Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation

Marcelo Glusman[1,2], Gila Kamhi[2], Sela Mador-Haim[2], Ranan Fraer[2], and Moshe Y. Vardi[3⋆]

[1] Computer Science Department, The Technion, Haifa, Israel
marce@cs.technion.ac.il
[2] Formal Property Verification, Intel Corporation, Haifa, Israel
{gila.kamhi,sela.mador-haim,ranan.fraer}@intel.com
[3] Department of Computer Science, Rice University
vardi@cs.rice.edu

Abstract. In this paper, we describe a completely automated framework for iterative abstraction refinement that is fully integrated into a formal-verification environment. This environment consists of three basic software tools: Forecast, a BDD-based model checker, Thunder, a SAT-based bounded model checker, and MCE, a technology for multiple-counterexample analysis. In our framework, the initial abstraction is chosen relative to the property under verification. The abstraction is model checked by Forecast; in case of failure, a counterexample is returned. Our framework includes an abstract counterexample analyzer module that applies techniques for bounded model checking to check whether the abstract counterexample holds in the concrete model. If it does, it is extended to a concrete counterexample. This important capability is provided as a separate tool that also addresses one of the major problems of verification by manual abstraction. If the counterexample is spurious, we use a novel refinement heuristic based on MCE to guide the refinement. After the part of the abstract model to be refined is chosen, our refinement algorithm computes a new abstraction that includes as much logic as possible without adding too many new variables, therefore striking a balance between refining the abstraction and keeping its size manageable. We demonstrate the effectiveness of our framework on challenging Intel designs that were not amenable to BDD-based model-checking approaches.

1 Introduction

One of the most significant recent developments in the area of formal design verification is the discovery of algorithmic methods for verifying properties of *finite-state* systems. In temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired temporal property by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this property [9, 27, 33, 37]. With the advent of symbolic techniques [7], model-checking tools have enjoyed a substantial and growing use over the last few years, showing ability

⋆ Supported in part by NSF grants CCR-9700061, CCR-9988322, IIS-9908435, IIS-9978135, and EIA-0086264, by BSF grant 9800096, and by a grant from the Intel Corporation.

to discover subtle flaws that result from extremely improbable events. While until recently these tools were viewed as of academic interest only, they are now routinely used in industrial applications [5, 16]. Nevertheless, model checking is still limited with respect to the size of the designs it can handle, due to the so-called *state-explosion problem*, which refers to the exponential complexity of model checking with respect to the number of state variables in the model. One of the most fundamental techniques for dealing with the state explosion problem is that of *abstraction*, in which the concrete model is abstracted to a simpler model that has a smaller state space, and, hopefully, retains the essential features of the concrete model [11]. The abstract model is typically an overapproximation of the concrete model–it allows behaviors that are not allowed in the concrete model. Thus, with respect to universal properties (properties that are defined in terms of universal quantification over traces), model checking the abstract model is *sound*–if the abstract model satisfies the property, then so does the concrete model. Since the abstract model, however, is an overapproximation, "false negatives" are possible; one can get *spurious* counterexamples, which cannot be extended to allowed behaviors in the concrete model.

Consequently, without automation, model checking by abstraction requires a fair amount of manual labor. First, one has to decide how to abstract the concrete model. If the model checker shows that the abstract model satisfies the property under verification, then so does the concrete model and we are done. Otherwise, the model checker returns a counterexample with respect to the abstract model. One then has to analyze the counterexample to see if it is real or spurious. If the counterexample is real, then it has to be extended to a counterexample of the concrete model, which is then returned to the verification engineer for debugging. If the counterexample is spurious, one has to refine the abstract model in order to bring it closer to the concrete model, so that the spurious counterexample is eliminated. This labor-intensive process reduces dramatically the productivity of the verification engineer. As a result, the usefulness of model checking to the formal verification of large industrial designs is seriously hampered. Over the last decade, there has been a consistent effort to automate the above process of model checking by abstraction, offering algorithmic support to *iterative abstraction refinement*. This consists of three basic steps: *abstract* the design's model, *analyze* the counterexample, *refine* the abstraction (see Figure 1). Starting with Balarin and Sangiovanni-Vincentelli [2], researchers described several ways in which these steps can be automated [11, 25, 31, 28, 10, 30, 17, 38] (see Related Work).

In this paper, we describe a completely automated prototype framework for iterative abstraction refinement that is integrated into a formal-verification environment consisting of three basic software tools: Forecast, a BDD-based model checker [16], Thunder, a SAT-based bounded model checker [13], and MCE, a technology for multiple-counterexample analysis [14]. As in [25], to abstract the model, we automatically cut (prune) the design logic, by turning chosen circuit nodes (not necessarily latches) into free, i.e., unconstrained inputs, therefore pruning the logic that drives them; also, the initial abstraction is chosen relative to the property under verification. Our framework applies bounded model checking techniques [4] to analyze the counterexample. The key insight here is that the abstract counterexample has a bounded length, which enables us to reduce this problem to a bounded model checking problem. Clarke et al in [8] have

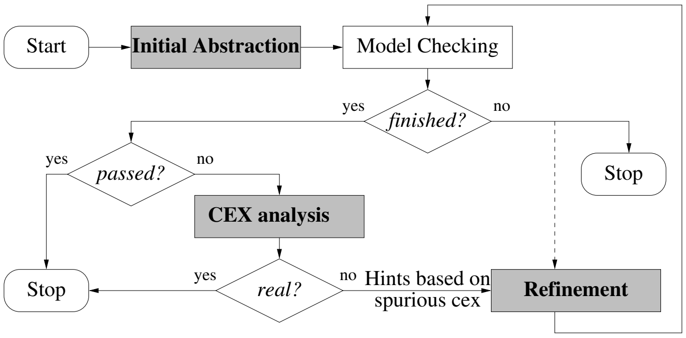

Fig. 1. Framework for automated abstraction and iterative counterexample-guided refinement

proposed similar SAT-based techniques to address this problem. In addition to being an integral part of the automatic refinement framework, this capability in our solution is also crucial to support manual abstraction methods, so it is also provided as a stand-alone utility. In summary, the task is to check whether an abstract counterexample holds in the concrete model. If it does, it has to be extended to a concrete counterexample, which the verification engineer can use for debugging.

If the analysis step indicates that the abstract counterexample is spurious, then the abstraction is too "loose" and it has to be refined. Some freed nodes have to be un-freed, adding new logic to the abstract model. This increases the model's logical complexity, which raises the computational complexity of model checking it. Thus, there is a trade-off here between eliminating spurious behaviors and increasing the size of the abstract model. One of our two main contributions is a novel heuristic to choose the nodes to be un-freed. A recent work shows how Multiple Counterexample (MCE) technology [14] supports the analysis of many counterexamples simultaneously. We show how an MCE-based heuristic can be used to guide the refinement process by selecting for un-freeing nodes that are more relevant for eliminating the spurious counterexamples.

Our second main contribution is to show how, after the nodes to be un-freed are chosen, we can refine the model by including as much logic as possible without adding too many new variables (freed nodes). Other iterative refinement frameworks [38] perform the refinement only at latch level. In many real-life designs, however, replacing a latch with all the new latches in its transitive fan-in can cause sudden addition of huge amounts of logic. Therefore it is desirable to be able to prune at the intermediate nodes. We describe a heuristic that analyzes the logic between the un-freed nodes and their fan-in latches to find a balance between refining the abstraction and keeping the abstract model's size manageable.

The rest of the paper is structured as follows: Section 2 describes related work in the published literature. Section 3 introduces the model checking environment in which our tool operates, and the way an initial abstraction is chosen. Sections 4 and 5 describe, respectively, the counterexample analysis module, and the refinement algorithm.

In Section 6 we evaluate the effectiveness of our prototype on a set of challenging Intel designs that were not amenable to BDD-based model-checking approaches.

We follow with concluding remarks in Section 7.

2 Related Work

The *cone-of-influence* (COI) *reduction* [12] is the most common abstraction technique, used today as a core component in most model-checking tools. The COI of a property consists of all the variables that affect the property directly or indirectly, based on the dependencies between the variables of the model. Variables that are not in the COI cannot influence the validity of the specification and can therefore be removed from the model. COI reduction is an *exact* abstraction, in the sense that the abstract model satisfies the property *if and only if* the concrete model does. More aggressive abstractions, which are the ones we consider in this paper, go one step further by eliminating parts of the COI that are believed to be irrelevant to the property being checked. In doing so, they create an overapproximation, as the abstract model might introduce spurious behaviors that were not present in the concrete one. A significant effort has been invested in automating the whole process, resulting in various *iterative refinement* frameworks [25, 2, 26, 31, 32, 28, 22, 10, 17, 38, 35]. We now discuss some of these works.

An early such framework is the *localization reduction* of Kurshan [25], defined in the context of ω-regular language containment, and implemented in COSPAN [19]. This reduction keeps the nodes (both latches and intermediate nodes) that are topologically close (in the node dependency graph) to the property being verified, while the other nodes are abstracted away with non-deterministic assignments. The refinement is counterexample guided, with each step adding additional nodes, again according to the dependency graph. Unfortunately, not enough details (for an effective implementation) are provided in [25] on the check for spurious counterexamples or the selection of a small set of nodes to eliminate such counterexamples.

Balarin and Sangiovanni-Vincentelli [2] present a similar iterative framework for checking language emptiness of communicating automata. To check for spurious counterexamples they synthesize an automaton from the error trace and intersect it with the automaton of the concrete model. The resulting language emptiness problem is submitted to a BDD-based model checker. Our experience shows that BDD-based methods have little chance of coping with the size of the concrete model. By contrast, SAT-based solutions like ours for counterexample analysis scale much better with the model size. Moreover, we avoid compiling the error trace into an automaton, to avoid introducing auxiliary variables. Instead, our reduction to bounded model checking translates the error trace into new constraints added to the SAT instance.

Clarke et al. [10] use counterexample-guided refinement for ACTL^* model checking. Their abstraction exploits the control structure of a HDL (Hardware Description Language) design rather than the dependency graph. This provides finer control by allowing several degrees of abstraction for each variable. Relying on such syntactic information, however, hampers the application of this technique to gate-level designs. To check for spurious counterexamples, BDD-based reachability analysis is performed on the concrete model constrained with the information provided by the error trace. This suffers from the same scalability problem mentioned above. This last issue is addressed in [3], where the analysis and reconstruction of counterexamples is performed on an intermediate model that is midway between the concrete model and the abstract one. The intermediate model is itself refined in an incremental way. As the other BDD-based approaches, this approach still has limited capacity as the examples reported in the paper do not exceed 400 variables. Today it is widely accepted that SAT or ATPG are more appropriate for the analysis of spurious counterexamples. That is the approach taken in [8] where SAT is used for this task. This paper suggests also two new refinement heuristics, one based on Integer Linear Programming and the other based on Machine Learning. Both heuristics try to find a minimal set of variables that separates between a set of dead states and a set of bad states. To that purpose, one needs to sample sufficiently many states in both sets. In the worst case we might have to sample an exponential number of states, so the sampling itself can be quite expensive.

The iterative refinement tool described in [38] was the first one to employ different verification engines. A hybrid BDD-ATPG engine is used for model checking, sequential ATPG is used for detecting spurious counterexamples, and a combination of 3-valued simulation and sequential ATPG is used to obtain refinement hints. We strongly believe that using additional engines in addition to BDD-based ones is, indeed, the only way to overcome the limitations of BDD-based model checking.

A distinguishing feature of our approach is the fine control over successive refinements' size growth. This stems from two critical aspects: First, our refinement does not attempt to eliminate one single error trace at all price. Our experience shows that this tends to push the refinement in the wrong direction by adding more logic than necessary. Instead, our refinement analyzes simultaneously all the error traces (of a given length) and adds a small number of nodes that are likely to be the root cause of all such spurious counterexamples. Second, we do not limit our pruning to the level of latches, but we also prune at the level of intermediate nodes in the design. This strategy was found to be critical in coping with nodes that have a large fan-in cone by avoiding to bring in the whole cone of these nodes at once. By carefully choosing the nodes to be pruned, we get fine control over the growth of the successive refinements' size. The Min-Cut technique we apply to achieve this effect and the flow problem we generate from the circuit when using this technique are quite standard, but their use for the specific purpose of choosing the next pruning of the model is new. In [38], a Min-Cut is computed in the counterexample analysis phase, but as we said, in that work only latches are chosen for pruning. A similar technique is also used in the Ketchum tool [21], when checking for unreachability of coverage states, as an optimization of automatic test-pattern generation. It is known (even though no details have been published) that in the FormalCheck tool a Min-Cut is used for refinement, but to our best knowledge, it is computed on a completely different flow problem.

3 Model Checking Environment and Initial Abstraction

The set-up: The hardware design being verified is given in a high-level hardware description language and compiled into a logical model, on which the model abstraction and refinement process takes place. The logical models on which we operate consist of a set V of Boolean variables used to represent the system states, a Boolean formula I representing the set of initial states, a collection *TR* of Boolean formulas representing transition constraints, and a collection F of formulas representing fairness constraints. All the formulas are over the variables in V, except for the formulas in *TR*, which are over the variables in $V \cup V'$, where $V' = \{x' \mid x \in V\}$. Given an assignment s to the variables in V, let s' denote the corresponding assignment to the variables in V'. A *fair trace* of this model is an infinite sequence $s_0, s_1, \ldots$ of assignments to the variables in V, such that s_0 satisfies I, for every $i \in \mathbb{N}$ and for all $\tau \in \mathit{TR}$ $s_i \cup s'_{i+1}$ satisfies τ, and for every $\varphi \in F$ there are infinitely many i such that s_i satisfies φ [29].

In our set-up, the specification and assumptions are given in Intel's ForSpec language [1], which is linear-time temporal logic augmented with regular expressions, clocks, and resets. Given a temporal assumption f and a temporal assertion g, the requirement we must check is that the model satisfies $f \rightarrow g$. This check is implemented via the automata-theoretic approach [36]. Both f and $\neg g$ are compiled into logical models of their respective Büchi automata. These models are conjoined with the logical model of the design under verification, and the model checker then tests the combined model for language emptiness.

The abstraction method we chose consists of selecting certain nodes in this combined model, and turning them into free inputs. The constraints in *TR* are typically of the form "$v' = \ldots$" or "$v = \ldots$", so we can turn v into a free input by removing from *TR* any constraints with v or v' on the left side. We call this operation *freeing v*. After it is done, part (or all) of the logic that drives v stops influencing the variables that define the property, thus allowing the COI reduction to prune that part of the logic from the model. The freed nodes therefore define a *frontier* that separates, in the concrete model, the logic included in the current abstract model from the logic that is pruned out.

Initial Abstraction: The first step is automatic generation of an initial abstract model, i.e., the first frontier. The specification and assumption's logic are probably very relevant to the property being checked, so pruning them is likely to cause spurious counterexamples. Moreover, the part of the model generated by the ForSpec compiler is relatively small. Our initial abstract model is thus obtained by freeing the first nodes of the original design's logic model that are connected to the ForSpec-generated logic (See Figure 2). Model checking this first abstract model usually takes very little time

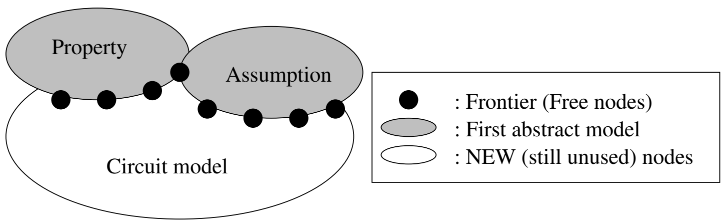

Fig. 2. Initial Abstraction

and finds only spurious counterexamples, unless the property is directly implied by the provided assumption, regardless of the design being verified.

Model Checking: Forecast, Intel's BDD-based model checker, searches for a counterexample trace, which is represented by a prefix and a fair cycle (i.e., one in which every fairness constraint is satisfied at least in one state). In this paper we focus on safety properties, for which only a prefix of a trace is needed [24]. In case an abstract counterexample is found, Forecast provides a trace that must be analyzed to determine if it is real (see Section 4). Forecast also provides Multiple-Counterexample (MCE) information [14], to be used by the refinement module (See Section 5.2).

We should note that Forecast is run using all the usual reductions and optimizations, such as the Cone of Influence reduction [12] and Dynamic Variable Reordering [34], which means that the order of the logic variables in the BDDs evolves during the model checking session. Every run's final variable order is saved in a file and later incorporated by the model checker in the next iteration, as part of its initial variable ordering. Every iteration's abstract model includes all the variables that appear in the preceding ones, so by incorporating the previously computed variable order we can obtain smaller BDDs.

4 Counterexample Analysis

Automatic abstractions as well as manual abstractions can result in false negatives. Therefore, the efficiency of any framework that supports abstractions highly depends on the ability to detect whether the abstract counterexamples resulting from a verification session are spurious or not. In this section, we describe a counterexample analysis method (hereafter called "CexAn") for determining whether an abstract counterexample is real, i.e., can be extended to the concrete model. If this is the case, CexAn extends the abstract counterexample to a concrete one. Hence, the benefits of CexAn are both in determining spurious failures and, in case of valid failure reports, easing the debugging of the abstract counterexamples since the provided concrete counterexamples include assignments for the concrete model's inputs.

CexAn is built on SAT-based bounded model checking technology. The inputs to CexAn are the concrete model M, and the abstract counterexample *AbstCex*. The counterexample analysis problem is translated to a k-bounded model checking problem, where $k + 1$ is the length of AbstCex. As in bounded model checking, the concrete model is unrolled k steps and a propositional formula is generated. The formula describes paths from s_0 to s_k such that s_0 is an initial state and for all $0 \leq i < k$, there is a transition from s_i to s_{i+1}: $Path(s_0, \ldots, s_k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} TR(s_i, s_{i+1})$
The construction of $Path(s_0, \ldots, s_k)$, while straightforward in principle, is a nontrivial computational task because of the size of the concrete model (the formula may have hundreds of thousands propositional variables!). Our bounded model checking tool, Thunder, implements the unrolling very efficiently. The AbstCex is also translated to a propositional formula *CexForm*, in the following way: Let $Phase_i$ be a Boolean formula describing the i-th phase ($0 \leq i \leq k$) in AbstCex. Every $Phase_i$ restricts the variables corresponding to abstract model nodes, to the values they take in the corresponding phase of CexForm. Then, we define: $CexForm(s_0, \ldots, s_k) = \bigwedge_{i=0}^{k} Phase_i$

The SAT engine looks for a satisfying assignment for the formula:

$$Path(s_0, \ldots, s_k) \wedge CexForm(s_0, \ldots, s_k).$$

In case a satisfying assignment is found, we report that the counterexample is real and provide the extended counterexample in terms of the concrete model's signals. If a satisfying assignment does not exist, we conclude that the concrete model does not display a counterexample agreeing with AbstCex on the signals present in the abstract model, so we report that the counterexample is spurious.

5 Refinement Algorithm

5.1 Rationale

For abstractions by freeing and pruning, refining an abstraction means moving the frontier of freed nodes, therefore defining a bigger abstract model. This involves two main steps: i) choosing the nodes in the frontier that will be "un-freed", therefore adding their whole "cone of influence" back into the model, and ii) choosing the nodes in the newly added logic that will be freed, to keep the abstract model small. In the new abstraction, the un-freed nodes' values are constrained by the added logic, so fewer behaviors will be possible. The refinement iterations terminate since we do not apply backtracking.

The goal of an iterative refinement process is to find an abstraction that does not display *any* spurious counterexamples. It may have no counterexamples at all, or it may reveal a real one. To achieve this goal in fewer iterations, every refinement step should add as much logic as possible. However, our search for this goal is carried out within the capacity limits of the model checking phase, so we must only add the logic that has the best chance to (eventually) eliminate *all* the spurious counterexamples.

Given a spurious counterexample, it is computationally difficult to find a minimal set of nodes that need to be un-freed to eliminate it [10]. One approach is to greedily add nodes to the abstract model until the spurious counterexample at hand is eliminated. This greedy approach may be quite suboptimal and lead to significant growth of the abstract model. Another approach is to overapproximate and add a large "chunk" of logic in one refinement step to guarantee elimination of the spurious counterexample. This may cause the sudden addition of too much logic to the abstract model. Thus, eliminating the spurious counterexample at hand in a single refinement step is not necessarily the best way of refining the abstraction.

The next section presents a new heuristic that, at every iteration, guides the refinement by hinting at nodes in the frontier that should be un-freed to eliminate all or many of the spurious counterexamples of a given length. For the reasons just described, we prefer to limit the amount of logic added in a single iteration, and decide on the next refinement steps based on fresh hints from the guiding heuristic. The tradeoff between adding more logic in a single iteration and limiting the abstract model's size is present throughout the rest of this section.

5.2 Choosing the nodes in the frontier to be un-freed

In counterexample-guided refinement frameworks, spurious counterexamples are analyzed, looking for hints as to which elements of the abstraction cause the spurious counterexamples. In our case, the hints we need should suggest which nodes in the frontier must be un-freed first. Our tool's refinement module implements a new technique for obtaining such hints. The new heuristic exploits Multiple Counterexample (MCE) information [14] provided as a multi-valued counterexample annotation by our BDD-based symbolic model checker, Forecast. A multi-valued annotation of a counterexample represents *all* the counterexamples of the same length. The use of MCE information facilitates the simultaneous elimination of several spurious counterexamples in a few refinement steps. This makes the new heuristic specially suited for attaining the goal defined in Section 5.1.

Multi-valued counterexample annotation Traditional symbolic model checkers provide a single counterexample as the output of a failing verification. It is specially difficult to diagnose a verification failure reported as a single counterexample trace. On one hand, the verification engineer has too much data, all the signal's values along the whole counterexample trace. On the other hand, he has too little data, only one counterexample among many possible ones. Forecast addresses the counterexample diagnosis problem by providing MCE information in the form of "multi-valued counterexample annotation", a concise and intuitive counterexample data representation. In this annotation, a counterexample trace is enhanced with a classification of signal values along the trace into three types: (a) *Strong* 0/1: indicates that in all counterexamples, the value of the signal at the given phase of the trace is 0 or 1, respectively. (b) *Conditional* 0/1: indicates that although the value of the signal at this phase is 0 or 1 for this counterexample, this value can be different for another counterexample illustrating the failure. (c) *Irrelevant* 0/1: indicates that the value of the signal at this phase is probably unrelated to the verification failure. Thus a single multi-valued annotated counterexample also provides information on all the other possible counterexamples of the same length.

Using MCE to obtain hints for refining abstractions The strong values provide insight on the pertinent signals causing the counterexample. For example, if the value of a signal at a certain phase of a counterexample is a strong zero, this means that correcting the design so that the value of the signal will be one at that phase often gets rid of all counterexamples of the same length as the counterexample at hand. Hence, the error rectification problem is often reduced to determining how to cause a strong-valued signal to take on a different value. We first assign a unique fixed weight to each of the three value types, where strong values get a higher weight. Then, given a multi-valued annotated spurious counterexample, we compute, for every node, the average weight of the values assigned to it along the whole trace. The nodes with the highest average weight are chosen for refinement.

The decision on how many nodes in the current frontier should be un-freed at every iteration involves the tradeoff described in Section 5.1. Choosing only the node with the highest average weight yields the most cautious refinement process, but a longer one. Experimentation, and examination of the list of frontier nodes sorted by decreasing weight, suggest that good results can be obtained by un-freeing only a few nodes at a time. In Section 6, we elaborate on experimental results that lead to a decision on the number of nodes to un-free at each iteration.

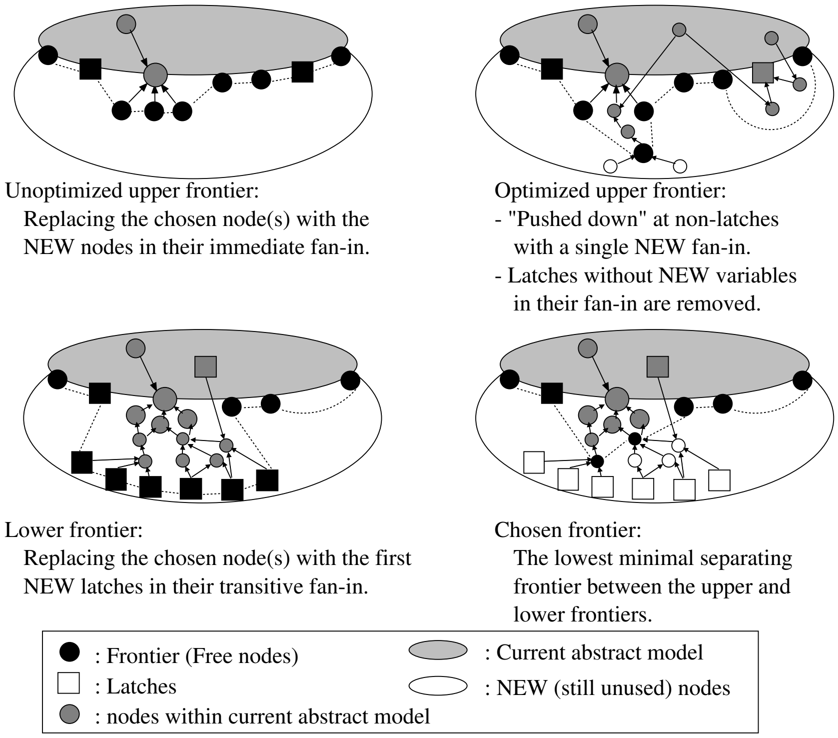

Fig. 3. Choosing a new frontier

5.3 Choosing the new frontier

In some iterative abstraction refinement frameworks in which the abstractions consist of pruning the model (e.g., [38]), only latches (i.e., registers) are chosen to become part of the frontier. After choosing the latches to be un-freed, the new latches that replace them in the frontier are those in their transitive fan-in (except for those that already appear within the current abstract model). Our experiments on real-life examples showed that this refinement policy causes, in many cases, a sudden increase in the abstract model's size, bigger than what we would like to have in a controlled refinement process – usually causing lots of irrelevant logic to be taken into the current abstract model, and therefore slowing down the model checking phase or even reaching its capacity limit. To keep the growth rate of the number of variables under control, we realized that it is crucial to be able to add any node to the free nodes frontier, without restricting ourselves to freeing only latches. The most cautious approach is to replace a node in the frontier with those nodes in its *immediate* fan-in (only the new ones, i.e., those which do not appear in the previous model). This yields a very slow refinement process. In the sequel, we refer to this frontier as the *upper frontier*, and we call the frontier that passes through the latches and input signals in the transitive fan-in – *lower frontier* (see Figure 3). Obviously, a tradeoff (based on the discussion in Section 5.1) has to be found between the upper frontier and the lower frontier. It is desirable to allow the model to grow as much as possible, while minimizing the number of variables in the BDDs, since this typically minimizes the size of the BDDs describing sets of reachable states. However, the addition of intermediate nodes increases the number of constraints, which may in turn increase the time cost of computing image sets. Between the upper and lower frontiers there are no latches, so the number of variables added depends on the number of nodes in the chosen frontier itself (because they are turned into free inputs). This means that we must seek the smallest frontier, and among those with minimal size, the one that has the most constraints on the transition relation.

First, we present some straightforward optimizations that can be applied to the upper frontier, so that it is pushed "downwards" without adding variables (and sometimes saving some). These optimizations (illustrated in Figure 3) are performed repeatedly on the upper frontier, until they are no longer applicable:

- Single-new-fan-in propagation: At every given moment, the nodes in the fan-in of a node in the frontier can be divided into two sets: those that already appear in the current abstract model, and those that do not (we call them *new* nodes). If a node n_1 in the frontier is not a latch, and it has a *single* new fan-in node n_2, then n_1 can be replaced in the frontier by n_2 without adding variables to the model.
- Latches without new fan-in variables: If a node in the frontier is a latch, and it does not have any new variables (i.e., latches or primary inputs that are not part of the current abstract model) in its transitive fan-in, then it can be un-freed (i.e., removed from the frontier). This adds to the abstract model all the new logic that drives the latch, without adding any variables to the model. This optimization saves many refinement iterations that would otherwise be spent un-freeing this kind of latches.

The frontier we are looking for must *separate* the upper and lower frontiers, in the sense that every path (following the fan-in relation) connecting a node in the lower frontier to a node in the upper one must include at least one node belonging to the frontier. This means that we are looking for a minimal cut in the part of the model between the optimized upper frontier and the lower frontier. The computation of such a minimal cut is a standard procedure in circuit analysis [23, 18], implemented by first translating the part of the model between the frontiers into a simple flow problem based on the fan-in relation, then executing a Maxflow algorithm [15], and finally applying the Max-Flow Min-Cut Theorem [6] to derive a minimal cut from the maximal flow found. If several such minimal cuts exist, we prefer the lowest one, i.e., the one that takes as much intermediate logic as possible into the next abstract model.

6 Results

We now report how our iterative refinement prototype performs in experiments carried out on Intel designs. Some are verification test cases, i.e., the model checker answers "true", and some are falsification test cases, where a real counterexample is reported. All the selected test cases are way beyond the capacity of Forecast, our BDD-based

	Iterations	Variables in concrete vs. last abstract model	Forecast CPU time (secs) Last iteration	Total
Real1	1	627/132	215	215
Real2	10	627/167	1390	8008
Real3	10	627/161	112	1653
Real4	5	627/149	251	914
Real5	5	627/148	310	915
Real6	5	627/155	240	809
Real7	5	635/157	291	975
Real8	18	627/244	2983	9558
Real9	77	903/192	2100	19800
Real10	8	669/218	22	227

Table 1. Results on ten real-life test cases beyond the capacity of BDD-based symbolic model checking. *Real1*-*Real9* are verification cases, *Real10* is a falsification case.

model checker (they did not complete within the timeout bound of 48 hours, even with COI reduction).

Table 1 records, for each test case, the number of iterations required to reach a definite result (pass or fail), the number of variables (latches and inputs) in the concrete model and in the abstract model of the last iteration (both after the COI reduction), Forecast CPU time for the last iteration, and total Forecast CPU time.

A significant problem in industrial-size projects is ensuring that the properties that are satisfied by a snapshot of the design under development, are also satisfied by later versions. In the context of conventional testing this is checked through regression testing. By saving the set of nodes that belong to the frontier defining the last abstraction (as well as the generated BDD variable order) we provide support for regression verification [20], which attempts to repeat a verification result after the design has been modified (e.g., if the changes affect only parts of the model that are pruned out after the nodes in the frontier are freed). For regression verification purposes, one would expect the running time to be quite close to Forecast CPU time on the last iteration. As is shown in Table 1, Forecast CPU times on the last iterations are negligible (less than 3000 seconds) in comparison to the original timeout bound (48 hours).

We also applied our prototype to five other designs: *Real11*, *Real2'*, *Real3'*, *Real4'*, *Real8'*. The results, reported in Table 2, demonstrate three points:

Robustness: The latter four designs are later versions of designs *Real2*, *Real3*, *Real4*, *Real8* from Table 1. As can be seen by the reported number of variables in the concrete models, these designs are larger than the earlier versions by about 50 variables. Such growth can pose a significant hurdle to a BDD-based model checker. In contrast, this growth did not have a marked impact on our prototype.

Regression: As discussed above, for regression verification a good starting point is the abstraction and variable order obtained from the last refinement iteration. To see the impact of variable order reuse, we ran the last iteration of the refinement both with the order obtained from prior iterations and without that order (in that case Forecast supplied an initial order based on static analysis). As can be seen in designs *Real11* and

	Un-freed nodes	Iterations	Variables (concrete vs. last abstract model)	Forecast time (secs) Last iteration		Iterations that removed the CEX
				reused order	no reuse	
Real11	4	26	652/302	1902	10229	8/26
	16	7	652/221	2276		5/7
	24	6	652/331	1714		4/6
Real2'	4	10	677/222	1529	1757	7/10
	16	5	677/218	1381		3/5
	24	5	677/218	1492		3/5
Real3'	4	10	652/187	430	426	5/10
	16	6	652/221	1250		5/6
	24	6	652/223	2762		4/6
Real4'	4	6	677/209	656	814	3/6
	16	4	677/211	265		3/4
	24	4	677/256	256		3/4
Real8'	4	17	677/290	1307	3692	5/17
	16	7	677/293	6053		5/7
	24	6	677/293	timeout(3 hrs.)		4/6

Table 2. Results on five real-life verification test cases that were beyond the capacity of Forecast, when the number of nodes un-freed at every iteration was 4, 16 or 24. The Forecast time is also shown when the last iteration is rerun without reusing initial variable order information.

Real8', order reuse can have a dramatic impact on performance.

MCE hints: To assess the power of MCE hints, we ran the prototype with different settings, un-freeing at each iteration 4, 16, or 24 variables. As is seen in the table, going from 4 to 16 reduced the number of iterations (and consequently also total running time) significantly, but going from 16 to 24 did not have such an effect. We also checked for each iteration whether the spurious counterexample has been eliminated and reported on the fraction of iterations in which this happened. Again, as is seen in the table, a significant effect occurred only in the transition from 4 to 16 un-freed variables. Note that this transition typically did not increase the number of variables in the final abstraction. Our experience shows that for each design there is a balance point between counterexample elimination and abstraction growth rate. This point depends only on the design and not on the property, since the design is the major contributor to state explosion.

We applied our iterative refinement flow on 12 distinct real-life test cases, comparing the results of using the lower frontier vs. the min-cut frontier,. The results, reported in Table 3, clearly demonstrate three advantages of the Min-Cut optimization:

- The run-time of the iterative refinement flow can be significantly reduced. In all but 3 tests, we achieved reduction in run time. In one case, Test2, the run did not complete at all without applying the Min-Cut optimization, timing-out after several hours. Applying Min-Cut, the verification completed in 1171 seconds.
- When Min-Cut optimizations are not used, un-freeing more than 2 latches at a time adds too much redundant logic and have a negative impact on the quality of the abstraction as well as run time, whereas when Min-Cut is applied the frontier can

	No-Mincut				Mincut			
	Iterations	Latches	Inputs	time	Iterations	Latches	Inputs	time
Test1	1	0	140	111	1	0	126	48
Test2	t/o			16571	9	38	154	1171
Test3	19	36	180	5003	9	40	151	2241
Test4	18	34	147	4185	8	30	129	897
Test5	14	26	142	2391	6	25	130	1305
Test6	22	42	286	4543	8	36	151	653
Test7	15	28	152	1549	5	27	121	555
Test8	1	0	141	115	1	0	138	62
Test9	1	0	141	116	6	19	136	528
Test10	1	0	141	117	1	0	138	60
Test11	1	0	146	69	1	0	139	86
Test12	1	0	141	111	6	18	138	1246

Table 3. Comparison between refining to the lower frontier vs. to the mincut frontier

pass through non-latch elements so the addition of logic for every un-freed frontier node can be more conservative. We were able to un-free more frontier nodes (8) at each iteration. As a result, the number of iterations is reduced, as seen in the table.
- The Min-Cut technique may not reduce significantly the number of latches in the final abstractions, but it reduced the number of inputs significantly, as seen in Test6.

7 Concluding Remarks

Automatic generation of abstractions and their iterative refinement is a successful method that expands the applicability of current model-checking tools. Our work contributes to this area in several ways: First, we describe an automatic counterexample analysis tool, which is integrated with a SAT-based bounded model checker. Besides detecting spurious abstract counterexamples for the iterative refinement framework, this tool can be used in isolation for analyzing the results of manual abstractions. Most useful is its ability to provide a concrete version for real counterexamples. Second, our work departs from the way counterexamples have been used to guide the refinement process until now, in that the refinement step need not necessarily eliminate the spurious counterexample at hand. Instead, the refinement is done so that *all* the spurious counterexamples are eventually eliminated. While doing it, one must choose only the most relevant logic in order to avoid reaching the capacity limits of the model checker before a successful abstraction is found. We present a novel heuristic based on Multiple Counterexample (MCE) technology, that specifically addresses this goal. The MCE heuristic hints at the nodes in the frontier that are most relevant for all the counterexamples of the same length as the one at hand. Third, our refinement algorithm achieves fine control over the abstract models' size growth by allowing the frontier to include intermediate logic nodes, and by applying optimizations aimed at minimizing the number of added variables while maximizing the constraints added to the abstract model.

Our experiments show the success of this approach for real-life test cases that were beyond the capacity of a state-of-the-art BDD-based model checker, and provide insight on how various tool configuration parameters can be tuned for a given set of test cases.

One possible avenue for future work is to combine hints from different sources when deciding which nodes will be un-freed. Other possible direction for development is to allow the refinement process to backtrack and try other refinement directions when the capacity limits of the model checker are reached.

References

1. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, LNCS. Springer-Verlag, 2002.
2. F. Balarin and A.L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *CAV'93*, LNCS, pages 29–40. Springer-Verlag, 1993.
3. S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In E. Brinksma and K. G. Larsen, editors, *CAV'02*, volume 2404 of *LNCS*, pages 65–77. Springer-Verlag, 2002.
4. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.
5. P. Biesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessors using satisfiability solvers. In *CAV'01*, volume 2102 of *LNCS*, pages 454–464. Springer-Verlag, 2001.
6. B. Bollobas. *Graph Theory*. Springer-Verlag, 1979.
7. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
8. 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. G. Larsen, editors, *CAV'02*, volume 2404 of *LNCS*, pages 265–279. Springer-Verlag, 2002.
9. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.
10. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, 2000.
11. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
12. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
13. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV'01*, volume 2102 of *LNCS*, pages 436–453. Springer-Verlag, 2001.
14. F. Copty, A. Irron, O. Weissberg, N.P. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In T. Margaria et. al., editor, *Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *LNCS*, pages 275–292. Springer-Verlag, 2001.
15. L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

16. R. Fraer, G. Kamhi, B. Ziv, M. Vardi, and L. Fix. Prioritized traversal: efficient reachability analysis for verication and falsification. In *CAV'00*, volume 1855 of *LNCS*, pages 389–402. Springer-Verlag, 2000.
17. G.S. Govindaraju and D.L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *ICCAD'00*, 2000.
18. G. Hachtel and F. Somenzi. *Synthesis and Verification Algorithms*. Kluwer, 1996.
19. R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV'96*, volume 1102 of *LNCS*, pages 423–427. Springer-Verlag, 1996.
20. R.H. Hardin, R.P. Kurshan, K.L. McMillan, J.A. Reeds, and N.J.A. Sloane. Efficient regression verification. *IEE Proc. WODES'96*, pages 147–150, 1996.
21. P.H. Ho, T. Shiple, K. Harer, J.H. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal simulation engines. In *ICCAD'00*, pages 120–126, 2000.
22. J. Jang, I. Moon, and G. Hachtel. Iterative abstraction-based CTL model checking, 2000.
23. B. Krishnamurthy. An improved Min-Cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.
24. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19:291–314, 2001.
25. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.
26. W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *ICCAD'96*, pages 76–81, 1996.
27. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, pages 97–107, 1985.
28. J. Lind-Nielsen and H.R. Andersen. Stepwise CTL model checking of state/event systems. In *CAV'99*, volume 1633 of *LNCS*, pages 316–327. Springer-Verlag, 1999.
29. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, January 1992.
30. K.S. Namjoshi and R.P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV'00*, volume 1855 of *LNCS*, pages 435–449. Springer-Verlag, 2000.
31. A. Pardo and G. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *CAV'97*, volume 1254 of *LNCS*, pages 12–23. Springer-Verlag, 1997.
32. A. Pardo and G. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.
33. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1981.
34. R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *ICCAD'93*, pages 42–47. IEEE Computer Society Press, 1993.
35. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In W.R. Cleaveland, editor, *TACAS'99*, volume 1579 of *LNCS*, pages 178–192. Springer-Verlag, 1999.
36. M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.
37. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.
38. D. Wang, P.H. Ho, J. Long, J.H. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Design Automation Conference*, pages 35–40, 2001.