches in the corresponding time-frames at which these latches were abstracted away by the dynamic abstraction algorithm. It can be shown that the original POU $\mathcal{P}$ is still fully contained in this abstracted SAT-BMC problem, provided the latches dynamically abstracted during invariant checking conform to Definition 3. Thus, the dynamic abstracted model will not have any counter-example to the given property in the first k steps. Note that the above holds for *any* sequence of *forward* or *backward* choices made by the *Choose_Direction* heuristic in Algorithm 4.1. In that sense Theorem 1 is a strict generalization of the main result of [26].

4.2 Optimizations

In the following we describe some inexpensive optimizations that can be used to further improve the performance of a model checking algorithm that performs dynamic abstraction as described above.
Bypassing the Error Check: A simple corollary of Theorem 1 is the following.

Corollary 1. *The error state check $S_N^f \cap S_N^b \neq \emptyset$ (line 8) in Algorithm 4.1 will always yield* `false` *in the first k iterations of the algorithm, $k = k_{fwd} + k_{bwd}$.*

This simple result obviates the need to perform the error state intersection check (line 8 of Algorithm 4.1) in the first k iterations of image computation. This check can be fairly expensive at deeper image computation steps and/or when the target states are not simply the negation of the property but an enlarged target computed through a few pre-image computation steps. For example, in case of benchmark $D7$ in Table 1 the error-state check at depth 11 costs more than 1600 secs. Note that this result is equally applicable to frameworks that only use static abstraction.
Early Quantification Re-scheduling: Typically the transition relation (TR) is maintained in an implicitly conjoined and partitioned form, along with an early quantification scheduling. During dynamic abstraction some state variables may be quantified out from the TR. As a result, the original early quantification scheduling may become sub-optimal to the modified TR. Our implementation solves this problem by modifying the early quantification scheduling when the TR changes.
Cone of Influence Reduction: As proposed in [17], an abstraction of some latches may create opportunities for further abstraction by applying the standard

[4] The complete proof is omitted for lack of space.

cone of influence (COI) reduction on the abstracted model. While the optimization was proposed in the context of the static abstraction procedure, the same can be done after each abstraction step in our dynamic abstraction algorithm. The key point is that any subsequent abstraction due to the COI reduction *does not* increase the space of allowable behaviors of the design. Thus, the quality of the abstraction is not diminished in any way, but the design becomes smaller and more tractable for the model checking.

5 Experimental Results

Experimental Set-up: We have implemented the proposed algorithms for dynamic abstraction and state traversal within the VIS framework [19]. The static abstraction algorithms of [11,17] have also been implemented in the same framework, for comparison. We use the CUDD package for BDD computations, and `zChaff` [24] as the SAT solver for BMC. The POU extraction in `zChaff` has been modified to perform the analysis necessary for dynamic abstraction.

The following specific heuristic for CHOOSE_ABSTRACTION_LATCHES() (line 4 of Algorithm 4.1), proposed in [26] has been used for the purpose of these experiments. More involved and potentially better options are of course possible and could further enhance the proposed algorithms.
Heuristic: *Dynamically abstract just once at* $\lceil \delta \cdot k \rceil$ *time-steps, (*$0 < \delta < 1$*), and abstract all latches in the candidate set at this point.*

The philosophy behind this heuristic is to minimize the overhead of abstraction by doing it only once and being aggressive by choosing all candidates for abstraction. δ is kept fairly low to increase the likelihood of the latches being redundant for future image computations. We used $\delta = 0.2$ in our experiments.

Since the tools of [11,17] are not publicly available, a direct comparison against those approaches is neither fair nor intended. However, we have attempted to incorporate the essence of these works and used them to derive the initial static model on which dynamic abstraction based invariant checking is applied.

We have tested our tool for safety properties on different modules from four real-life industrial designs. In addition we have also experimented with circuits from the VIS Verilog suite [20] and some of the larger ISCAS89 sequential circuits. In the case of the ISCAS'89 circuits the property is the justification of a randomly generated state.

All experiments were run on a 3.0 GHz Pentium 4 Linux machine with 1G RAM, and a 24 hour time-out limit for each problem. Table 1 shows results for a subset of our benchmarks, representative of most of the interesting scenarios we encountered. *D1 - D9* are problem instances from in-house industrial designs, *blackjack_5* and *vsa16a_7* are benchmarks from the VIS Verilog suite while *s38584_2* is an instance from the ISCAS'89 circuits. The second column shows if the property is a passing property or the length of the shortest counterexample if it is a failing property. Column 6 shows the number of latches in the statically abstracted model, while column 8 reports the number of latches in the final dynamically abstracted model. Column 7 is the cumulative CPU time for iterative static abstraction refinement and invariant checking. Columns 9 and 10

Problem	Pass/ cex leng.	Concrete Model			Static Abstraction		Dynamic Abstraction		
		# PIs	# FFs	# Gates	# FFs	Time (secs.)	# FFs (diff.)	Fwd (secs.)	Intrlvd (secs.)
D1	Pass	85	161	1385	101	472	73(-28)	128	**13**
D2	Pass	118	375	1562	161	>24h(30)	129(-32)	952	**288**
D3	36	289	654	4826	170	>24h(28)	160(-10)	**1069**	1400
D4	29	289	654	4823	201	52333	168(-33)	10098	**617**
D5	60	308	746	3837	123	272	81(-42)	261	**216**
D6	Pass	330	1158	5155	264	67	204(-60)	**41**	58
D7	27	356	1644	7408	257	>24h(11)	244(-13)	>24h(13)	**456**
D8	Pass	1015	2971	10044	286	236	216(-70)	**100**	144
D9	Pass	1950	5564	19161	224	811	187(-37)	**114**	323
blackjack_5	Pass	7	109	1061	95	**460**	94(-1)	18104	7143
vsa16a_7	Pass	34	205	1939	108	24	105(-3)	32	**8**
s38584_2	73	13	615	2575	93	>24h(20)	66(-27)	>24h(39)	**10506**

Table 1. Results: Static Abstraction and Proposed Dynamic Abstraction

report similar times but for dynamic abstraction with forward and interleaved model checking respectively. Both these times include the time for static abstraction as well, since the static abstraction is the starting point for dynamic abstraction based invariant checking. Note that Column 9 represents the results of [26], albeit enhanced with the optimizations of Section 4.2. In cases where the model checking timed out after 24 hours, the number of image computation steps successfully completed, in the last iteration of abstraction refinement, is noted in parenthesis.

Analysis of results: As shown in Table 1, the proposed method of combining dynamic abstraction with interleaved state space traversal, during invariant checking, shows significant improvements over plain static abstraction refinement (column 7) as well as over dynamic abstraction with plain forward invariant checking (column 9), for both passing and failing properties. In some cases, such as *D7*, the improvement can be quite dramatic. In a few cases such as *D3*, *D6*, *D8* and *D9*, the interleaved method is moderately slower than dynamic forward invariant checking. These are passing properties, where the forward direction turns out to be the optimum direction of traversal and any mis-predicted backward traversal performed by the interleaved method merely serves as an overhead. In the very rare case, such as *blackjack_5*, where the both dynamic abstraction methods are slower than plain static abstraction, the dynamic interleaved method actually improves upon the dynamic forward method, thereby offsetting some of the losses incurred in using dynamic abstraction. In that sense the combination of interleaved traversal and dynamic abstraction is a more stable algorithmic configuration. Overall, our conclusions are that the combination of interleaved state space traversal with dynamic abstraction, in a framework for abstraction refinement, can provide significant performance gains over either plain static abstraction or even dynamic abstraction with pure forward traversal. In several cases, this combination can successfully complete the verification where the other methods time-out. The slow-down due to the overhead of mis-predicted interleaved traversals is usually moderate and, in our opinion, an acceptable trade-off considering the stability and significant performance improvements offered by the method.

6 Conclusions & Future Work

The notion of dynamic abstraction was recently introduced [26] as a means of abstracting a model during the process of model checking. In this paper we have extended the theory and implementation of this idea in several ways. We have presented algorithms to implement dynamic abstraction within different state traversal techniques namely forward, backward and interleaved traversal and formalized the correctness guarantees that can be made under different traversal algorithms operating on a dynamically abstracted model. We have also presented several optimizations to enhance the performance of the proposed algorithms. Our experiments on several large benchmarks from industrial designs as well as the public domain demonstrate that in several instances the integration of dynamic abstraction and interleaved traversal is necessary to complete the invariant check. In other cases the use of interleaved traversal either provide a significant performance improvement or a modest overhead.

There are several avenues for enhancing the proposed algorithms. One of the possibilities would be to reduce the overhead that the interleaved traversal incurs. This could be done by refining the heuristic for choosing the direction of traversal or by use of the "2-DD manager" idea proposed in [18]. Another direction would be to integrate the proposed algorithms into a hybrid framework combining counter-example guided and proof based abstraction such as the one proposed in [1].

References

1. N. Amla and K. L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In A. J. Hu and A. K. Martin, editors, *Proc. of Formal Methods in CAD*, volume 3312 of *LNCS*, pages 260–274. Springer, Nov 2004.
2. P. Bjesse and J. Kukula. Using Counter Example Guided Abstraction Refinement to Find Complex Bugs. In *Proc. of the Design Automation and Test in Europe*, pages 156–161, February 2004.
3. G. Cabodi, P. Camurati, and S. Quer. Efficient State Space Pruning in Symbolic Backward Traversal. In *Proc. of the Intl. Conf. on Computer Design*, pages 230–235, October 1994.
4. G. Cabodi, S. Nocco, and S. Quer. Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification. In E. Brinksma and K. Larsen, editors, *Proc. of the Intl. Conf. on Computer Aided Verification*, volume 2404 of *LNCS*. Springer, July 2002.
5. P. Chauhan, E. M. 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 M. Aagaard and J. W. O'Leary, editors, *Proc. of the Intl. Conf. on Formal Methods in CAD*, volume 2517 of *LNCS*, pages 33–51, Nov. 2002.
6. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, July 2001. Kluwer Academic Publishers.
7. E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT-based Abstraction Refinement Using ILP and Machine Learning Techniques. In E. Brinksma and K. Larsen, editors, *Proc. of the Intl. Conf. on Computer Aided Verification*, volume 2404 of *LNCS*, pages 265–279. Springer, July 2002.

8. M. Glusman, G. Kamhi, S. M.-H., R. Fraer, and M. Y. Vardi. Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. In H. Garavel and J. Hatcliff, editors, *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 176–191. Springer, April 2003.
9. E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proc. of the Design Automation and Test in Europe*, pages 886–891, March 2003.
10. S. G. Govindaraju and D. L. Dill. Verification by approximate forward and backward reachability. In *Proc. of Intl. Conf. on CAD*, pages 366–370, Nov. 1998.
11. A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative Abstraction Using SAT-based BMC with Proof Analysis. In *Proc. of the Intl. Conf. on CAD*, pages 416–423, Nov. 2003.
12. H. Iwashita and T. Nakata. Forward model checking techniques oriented to buggy designs. In *Proc. of Intl. Conf. on CAD*, pages 400–404, Nov. 1997.
13. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1995.
14. B. Li and F. Somenzi. Efficient Computation of Small Abstraction Refinements. In *Proc. of the Intl. Conf. on CAD*, Nov. 2004.
15. F. Y.-C. Mang and P.-H. Ho. Abstraction Refinement by Controllability and Cooperativeness Analysis. In *Proc. of the Design Automation Conf.*, pages 224–229, June 2004.
16. K. L. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
17. K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Proc. of the Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 2–17. Springer, April 2003.
18. C. Stangier and T. Sidle. Invariant Checking Combining Forward and Backward Traversal. In A. J. Hu and A. K. Martin, editors, *Proc. of* 5th *Intl. Conf. on Formal Methods in CAD*, volume 3312 of *LNCS*, pages 414–429. Springer, Nov. 2004.
19. The VIS Group. VIS: A system for Verification and Synthesis. In R. Alur and T. Henzinger, editors, *Proc. of the Intl. Conf. on Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer, July 1996.
20. VIS Verilog Benchmarks. http://vlsi.colorado.edu/~vis.
21. C. Wang, G. D. Hachtel, and F. Somenzi. Fine-Grain Abstraction and Sequential Don't Cares for Large Scale Model Checking. In *Proc. of the Intl. Conf. on Computer Design*, October 2004.
22. C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement. In *Proc. of the Intl. Conf. on CAD*, pages 408–415, Nov. 2003.
23. D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal Property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines. In *Proc. of the Design Automation Conf.*, pages 35–40, June 2001.
24. http://ee.princeton.edu/~chaff/zchaff.php, Dec. 2003.
25. L. Zhang and S. Malik. Validating SAT Solvers using an Independent Resolution-based Checker: Practical Implementations and Other Applications. In *Proc. of the Design Automation and Test in Europe*, pages 880–885, March 2003.
26. L. Zhang, M. R. Prasad, M. Hsiao, and T. Sidle. Dynamic Abstraction Using SAT-based BMC. In *Proc. of the Design Automation Conf.*, pages 754–757, June 2005.