Fine-Grain Abstraction and Sequential Don't Cares for Large Scale Model Checking [*]

Chao Wang   Gary D. Hachtel   Fabio Somenzi

Department of Electrical and Computer Engineering
University of Colorado at Boulder, CO 80309-0425
E-mail: {Chao.Wang,Hachtel,Fabio}@Colorado.EDU

Abstract

Abstraction refinement is a key technique for applying model checking to the verification of real-world digital systems. In previous work, the abstraction granularity is often limited at the state variable level, which is too coarse for verifying industrial-scale designs. In this paper, we propose a finer grain abstraction in which intermediate variables are selectively inserted to partition large combinational logic cones into smaller pieces; these intermediate variables, together with the state variables, are then treated as "atoms" in abstraction refinement. With this fine-grain approach, refinement is conducted in two different directions, sequential and Boolean. We propose a SAT-based method for predicting the appropriate refinement direction, and apply greedy minimization in both directions to keep the refinement set small. We also explore the use of approximate reachable states of the remaining submodules to help verifying the abstract model. Experimental studies show that the proposed techniques significantly improve the performance of abstraction refinement, and therefore increase the model checker's ability to handle large designs.

1. Introduction

Model checking is a formal method for proving that a model satisfies a given specification under all possible input conditions. The model of a digital system must have a finite number of states; the user-defined specification, or property, is often expressed in temporal logics. The major challenge in applying model checking to large systems remain the *state explosion* problem—the number of states of the model is exponential in the number of its concurrent subcomponents. Because of this, state-of-the-art model checkers cannot directly handle most industrial-scale designs.

∗ This work was supported by SRC contract 2003-TJ-920.

Abstraction is a key to bridge this capacity gap, especially for verifying real-world properties that have a certain degree of locality. The idea is to use a simplified model to help verifying the property in the original model. Simplification is often achieved by pruning irrelevant details as much as possible. Abstraction refinement is an iterative process for the search of such a simplified model—one with which the given property can be either proven or refuted. In this process, one starts with a very small abstract model, in which the given property is checked. If the property passes, it is guaranteed to pass also in the original model; in this case, the model checking problem is solved. However, if the property fails, the counter-examples generated in the abstract model must be checked against the original model, to see if they contain real paths. If real counter-examples exist, the property is refuted. Otherwise, more details of the system are added to refine the current abstract model, after which the property is checked again. This process continues until either the property is decided, or the available computing resources (CPU time and memory) are exhausted.

For a given pair of model M and property ψ, there exists a minimum size abstract model M_{opt} with which the property can be decided. (When the property fails, the length of the shortest counter-examples in M_{opt} must match the length of the shortest counter-examples in M.) The smaller the size of M_{opt}, the higher is the degree of locality of the given property. However, finding the optimal abstraction is hard, and existing methods all use various heuristics in the hope of coming close to such an optimal abstract model. We define the *abstraction efficiency* in the following, as a way of evaluating the performance of different abstraction refinement methods:

$$\eta_{(M,\phi)} = 1 - \frac{final\ abstract\ model\ size}{concrete\ model\ size} \quad .$$

Each verification problem has a maximum abstraction efficiency; problems with higher η_{opt} are more suitable for abstraction refinement. Furthermore, how close to this optimal value a certain abstraction refinement algorithm can get is a good indicator of how good the algorithm is.

In many existing methods [4, 5, 10, 16, 8], the model size is given in terms of the number state variables (or latches). This corresponds to a "coarse-grain" abstraction approach, in which each state variable is treated as an *atom*—its entire fan-in combinational logic cone is either included in or completely excluded from the abstract model. However, not every one of its fan-in logic gates might be necessary for the verification, even if the state variable itself is indeed necessary. Including these redundant gates may significantly increase the complexity of the abstract model. This is especially true for industrial-scale designs that have huge combinational logic cones—an abstract model with few state variables may contain a large number of logic gates. As a result, the BDD-based image and pre-image computations in the abstract model become too expensive to perform.

In this paper, we propose a finer grain abstraction refinement approach for the verification of safety properties. Intermediate variables are selectively inserted in large combinational logic cones to partition them into smaller pieces. In the abstraction as well as the successive refinement steps, these immediate variables, or *Boolean network variables (BNVs)*, are treated as state variables—both are considered atoms. By controlling the size of the partition clusters, one can fine-tune the granularity of the abstraction. With the fine-grain abstraction, refinement can be conducted in two different directions: the *sequential* direction, in which more state variables are added to the current abstract model, and the *Boolean* direction, in which more Boolean network variables are added. We show that the control of refinement directions is important to achieve a better performance, and propose a SAT-based method for predicting the appropriate direction. Greedy minimization is also applied in both directions to remove the possible redundant variables from the refinement set.

We also explore the use of approximate reachable states of the remaining submodules (those that are abstracted away) to help the verification. In previous works, the remaining submodules were discarded; to the best of our knowledge, we are the first to analyze these remaining submodules in abstraction refinement. Approximate reachable states are used to constrain the behavior of the abstract model, by disabling certain valuations of its pseudo-primary inputs. With this approach, abstraction refinement becomes a multi-way circuit partition refinement process, in which all parts are considered during verification.

After surveying related work in Section 2 and establishing notation in Section 3, we present our fine-grain abstraction approach in Section 4. In Section 5, we explain the SAT-based method for predicting the refinement direction, and the application of greedy minimization in both directions; we also briefly review the algorithm we have adopted in picking refinement variables. The use of sequential don't cares is illustrated in Section 6, followed by the experimental results in Section 7. We conclude in Section 8.

2. Related work

Abstraction refinement was first introduced by Kurshan [9] in verifying linear time properties. Since then, significant progress has been made on the concretization test and the refinement algorithms [4, 5, 2, 10, 16, 8]. However, in these methods, the abstraction granularity remains at the state variable level. In [17], Wang *et al.* propose to use min-cut to further reduce the size of the abstract model. They compute a minimum size cut-set of signals between the *free-cut* signals and the combinational inputs, and then include only logic gates above this cut in the reduced *min-cut model*. The transition relation may become simpler when expressed in terms of these min-cut signals; however, the abstraction granularity is still at the state variable level. In particular, logic gates above the *free-cut* are always included in the abstract model, even though they sometimes may not be necessary for verification. A similar approach was also proposed by Chauhan *et al.* in [2], where further reduction of the abstract model is achieved by *pre-quantifying* the pseudo-primary inputs dynamically.

In [7], Glusman *et al.* pointed out that restricting the abstraction granularity at the state variable level might be too coarse. They compute a min-cut set of signals between the boundary of the current abstract model and the combinational inputs, and include logic gates above this cut in the refined model. Since an arbitrary subset of the fan-in combinational logic gates of a state variable can be added to the abstraction, the abstraction granularity is at the gate level. However, there are some significant differences between our approach and theirs: (1) we treat each elementary transition relation cluster as an abstraction atom and our goal is to reduce the BDD size of the abstract transition relation, while they aimed at reducing the number of cut-set variables (or pseudo primary inputs) in the refined model; (2) we use a SAT-based method for predicting the appropriate refinement direction, while they did not differentiate between the sequential and Boolean directions during the refinement; (3) in their approach, logic gates cannot be removed once they are added, while in our approach, they may be removed from the abstract model if later they are proven to be irrelevant during the greedy minimization.

In [11], McMillan and Amla proposed an automatic abstraction technique based on the unsatisfiability proof of bounded model checking instances. When abstract models are constructed directly from the UNSAT proof, the granularity can be at the gate level. Its major difference from our approach is that, it is a SAT-based method and is based on the construction of an UNSAT subformula, while ours is based on BDD computations. Furthermore, the abstractions are not cumulative in their method, and are not minimized with respect to the spurious counter-examples to remove the possibly redundant refinement variables.

The concept of *abstraction efficiency* was first mentioned in [16] as a performance measurement of refinement algorithms. The GRAB algorithm for picking refinement variables was also proposed there; experimental studies showed that, guided by all shortest counter-examples, GRAB is often superior to single counter-example driven algorithms. In this paper, we also adopt the algorithm for picking refinement variables of GRAB. However, the main concern of this paper is the abstraction granularity (the granularity of [16] is at the state variable level). We will demonstrate that, with the fine-grain abstraction and the use of sequential don't cares, our method significnatly outperforms GRAB. With the fine-grain approach, abstraction efficiency can be defined in terms of the number of transition relation clusters (e.g., latches and BNVs), or even the number of logic gates.

3. Prelimaries

In symbolic model checking, both transition graphs and sets of states are represented symbolically by Boolean functions. Let $x = \{x_1, ..., x_m\}$ be the present-state variables, $y = \{y_1, ..., y_m\}$ be the next-state variables, and $w = \{w_1, ..., w_n\}$ be the primary inputs; the model is represented symbolically by $M = \langle T(x, w, y), I(x)\rangle$, where $T(x, w, y)$ is the transition relation, and $I(x)$ is the set of initial states. The transition relation is the conjunction of all the transition *bit-relations*, each of which is associated with a binary state variable. Let $J = \{1, ..., m\}$, then

$$T(x, w, y) = \bigwedge_{j \in J} T_j(x, w, y_j) \ \ .$$

Here, $T_j(x, w, y_j) = y_j \leftrightarrow \Delta_j(x, w)$ is the *bit-relation* of the j^{th} binary state variable, and $\Delta_j(x, w)$ is the transition function in terms of the present-state variables and inputs.

In conventional, or coarse-grain, abstraction methods, the transition bit-relations of state variables are treated as atoms—a bit-relation is either included in or completely excluded from the abstract model, depending on whether the corresponding state variable is included or not. Let the abstract model contain a subset of state variables $\widehat{J} = \{1, ..., k\} \subseteq J$, and let $\hat{x} \subseteq x$ and $\hat{y} \subseteq y$ be the subsets of present- and next-state variables, respectively. The coarse-grain abstract model is represented by $M_a = \langle \widehat{T}, \widehat{I}\rangle$. The abstract transition relation is defined as follows:

$$\widehat{T}(x, w, \hat{y}) = \bigwedge_{j \in \widehat{J}} T_j(\{\hat{x}, \breve{x}\}, w, y_j) \ \ ,$$

where $\hat{x}$ is the set of *visible state variables,* and $\breve{x} = x \setminus \hat{x}$ is the set of the *invisible state variables*. The transition bit-relations of invisible variables are replaced with tautologies;

```
abstractionRefine($M, \psi$) {
    $M_a$ = initAbstraction($M, \psi$)
    **while** (1) {
        $ACEs$ = computeAbstractCEx($M_a, \psi$)
        **if** ($ACEs$ is empty)
            **return** TRUE
        $CCE$ = computeConcreteCEx($M, Gp, ACEs$)
        **if** ($CCE$ not empty)
            **return** (FALSE, $CCE$)
        $M_a$ = refineAbstraction($M, M_a, ACEs$)
    }
}
```

Figure 1. Generic abstraction refinement.

therefore, invisible variables are considered as inputs in the abstraction model (or *pseudo-primary inputs*). The abstract initial states $\widehat{I}(\hat{x})$ are the projection of $I(x)$—an abstract state is initial if and only if there is a concrete one inside it.

Since more transitions are allowed by $\widehat{T}$, an abstract model is an over-approximation of the concrete model; this naturally implies the simulation relation $M \preceq M_a$. Passing universal properties, such as the invariant property **G**p, are preserved: If $M_a \models \psi$, it is also true that $M \models \psi$; however, if $M_a \not\models \psi$, the property may not hold in M—that is, there may be *false negatives.*

The generic abstraction refinement framework for checking universal properties with over-approximated abstractions is given in Fig. 1. It starts with a primitive abstract model, which contains only the variables mentioned in the given property. The property is then checked in the abstract model, with the model checker. If $M_a \models \psi$, the property has been proven for the concrete model as well. Otherwise, a non-empty set of abstract counter-examples (ACEs) are generated. These counter-examples are then checked against the concrete model, to see if they contain real paths. If a concrete counter-example (CCE) exists, the property is refuted. Otherwise, some invisible variables and their corresponding transition bit-relations are added to refine M_a.

Counter-examples to the property **G**p (i.e., the propositional formula p is always true) are paths from initial states to the states labeled $\neg p$. All the counter-examples of the shortest length can be captured by a data structure called the Synchronous Onion Rings (SORs): With breadth-first search starting from the initial states, the sets of new states encountered in the BFS steps form the forward reachable onion rings. As soon as some $\neg p$ states are reached, the backward breadth-first search from these "bad states" gives the backward reachable onion rings. The pair-wise intersection of the forward and backward onion rings forms the SORs. Let $\{S^0, S^1, ..., S^L\}$ be the SORs, and $s_0 s_1 ... s_L$ be an arbitrary shortest counter-example; then, $s_i \in S^i$, $S^0 \subseteq I$, and $S^L \subseteq \neg p$.

A Boolean satisfiability (SAT) solver can be used to

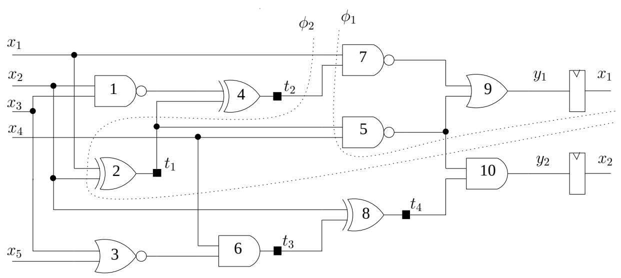

Figure 2. Example of fine-grain abstraction.

check whether the ACEs are real or not (i.e., if they pass the *concretization test*). The SAT problem is formulated as $\Psi = \Psi_M \wedge \Psi_S$, defined as follows:

$$\begin{aligned} \Psi_M &= I(X^0) \wedge \textstyle\bigwedge_{0 \le l < L} T(X^l, W^l, X^{l+1}) \\ \Psi_S &= \textstyle\bigwedge_{0 \le l \le L} S^l(X^l) \ , \end{aligned}$$

where Ψ_M represents the unrolling of the concrete model for L time frames, Ψ_S represents the constraints coming from the abstract SORs, X^l and W^l are the state and input variables at the l^{th} time frame, and $S^l(X^l)$ is the CNF encoding the l^{th} ring. Real counter-examples of length L exist if and only if the Boolean formula Ψ is satisfiable, in which case the satisfiable assignments induce a concrete counter-example.

The selection of refinement variables can be guided by the analysis of spurious counter-examples. The algorithm adopted in this paper for picking good refinement variables is the GRAB algorithm of [16]. We will briefly explain it in Section 5.2. However, our refinement must be conducted in both the *sequential* and the *Boolean* directions. Before delving into the details of refinement, we present our fine-grain abstraction approach in Section 4.

4. Fine-grain abstraction

Our fine-grain abstraction considers not only the state variables, but also the Boolean network variables. Boolean network variables are the intermediate variables selectively inserted in the combinational logic cones of latches; they are used to partition large combinational logic cones so that a compact BDD representation of their transition relations can be achieved. Once inserted, each Boolean network variable is associated with a small area of the combinational circuit; similarly to the state variables, there is an elementary transition relation for each Boolean network variable. The transition relation of the entire system is the conjunction of all these elementary transition relations. The following example shows how fine-grain abstraction works.

In Fig. 2, there are 10 gates in the fan-in combinational logic cones of the two latches; y_1 and y_2 are the next-state variables, and $x_1, ..., x_5$ are the present-state variables, among which, x_1, x_2 correspond to y_1, y_2. Let Δ_{y_1} be the output function of Gate 9 in terms of the present-state variables and inputs only; similarly, let Δ_{y_2} be the output function of Gate 10. Δ_{y_1} and Δ_{y_2} are also called the transition functions of Latch 1 and Latch 2, respectively. According to the definition in the previous section, the transition bit-relation of Latch 1 is defined as follows:

$$T_1 = y_1 \leftrightarrow \Delta_{y_1}(x_1, x_2, x_3, x_4) \ .$$

Boolean network variables can be inserted in the fan-in combinational cones to partition them into smaller pieces. To illustrate this, we insert 4 intermediate variables, t_1, t_2, t_3, and t_4, into this piece of circuit. We use δ_v to represent the output function of the signal v, but in terms of both present-state variables and Boolean network variables. These new functions and their corresponding finer grain elementary transition relations are defined as follows:

$$\begin{aligned} \delta_{t_1} &= x_1 \oplus x_2 & T_{t_1} &= t_1 \leftrightarrow \delta_{t_1} \\ \delta_{t_2} &= \neg(x_2 \wedge x_3) \oplus t_1 & T_{t_2} &= t_2 \leftrightarrow \delta_{t_2} \\ \delta_{t_3} &= \neg(x_3 \vee x_5) \wedge x_4 & T_{t_3} &= t_3 \leftrightarrow \delta_{t_3} \\ \delta_{t_4} &= x_2 \oplus t_3 & T_{t_4} &= t_4 \leftrightarrow \delta_{t_4} \\ \delta_{y_1} &= \neg(x_1 \wedge t_2) \vee \neg(x_4 \wedge t_1) & T_{y_1} &= y_1 \leftrightarrow \delta_{y_1} \\ \delta_{y_2} &= \neg(x_4 \wedge t_1) \wedge t_4 & T_{y_2} &= y_2 \leftrightarrow \delta_{y_2} \end{aligned}$$

Note that y_1 is now associated with the elementary transition relation δ_{y_1} only. The transition bit-relation of Latch 1 is a conjunction of three elementary transition relation clusters

$$T_1 = T_{y_1} \wedge T_{t_1} \wedge T_{t_2} \ .$$

In coarse-grain abstraction methods where only state variables are treated as atoms, when Latch 1 is included in the abstract model, all the six fan-in gates (Gates 1, 2, 4, 5, 7, and 9) are also included; that is, $\widehat{T} = T_{y_1} \wedge T_{t_1} \wedge T_{t_2}$. However, not all these gates might be necessary for the verification of the given property, even if Latch 1 is indeed necessary. In our *fine-grain* abstraction, Boolean network variables, as well as the latches, are treated as atoms. When Latch 1 is in the abstraction, only those gates covered by the elementary transition relation cluster T_{y_1} are included; this is indicated in the figure by the cut ϕ_1, which contains Gates 5, 7, and 9. In successive refinements, only the clusters that are relevant to the verification are added. In the next section, we present an algorithm that can identify which clusters should be included. Meanwhile, let us assume that the current abstract model contains only Latch 1 (e.g., includes Gates 5, 7, and 9), and the property fails in the abstract model. At this point, we may add the Boolean network variable t_1 as indicated by the new cut ϕ_2 in the figure; this means $\widehat{T} = T_{y_1} \wedge T_{t_1}$. Note that this time the abstract model contains Latch 1 and Gates 2, 5, 7, and 9. Continuing this process, we may keep adding $y_2, t_4, \ldots$ until we get a proof or refutation.

It is possible in our fine-grain approach that gates covered by the transition cluster T_{t_2} (i.e., Gates 1 and 4) never appear in the abstract model—if they are indeed irrelevant to the verification of the given property. This demonstrates the advantage of fine-grain abstraction. In a couple of real-world circuits, we have observed that over 90% of the gates in some large fan-in cones are indeed redundant, even though the corresponding latches are necessary for verification of the given property.

The granularity of abstraction depends on the size of the transition relation clusters, as well as the algorithm used to perform the partition. In our current implementation, the *frontier* [14] method is used to selectively insert the Boolean network variables. The procedure works as follows: First, the elementary transition function of each gate is computed from the combinational inputs to the combinational outputs, in some topological order. If the BDD size of an elementary transition function exceeds a given threshold, a Boolean network variable is inserted to associate with that gate. For all the gates in the fan-outs of that gate, their elementary transition functions are computed in terms of the new Boolean network variable.

When the partition threshold is set to 1, a Boolean network variable will be inserted for every logic gate; in this extreme case, the optimal abstraction, among all the possible final proofs, is the one that contains the smallest number of gates. In other words, abstraction refinement becomes a process of synthesizing such an optimal abstract model.

5. Smart refinement

Refinement is to select a small set of invisible variables and then add their elementary transition relations back to $\widehat{T}$. To achieve higher abstraction efficiency, we want to add a small subset of invisible variables such that after refinement, the chance of removing the spurious ACEs is maximized. This boils down to three questions: (1) what type of variables to add? (2) how to identify the important variables? and (3) how to remove the possible redundant variables from the refinement set? For picking the important variables, we use the GRAB algorithm in [16], and will explain it briefly in Section 5.2. The other two questions, however, are unique to our fine-grain abstraction approach.

5.1. Refinement directions

With the fine-grain abstraction, refinement can be conducted in two different directions. In the *sequential* direction, more invisible state variables can be added; in the *Boolean* direction, more Boolean network variables (e.g. logic gates) can be added. Adding Boolean network variables may remove the transitions allowed by the current abstract model; adding state variables, however, will also increase the state space.

According to our experience, if no distinction is made between these two types of variables, the refinement result can be quite sub-optimal—many redundant state variables may be added during the refinement process. This suggests that we need a method to predict which refinement direction to go at a certain time. A satisfiability check similar to the *concretization test* can be used to get the refinement direction. The idea is that, if adding the entire fan-in cones of the current visible state variables cannot remove the spurious ACEs, more invisible state variables are needed—we should refine in the sequential direction; otherwise, we refine in the Boolean direction.

Given a fine-grain abstract model, its *extended abstract model* is defined as the one with the same set of visible state variables, but containing all the gates in their fan-in cones. Refer to Fig. 2 as an example: when the current abstract model contains Latch 1 and Gates 5, 7, and 9, the extended abstract model contains Latch 1, and Gates 1, 2, 4, 5, 7, and 9. Let $\widehat{T}_\epsilon$ be the transition relation of the extended abstract model, the SAT formula for predicting the refinement direction is defined as $\Psi' = \Psi'_M \wedge \Psi_S$, where

$$\Psi'_M = I_0(X^0) \wedge \bigwedge_{0 \le i < L} \widehat{T}_\epsilon(X^i, W^i, X^{i+1})$$

is the unrolling of the *extended abstract model* for exactly L time frames, and Ψ_S is the constraint from the SORs. If Ψ' is unsatisfiable, we choose to refine in the Boolean direction. Otherwise, even adding all the logic gates in the Boolean direction cannot kill all the spurious counter-examples; therefore, we need to refine in the sequential direction by adding more state variables.

5.2. Refinement variable selection

For picking good refinement variables, we use the GRAB algorithm in [16], which is explained with the example in Fig. 3 as follows. Let f and g be the invisible variables appearing in one segment of the spurious abstract SORs; according to the figure: (1) Before refinement, every state in S^i has successors in the next ring—this is because invisible variables are treated as inputs and are existentially quantified during symbolic traversal. Therefore, we can always reach the next ring from every state in S^i. (2) After f is added to the abstract model, only three out of the four new states have successors in the next ring—State $[2, \neg f]$ now becomes a dead-end state. Note that f becomes a state variable after refinement. (3) After g is added instead of f, only two out of the four new states have successors in the next ring—both $[1, \neg g]$ and $[2, g]$ become dead-end states.

From the above analysis, we draw the following conclusions: (1) g is a better candidate for refinement; and (2)

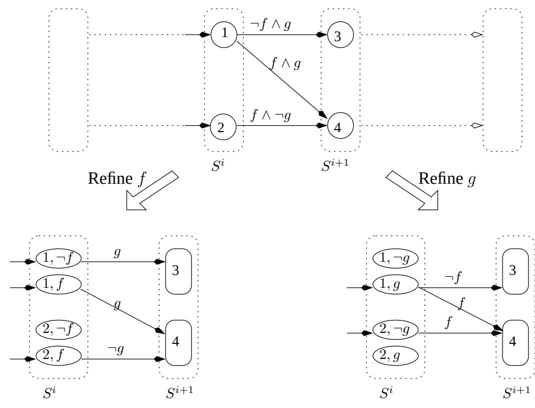

Figure 3. Analysis of refinement variables.

even if we refine both f and g, the spurious counter examples may still exist—$\{f, g\}$ is not a sufficient refinement set, and we need to add more invisible variables. The quantitive measurement of the chance of reaching the next level after refining f is computed as follows(cf. [16]):

$$N^{\{f\}} = \frac{|S^j(\widehat{x}) \wedge \exists(\check{x}, w).\forall f.\exists\widehat{y}.[\widehat{T}(\check{x}, \widehat{x}, w, \widehat{y}) \wedge S^{j+1}(\widehat{y})]|}{|S^j(\widehat{x})|},$$

where $|.|$ is the cardinality of a set. For more detailed explanation, the readers are referred to the original paper.

5.3. Refinement minimization

Once the entire spurious SORs are gone, all the newly added variables forms a sufficient refinement set—that is, they are sufficient for removing all the current length spurious ACEs. This refinement set, however, may not be minimal. SAT-based greedy minimization with respective to the entire SORs can be used to remove the possible redundant variables [17, 2]. The greedy minimization is based on trial-and-error: for each variable in the new refinement variable list, if dropping it does not make the spurious counter examples come back, we know it is redundant; otherwise, it is necessary. In our approach, however, the minimization must be applied in both refinement directions.

For the ACEs with a certain length, we choose to refine first in the sequential direction. As soon as a sufficient set of latches are added, we minimize it with respect to the entire bundle of ACEs, before shifting to the Boolean direction. Every time a latch is removed, all the Boolean network variables that are used only by this latch are also pruned away. Note that although we have added a sufficient set of latches, the ACEs may not be killed entirely at this point. After shifting to the Boolean direction, we keep adding Boolean network variables only until the entire SORs is removed; at this point, we greedily minimize the set of newly added Boolean network variables.

6. Sequential don't cares

Previous work in abstraction refinement divided the original system into two parts: a set of visible variables and a set of invisible variables. Model checking was applied to the abstract model that contains only the elementary transition relations of visible variables. The elementary transition relations of invisible variables, on the other hand, were completely ignored. Because the transition constraints are removed, the invisible variables are treated as *pseudo-primary inputs* in model checking—they can take arbitrary values at all times. These pseudo-primary inputs are the reason why the abstract counter-examples may be spurious: The valuations of these variables that are responsible for triggering these counter-examples may not be allowed in the concrete system. In this section, we show that with additional analysis of the invisible part of the system, we can further constrain the invisible variables.

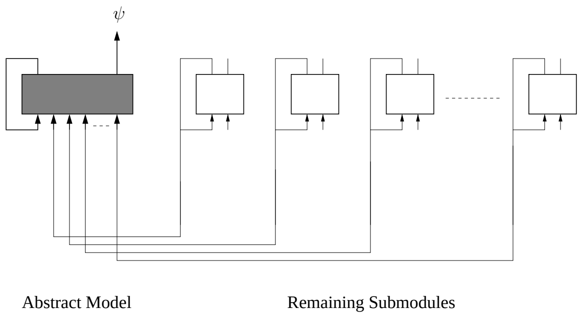

Figure 4. DCs from remaining submodules.

As illustrated in Fig. 4, we decompose the invisible part of the system into a series of submodules, each of which contains a subset of the invisible latches. The decomposition is based on the machine decomposition algorithm originally proposed by Cho *et al.* [3] in the context of reachability analysis and improved in [12]. Approximate reachable states of the invisible part can be computed by analyzing each submodule in turn, assuming that the other submodules are in any states that have already been estimated to be reachable. The process is iterated until a least fixpoint is reached.

The set of approximate reachable states of the invisible part is an upper bound on the set of exact reachable states. It can be used to constrain the behaviors of the invisible variables, or pseudo-primary inputs, of the abstract model. If certain valuations of the invisible variables are not even in the set of approximate reachable states, they can never appear in the original system. During the reachability analysis of the abstract model, these pseudo-primary input conditions can be disabled.

In our current implementation, the machine decomposition is applied on the entire system, followed by the LMBM traversal of the submachines [12]. The approximate reachable states are computed only once at the beginning; they are then used in abstraction refinement to constrain the forward reachability analysis of the abstract models at different abstraction levels. Specifically, the BDD operation *constrain* [6] is used to remove spurious transitions from $\widehat{T}$ using the approximate reachable states as the *care set*. Constraints on the behavior of the abstract model due to the neighboring submachines prevent some spurious abstract counter-examples from occurring, possibly leading to the decision of a property earlier in the refinement cycle.

A more systematic integration of machine decomposition and approximate reachability into the abstraction refinement paradigm is possible. The result is a multi-way partition refinement process. Partitioning of the model into submachines can be done so that the abstract model is one of the many submachines. Refinement is considered as merging the abstract model with some other submachines. We leave this for the future work.

7. Experiments

We have implemented our new methods in VIS-2.0 [1, 15], and compared it with the GRAB algorithm in [16]. Our implementation uses the CUDD package for the BDD-based computation, and Chaff [13] for the SAT-based computation. The comparison is based on the same set of test cases used in [16], which are real-world hardware designs with invariant properties, kindly provided by the authors of [2]. In our implementation, the *frontier* partition threshold was set to 1000: every time the BDD size of the transition function went beyond this threshold, a Boolean network variable was inserted. The latch group size for computing the approximate reachable states with LMBM was set to 8. Dynamic variable reordering was enabled with method *sift* for all BDD operations. The experiments were done on 1.7 GHz Intel Pentium 4 machines with 2 GB of RAM. We set a time limit of 8 CPU hours. The experimental results are shown in Table 1.

The first four columns of Table 1 give the statistics of the test cases: the first column shows the names of the designs; the second and third columns shows the numbers of registers and gates in the cone of influence, respectively. The forth column indicates whether the properties are true (T) or false (F). When the properties are false, the lengths of the shortest counter-examples are given. The following six columns compare the performance of three different methods: GRAB is the algorithm in [16], FINEGRAIN is our fine-grain abstraction method, and +ARDC is our fine-grain abstraction method augmented with the use of sequential don't cares. For each method, the CPU time in seconds and the number of registers in the final abstract model are shown. Note that, for the purpose of controlled experiments, the underlying algorithm for picking refinement variables is the same for the three methods.

Test Cases				GRAB [16]		FINEGRAIN		+ARDC	
name	regs	gates	T/F	CPU	regs	CPU	reg	CPU	regs
D24-p1	147	8 k	9	1	4	1	4	1	4
D24-p2	147	8 k	T	3	8	3	8	3	8
D24-p3	147	2 k	T	20	8	4	6	**2**	5
D24-p4	147	8 k	T	43	8	4	6	**2**	5
D24-p5	147	8 k	T	3	5	4	6	**2**	5
D12-p1	48	2 k	16	**14**	23	24	23	19	24
D23-p1	85	3 k	5	20	21	**3**	21	14	21
D5-p1	319	25 k	31	**31**	18	42	13	32	13
D1-p1	101	5 k	9	**9**	21	12	20	14	20
D1-p2	101	5 k	13	51	23	**27**	23	29	23
D1-p3	101	5 k	15	56	25	**32**	23	33	23
D16-p1	531	34 k	8	92	14	25	14	**21**	14
D2-p1	94	18 k	14	180	48	108	49	**59**	48
M0-p1	221	29 k	T	**136**	16	204	13	942	13
rcu-p1	2453	38 k	T	195	10	**188**	10	216	10
B-p0	124	2 k	T	**1256**	24	1507	24	1484	24
B-p1	124	2 k	T	173	18	189	19	**159**	18
B-p2	124	2 k	17	93	7	95	7	**90**	7
B-p3	124	2 k	T	223	43	76	43	**62**	43
B-p4	124	2 k	T	393	42	**101**	43	108	42
D22-p1	140	7 k	10	720	132	242	132	**191**	132
D4-p2	230	8 k	T	1103	38	204	38	**195**	38
D21-p1	92	14 k	26	2817	66	2725	70	**622**	67
D21-p2	92	14 k	28	4635	70	1748	75	**868**	67
IU-p1	4494	154 k	T	$>$8 h	-	**2226**	12	2263	12
IU-p2	4494	154 k	T	$>$8 h	-	930	14	**699**	12

Table 1. Comparison of the three algorithms.

Our fine-grain abstraction approach shows a significant performance improvement over GRAB. First, it finishes the two largest test cases that cannot be verified by GRAB. Careful analysis of *IU-p1* and *IU-p2*, two problems from the instruction unit of the PicoJava microprocessor, shows that some of their registers have extremely large fan-in combinational logic cones. Without the fine-grain abstraction, abstract models with less than 10 registers will be too complex to be dealt with by the model checker. Furthermore, for the test cases that both methods managed to finish, FINEGRAIN is significantly faster than GRAB. In fact, the total CPU time to finish these remaining 24 test cases is 12,207 seconds for GRAB, and 7,562 seconds for FINEGRAIN.

With the use of sequential don't cares (DCs), the performance of our method is further improved. For more than half of the 26 test cases, +ARDC is significantly faster than both FINEGRAIN and GRAB; for the remaining ones, the performance is comparable. The total CPU time to finish all the 26 test cases is 10,724 seconds for FINEGRAIN, and 8,130 seconds for +ARDC; this is an average of 25% speed-up.

8. Conclusion

We have introduced a finer grain abstraction refinement approach for model checking industrial-scale digital systems. Boolean network variables are selectively inserted to partition large combinational logic cones into smaller pieces. These Boolean network variables, together with the state variables, are treated as *atoms* for abstraction refinement. With this fine-grain approach, refinement can be conducted in two different directions, *sequential* and *Boolean*. We have proposed a SAT-based method for computing the appropriate direction, and have applied greedy minimization in both directions to keep the *refinement set* small. Experimental results have shown that our fine-grain abstraction and smart refinement is a must for dealing with large designs. In particular, the two large test cases can only be verified with fine-grain abstraction enabled.

We have also explored the use of approximate reachable states of the remaining submodules to help verifying the abstract model. This, in general, corresponds to multi-way machine decomposition. According to our experimental studies, the use of DCs has improved the performance of abstraction refinement significantly.

Future work includes exploring the many different methods for inserting Boolean network variables into the combinational logic cones, which is currently limited to the *frontier* method. Machine decomposition and approximate reachability analysis of the remaining submodules can also be integrated more tightly into the abstraction refinement process. In particular, the reachable states of the abstract model can be fed back to the remaining submodules to improve the quality of the ARDCs. With this approach, abstraction refinement becomes a special case of the machine decomposition and approximate state space traversal.

References

[1] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[2] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, Nov. 2002. LNCS 2517.

[3] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. *IEEE Transactions on Computer-Aided Design*, 15(12):1465–1478, Dec. 1996.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 154–169. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[5] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.

[6] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 126–129, Nov. 1990.

[7] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 176–191, Warsaw, Poland, Apr. 2003. LNCS 2619.

[8] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 416–423, Nov. 2003.

[9] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[10] B. Li, C. Wang, and F. Somenzi. A satisfiability-based approach to abstraction refinement in model checking. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/volume89.html.

[11] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, Warsaw, Poland, Apr. 2003. LNCS 2619.

[12] I.-H. Moon, J. Kukula, T. Shiple, and F. Somenzi. Least fixpoint MBM: Improved technique for approximate reachability. Presented at IWLS99, Lake Tahoe, CA, June 1999.

[13] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[14] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, Lake Tahoe, CA, May 1995.

[15] URL: http://vlsi.colorado.edu/∼vis.

[16] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 408–415, Nov. 2003.

[17] 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 *Proceedings of the Design Automation Conference*, pages 35–40, Las Vegas, NV, June 2001.