Abstraction Refinement by Controllability and Cooperativeness Analysis

Freddy Y.C. Mang and Pei-Hsin Ho

Advanced Technology Group, Synopsys, Inc.

{fmang, pho}@synopsys.com

ABSTRACT

We present a new abstraction refinement algorithm to better refine the abstract model for formal property verification. In previous work, refinements are selected either based on a set of counter examples of the current abstract model, as in [5][6][7][8][9][19][20], or independent of any counter examples, as in [17]. We (1) introduce a new "*controllability*" analysis that is independent of any particular counter examples, (2) apply a new "*cooperativeness*" analysis that extracts information from a particular set of counter examples and (3) combine both to better refine the abstract model. We implemented the algorithm and applied it to verify several real-world designs and properties. We compared the algorithm against the abstraction refinement algorithms in [19] and [20] and the interpolation-based reachability analysis in [14]. The experimental results indicate that the new algorithm outperforms the other three algorithms in terms of runtime, abstraction efficiency (as defined in [19]) and the number of proven properties.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: *Computer-Aided Design (CAD).*

General Terms

Algorithms, Experimentation, Verification.

Keywords

Formal Verification, Abstraction Refinement, Controllability, Cooperativeness.

1. INTRODUCTION

Formal property verification exhaustively verifies logic designs against some desired properties of the designs with respect to all possible input sequences of any length. In this paper we focus on the verification of *safety properties*. Informally, safety properties specify that some "bad" states are not reachable from the initial states through any traces of the design. A counter example of a safety property is a trace that reaches a bad state from an initial state of the design.

Without abstraction, state-of-the-art formal property verification engines based on Binary Decision Diagrams (BDD's) [16] or clauses in Boolean satisfiability (SAT) solvers [15] usually cannot verify properties of designs with more than a couple of hundred registers. As a result, formal property verification relies on automatic abstraction techniques to verify real-world logic designs.

Interpolation-based reachability analysis [14] and abstraction refinement [5][6][7][8][9][17][19][20] are commonly viewed as the most practical automatic abstraction methods. Interpolation-based reachability analysis uses a SAT solver to compute the interpolants of two Boolean formulas derived from the bounded-model-checking formula in [3] on the whole design as a conservative abstraction of the set of reachable states from the initial states of the design.

Abstraction refinement incrementally refines the abstract model, a subset of the design, by including more and more logic from the original design until the underlying formal property verification engine verifies or falsifies the property. More precisely, the generic abstraction refinement algorithm is as follows.

1. Generate the abstract model.
2. Prove the property or search for a counter example on the abstract model; if the property is proven for the abstract model, stops (it is proven for the original design).
3. Use the counter example found on the abstract model to guide the search of a counter example on the original design; if a counter example is found, stops (the property is falsified for the original design).
4. Refine the abstract model by adding more details to the abstract model; go back to Step 1.

Figure 1 Abstraction refinement algorithm

Since the performance of an underlying formal property verification engine decreases as the complexity of the abstract model increases, the biggest challenge of an abstraction refinement algorithm is to buildup an abstract model that is as simple as possible but contains enough details to verify the property. Therefore, Step 4 is usually an abstraction refinement algorithm's key differentiator, since it determines if the property can be proven at Step 2 or falsified at Step 3.

In this paper we present a new algorithm to improve Step 4. In previous work, the refinement schemes are computed either based on a set of counter examples of the current abstract model as in [5][6][7][8][9][19][20] or independent of any counter examples of the current abstract model as in [17]. We introduce (1) a counter-example independent "*controllability*" analysis, (2) a counter-example dependent "*cooperativeness*" analysis and (3) combine both methods to better refine the abstract model.

DAC 2004, June 7-11, 2004, San Diego, California, USA.
Copyright 2004 ACM ***1-58113-828-8/04/0006***…$5.00.

We implemented the algorithm and compared it against the abstraction refinement methods in [19] and [20] and the interpolation-based reachability analysis in [14]. Experimental results indicate that the new method outperforms the other three methods in terms of runtime, abstraction efficiency (how much of the design is required to prove the properties, as defined in [19]) and the number of proven properties.

The rest of the paper is organized as follows. In Section 2 we introduce some terminology before we present the algorithm in Section 3. We survey the related work in Section 4. We present the experimental results in Section 5 and conclude the paper in Section 6.

2. PRELIMINARIES

We define in this section the syntax and semantics of designs and properties and the *k*-controllability and *k*-cooperativeness.

2.1 Designs and Safety Properties

At a high level, a design consists of a collection of inputs, registers and the transition functions of the registers. Formally, a *design* $D = (V, T)$ is an ordered pair where $V = \{v_1, ..., v_n\}$ is a set of Boolean variables and $T = \{t_1, ..., t_m\}$ is a set of Boolean functions called the *transition functions*. A *(partial) state* s is a Boolean function whose domain is (a subset of) the set V of variables. Each transition function t_i of T maps each state s to a Boolean value and is associated with a distinct variable v_i of V. A variable v_i is called a *register* if it has a correspondent transition function in T and otherwise an *input*. Let $Q(D)$ and $R(D)$ denote respectively the set of inputs and registers of the design D. The relation $\tau(D) = \wedge_{v_i \in R(D)} (v_i' = t_i)$ is the *transition relation* of the design D where the primed variables v_i' are the *next-state variables* of the registers v_i.

A *(partial) trace* of the design D is a sequence $\delta = s_0, ..., s_k$ such that (1) $s_0, ..., s_k$ are (partial) states of D and (2) $\tau(D)(s_i, s_{i+1})$ for each $i = 0, ..., k-1$. If $\delta = s_0, ..., s_k$ is a trace of D we say that the state s_k is *reachable* from the state s_0.

Two partial states are *consistent* if they map each variable in the intersection of their domains to a consistent value. The union (s_1, s_2) of two consistent partial states is a partial state that maps each variable v in the domain of s_1 to $s_1(v)$ and each variable v in the domain of s_2 to $s_2(v)$. Two partial traces are *consistent* if they are of the same length and the corresponding states of the partial traces are consistent. An *(partial) input vector* u is a partial state whose domain is (a subset of) the set $Q(D)$ of inputs. An *(partial) input trace* is a partial trace of (partial) input vectors. For each input trace $\varepsilon = u_0, ..., u_k$ and starting state s_0 that is consistent with the input vector u_0, there is a unique trace $\delta = s_0, ..., s_k$ that is consistent with the input trace ε. We say that the input trace ε *generates* the trace δ. A partial trace δ is *valid* if there exists a trace that is consistent with the partial trace δ. Otherwise it is *invalid*. Note that every input trace is valid.

With a construction similar to the one in [12], we can define without loss of generality that a *safety property* P specifies an *initial state* s_a and a *fail variable* v_f. A *counter example* $\delta = s_0, ..., s_k$ of a design D for a safety property P is a trace such that $s_0 = s_a$ and $s_k(v_f) = 1$. A state that maps the fail variable to 1 is called a *fail state*. The safety property *P* is *True* for the design *D* if and only if there does not exist a counter example of *D* for the property *P*. An input trace ε *violates* the safety property *P* if ε generates a counter example of *D* for the property *P*. In this paper we only consider safety properties, the most widely verified properties.

A design $C = (V, T')$ is an *abstract model* of the design *D* if the set T' is a subset of the transition functions *T* of *D*. In other words, some registers of the design *D* become inputs of the abstract model *C*. Clearly, if a property is *True* for an abstract model of *D*, the property is also *True* for the design *D*.

2.2 K-Controllability and K-Cooperativeness

To define *k*-controllability and *k*-cooperativeness, we first introduce the notion of *k-dominancy*. Let $C = (V, T')$ be an abstract model of the design *D*. We say that a subset *Q* of the inputs of *C* is *k-dominant* for the abstract model *C* and the property *P* if for any partial input trace $w_0, ..., w_k$ of the rest of the inputs $Q(C) \setminus Q$, there exists a partial input trace $u_0, ..., u_k$ of the inputs Q such that the input trace $(u_0, w_0), ..., (u_k, w_k)$ is not consistent with any counter example of *C* for the property *P*. In other words, imagine a game of two players played on the abstract model *C*, in which player 1 wins if she can control the inputs $Q(C) \setminus Q$ to drive the abstract model to a fail state in *k* steps and player 2 wins if she can control the inputs Q to steer the abstract model clear of any fail states for *k* steps. Then player 2 can win if and only if *Q* is *k*-dominant.

Second we introduce the notion of *partitioned abstract models*. A *partitioned abstract model* $(C_c, C_s) = (V, T_c \cup T_s)$ of the design *D* is an abstract model where $C_c = (V, T_c)$ and $C_s = (V, T_s)$ are also abstract models called respectively the *core* and the *shield* such that the registers $R(C_s)$ of the shield is a subset of the inputs $Q(C_c)$ of the core. In other words, the registers of the shield drive some inputs of the core. Figure 2 depicts a partitioned abstract model of a design *D*. Among the five inputs of the core, inputs *r1* and *r2* are driven by the registers of the shield, inputs *q1* and *q2* are driven by some other registers of the design *D* and input *i1* is an input of *D*. The triangles represent the transition functions of these variables.

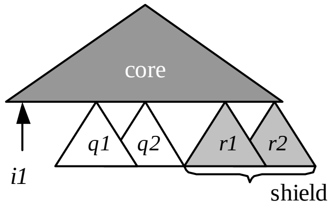

Figure 2 Partitioned abstract model

Let $C = (C_c, C_s)$ be a partitioned abstract model of a design *D*. An input $v_i \in (Q(C_c) \cap R(D)) \setminus R(C_s)$ of the core not driven by the shield (e.g., inputs *q1* and *q2* in Figure 2) is *k-controlling* for the partitioned abstract model *C* if (1) the set $Q(C_c) \cap R(C_s)$ of inputs of the core driven by the shield (e.g., inputs *r1* and *r2* in Figure 2) is not *k*-dominant for the core C_c but (2) the set $\{v_i\} \cup (Q(C_c) \cap R(C_s))$ is *k*-dominant for the core C_c. In the 2-player game scenario, suppose that the players played the game on the core and player 2 was controlling the set of inputs driven by the shield and lost because the set $Q(C_c) \cap R(C_s)$ is not *k*-dominant. If player 2 gets to pick one additional input to control in a new game, then player 2 will be able to win the new game if and only if she picks a *k*-controlling input. Note that *k*-controllability is counter-example independent.

An input $v_i \in (Q(C_c) \cap R(D)) \setminus R(C_s)$ with transition function t_i is *k-cooperative* for the partitioned abstract model $C = (C_c, C_s) = (V, T')$ and the property *P* if (1) there is an input trace $\varepsilon = u_0, ..., u_k$ of *C* violating the property *P* and (2) the partial trace ε is invalid on the abstract model $(V, T' \cup \{t_i\})$. Clearly, *k*-cooperativeness is counter-example dependent.

On the one hand, the *k*-controllability estimates the potential of a variable in invalidating counter examples if it was under full control of player 2 (but in reality, the transition function, not player 2, controls the behavior of the variable). On the other hand, the *k*-cooperativeness demonstrates that the transition function is cooperative with player 2 on invalidating *some* counter examples. Variables that are both *k*-controlling and *k*-cooperative have the potential and seem to be willing to help prove the property. Therefore we want to include their transition functions to refine the abstract model.

3. ABSTRACTION REFINEMENT ALGORITHM

Our algorithm aims to do a better job in Step 4 of the generic abstraction refinement algorithm in Figure 1. The first three steps of our abstraction refinement algorithm are the same as the RFN method in [20]. In Step 1, the abstract model initially contains only the fail variable and its transition function. In Step 2 RFN applies a hybrid ATPG-BDD method to find a set of counter examples represented by a sequence of cubes (partial input vectors). If the property is *True* for the abstract model, RFN reports that the property is *True* for the design and terminates. Otherwise in Step 3 RFN uses the counter example found on the abstract model to guide sequential ATPG to search for a counter example on the design. If such a counter example is found, RFN reports that the property is *False*, reports the counter example and terminates. Otherwise RFN proceeds to Step 4.

In the rest of the section, we will discuss how to compute *k*-controllability and *k*-cooperativeness and use them to better refine the abstract model in Step 4.

3.1 Computing *k*-Controllability

Let $C = (V, T')$ be an abstract model, let Q be a subset of inputs of the C, and let φ be a Boolean formula representing a set of states of the abstract model. We define the *controllable predecessor predicate* to be the following Boolean predicate:

$$cpre(\varphi) = \forall (Q(C) \setminus Q). \exists Q. \exists R'(C). (\tau(\mathrm{C}) \wedge \varphi') ,$$

where $R'(C)$ is the set of the next-state variables and φ' is the Boolean formula obtained by substituting all variables in φ by their corresponding next-state variables. In other words, the controllable predecessor predicate $cpre(\varphi)$ computes a set of states of the abstract model *C* such that, no matter what values player 1 choose for the inputs $Q(C) \setminus Q$, player 2 can always choose some values for the inputs Q to transition to some states in φ. The controllable predecessor predicate can be computed by BDD operations.

To check if the set Q of inputs is *k*-dominant, we iteratively compute the controllable predecessor predicate from the non-fail states as follows. Let $X_0 = \neg v_f$. For $i = 0, ..., k-1$, we compute $X_{i+1} = \neg v_f \wedge cpre(X_i)$, which represents the set of states from which player 2 can avoid violating the property *P* for i steps. Thus the following theorem follows.

Theorem 1. The set Q of inputs is *k*-dominant if and only if $s_a \rightarrow X_k$, where s_a is the initial state of the design *D*.

Knowing how to check whether a set of inputs is *k*-dominant we can determine if a variable $v_i \in (Q(C_c) \cap R(D)) \setminus R(C_s)$ of a partitioned abstract model (C_c, C_s) is *k*-controllable by the definition of *k*-controllability.

We implemented the above algorithm using the CUDD [18] package. We simplify each predicate X_i by treating the states that are not reachable from the initial state in *k-i* steps as don't cares in the BDD operations. The predicate that represents the set of reachable states of the abstract model C can be computed by a BDD-based forward image computation from the initial state.

3.2 Computing *k*-Cooperativeness

Given an abstract model *C* and a partial trace $\delta = s_0, ..., s_k$ that represents a set of counter examples of *C* for *P*, we identify a subset of the *k*-cooperative inputs of *C* by performing 3-value simulation on the design. The advantage of 3-value simulation is that the runtime is linear in the size of the design times the length *k* of the partial trace. We use the unknown value **X** to over-approximate the value of a variable that is of value either 0 or 1. As a result, we may not identify all *k*-cooperative inputs.

We arrange the design so that the registers and inputs of the design are on the left and the next-state registers are on the right. At the *i*-th step of the 3-value simulation, for each $i = 0, ..., k-1$, we (1) drive the registers and inputs v on the left with $s_i(v)$ if v is in the domain of s_i and with the unknown value **X** if v is not in the domain of s_i, (2) perform 3-value simulation on the design to compute a partial state s'_{i+1} of the next-state variables on the right, and (3) record the *conflict variables*, variables $v \in (Q(C_c) \cap R(D)) \setminus R(C_s)$ that have conflicting values (the unknown value **X** does not conflict with any value) between the partial states s'_{i+1} and s_{i+1}. It is clear that the following theorem is true.

Theorem 2. If a variable $v \in (Q(C_c) \cap R(D)) \setminus R(C_s)$ has conflicting values in the result s'_{i+1} of the *i*-th cycle of 3-value simulation and the next partial state s_{i+1} of the counter example, then the variable v is *k*-cooperative.

The *k*-cooperative variables are ranked and placed in a priority queue first by their *frequencies of appearances*, the number of conflicts that they introduced in the counter example (the higher the better); then by *sequential distances*, the least number of registers on a path between the variable and the fail variable (the less the better); and finally by *input widths*, the number of inputs that appear in the BDD representation of the transition function of the variable (the fewer the better). We will use this priority queue in the following abstraction refinement algorithm.

3.3 Abstraction Refinement

In Step 4, we have a set of counter examples for the abstract model C represented by a partial trace $\delta = s_0,...,s_k$ of length k. We maintain the abstract model $C = (C_c, C_s)$ in the form of a partitioned abstract model. We select exactly one transition function to be included in the abstract model in every iteration of the abstraction refinement algorithm. This differs from RFN, which may add multiple transition functions in an iteration.

We first compute the priority queue of the *k*-cooperative variables using the simulation-based method in Section 3.2. Before the property is proven, the queue is seldom empty in our experience. If the queue is indeed empty, we apply the netlist-topology based BFS method in [11] to select an input of the core that is not driven by the shield as our variable for refinement. We then add the transition function of the chosen variable to the shield of the abstract model.

Otherwise the priority queue of *k*-cooperative variables is not empty and we search for a *k*-controlling variable among the first three variables in the queue using the BDD-based method in Section 3.1. The first *k*-controlling variable of the top three *k*-cooperative variables becomes our variable for refinement. In our experience, there is a good chance (greater than 30%) finding such a refinement variable at each iteration of the abstraction refinement algorithm. We then add the transition function of the refinement variable to the shield of the abstract model.

We must maintain a loop invariant during the abstraction refinement iterations: the set of inputs of the core driven by the shield is not *k*-dominant for the core. Without this loop invariant, every input of the core not driven by the shield might become *k*-controlling in the next iteration of the abstraction refinement algorithm. To maintain the loop invariant, after including a *k*-controlling variable into the shield, we move one-by-one the “oldest” variables (the variables that were included first) of the shield into the core, until the set of inputs of the core driven by the shield is no longer *k*-dominant.

If we do not find a *k*-controlling variable among the first three *k*-cooperative variables, we simply add the head of the priority queue, a *k*-cooperative variable, to the shield of the abstract model. Note that we could have searched deeper in the priority queue for *k*-controlling variables. But since *k*-controllability analysis requires non-trivial BDD operations, checking only the first three variables in the priority queue for *k*-controllability makes a good tradeoff.

4. RELATED WORK

Kurshan first introduced the abstraction refinement algorithm and implemented it in the tool COSPAN [13]. But there is little published detail about the method.

Clarke et al [6] proposed an abstraction-refinement algorithm for ACTL* model checking. The method identifies in a counter example (1) the *deadend states* that are reachable from the initial states in the design and can reach the fail states in the abstract model, and (2) the *bad states* that are reachable from the initial states in the abstract model and can reach the fail states in the design. The method refines the abstract model to separate the deadend states from the bad states. It requires BDD-based image computation on the whole design, which severely limits its capacity. The methods in [5] and [7] were designed to fix this capacity problem by applying SAT solvers and ILP solvers to identify deadend states and pick minimal refinement to separate the deadend states from the bad states. However, in our experience, when the design is large, running SAT solvers on the whole design is often time consuming and likely to produce many conflict clauses that do not help zero in on good refinement candidates for proving the property. Experimental results in [19] seem to agree with this observation.

The RFN method in [20] was designed to verify large real-world designs by avoiding expensive computations on the whole design. It also applies a BDD-ATPG hybrid method to find counter examples on bigger abstract models, which limits RFN to only consider the set of counter examples that can be represented by a single sequence of cubes, or a partial input trace. The method in this paper is basically an improvement of RFN by controllability and cooperativeness analysis. We compare our algorithm against RFN in Section 5.

The work in [8] enhanced RFN by (1) considering multiple counter examples represented by a sequence of BDDs instead of cubes to identify refinement candidates and (2) performing refinement at a finer granularity --- in terms of logic gates rather than registers with their whole combinational transitive fan-ins.

The method GRAB in [19] picks the refinement variables by analyzing all shortest counter examples represented as a sequence of BDDs called synchronous onion rings (SORs). GRAB scores each design register that is an input of the abstract model based on a game-theoretic formula that estimates how many transitions within the SOR the variable can invalidate if it was under the control of player 2. The abstract model is then augmented with the variables with the highest scores until the whole set of SORs is invalidated. The controllable predecessor predicate used in our *k*-controllability analysis is similar to the game-theoretic formula of GRAB. The major differences between the two methods are: (1) a *k*-controlling variable may not get the highest score from the game-theoretic formula of GRAB and the variable getting the highest score may not be a *k*-controlling variable, (2) the *k*-controllability analysis is independent of the SORs and (3) GRAB does not try to analyze the rest of the design as what *k*-cooperativeness analysis does using 3-value simulation. We implemented this method for the experiments in Section 5.

In [1] and [2] a predicate similar to the controllability predecessor predicate is used for the purpose of compositional verification but not abstraction refinement.

Refinement is selected based on proofs rather than counter examples in [9] and [17]. The abstract models consist of the clauses that are involved in refutation proofs generated by the SAT solvers. Our method does not rely on running SAT solvers on the whole design to identify refinement candidates.

Interpolation-based reachability analysis [14] is an automatic abstraction method that is not based on abstraction refinement. It has drawn lots of interests lately because of its promising results on verifying some large designs. We compare the performance of an implementation [4] of this method in the next section.

5. EXPERIMENTAL RESULTS

We compared an implementation of our method, CNTL, against (1) RFN, an implementation of the algorithm in [20], (2) mGrab, an implementation of the GRAB algorithm in [19], and (3) INT, an implementation [4] of the interpolation method in [14]. Note that our implementations may not be as effective as the original implementations.

The comparison is based on 7 properties specified for 6 real-world designs. All properties are *True*, as we are more interested in comparing the property verification capabilities of the four methods. All experiments were done on SUN workstations with 750MHz SPARC processors and 4G-byte main memory running Solaris 5.8 OS. We set a time limit of 10 CPU hours. The experimental results are shown in Table 1.

Table 1 Runtime comparison on 7 industrial properties

	#gates /#rgstrs	CNTL	RFN	mGrab	INT
p1	481 /60	2246.3s	3826.6s	3333.9s	>10hr
p2	8372 /697	1091.9s	13555.6s	>10hr	65.1s
p3	61552 /4986	737.0s	310.2s	>10hr	194.2s
p4	77545 /2122	202.8s	1049.0s	>10hr	>10hr
p5	127229 /4891	10004.9s	10027.2s	>10hr	>10hr
p6	127261 /4895	8311.6s	10920.5s	>10hr	>10hr
p7	137365 /4494	230.3s	340.7s	>10hr	>10hr

The first column of Table 1 shows the names of the properties. The second column shows the numbers of gates and registers in the cone of influence (COI) of each property (the sequential transitive fan-in cone of the fail variable). The third through the sixth columns show the runtimes of the four methods on those properties.

On average our method CNTL is 3.4x faster than RFN. Both methods however managed to prove all properties. The mGrab program completed one proof within the time limit. We make the following observations. First, the time taken for input scoring is directly proportional to the number of inputs in the abstract model as well as the length of the SORs. Both quantities can grow rapidly as the abstract models grow. In our experiments, an abstract model with 10 registers could have 60 to 760 inputs - while the length of SORs ranged from 16 to 100. The mGrab program spent a lot of time scoring these inputs. Second, mGrab spent a significant amount of time computing the SORs.

INT is on average 10.3x faster than CNTL on the properties p2 and p3, but did not complete the proofs for the other 5 properties. Our impression is that since the interpolation method uses SAT engines that make decisions for each gate of the COI, its effectiveness tend to be impacted by the number of gates in the COI (INT did not terminate on COI with more than 70K gates). On the other hand, the complexity of the abstract models impacts the effectiveness of CNTL and RFN more than the size of the COI (only 3-value simulation and limited sequential ATPG were performed on the COI). Thus we think that the abstraction refinement methods may be more scalable than the interpolation method. INT proved the properties p2 and p3 with COI of 697 and 4986 registers respectively, which clearly outperformed any BDD-based methods. Thus we think that INT could serve as a powerful alternative proof engine for verifying the abstract model during abstraction refinement.

Table 2 shows the number of registers in the abstract models generated by the three abstraction refinement methods. The first column shows the names of the properties. The second column shows the number of registers in the abstract models generated by CNTL. The third column shows the number of *k*-controllable variables found during abstraction refinement. The fourth and fifth columns show the number of registers in the abstract models generated by RFN and mGrab respectively.

Table 2 Number of registers in the abstract models

	CNTL	*k*-controllable & *k*-cooperative	RFN	mGrab
p1	51	38	57	52
p2	62	28	75	> 66
p3	21	7	17	> 27
p4	10	10	34	> 9
p5	51	34	51	> 45
p6	54	32	60	> 50
p7	13	13	23	> 8

RFN on average uses about 49% more registers than CNTL to prove the properties, which means that CNTL provides better abstraction efficiency. In general mGrab selected fewer registers than CNTL before the time out. As mentioned above, the scoring of each input of the abstract models and the construction of the SORs bogged down the performance of mGrab.

Table 2 also shows that between 33 and 100 percent of the variables that CNTL included in the abstract models are both *k*-controllable and *k*-cooperative. The rest of the variables are *k*-

cooperative only. The abstract model that CNTL built for property p3 has the lowest percentage of *k*-controllable variables (33 percent); property p3 is also the only property that CNTL built a bigger abstract model and took more time to prove than RFN. Our conjecture is that the more *k*-controllable variables can be found, the more efficient CNTL is compared to RFN.

6. CONCLUSIONS

We introduce *k*-controllability analysis, a counter-example independent analysis that identifies variables that have the potential to invalidate all shortest counter examples; and *k*-cooperativeness analysis, a counter-example dependent analysis that identifies variables whose transition functions invalidate some counter examples. The *k*-controllability is computed by BDD operations on the abstract models and *k*-cooperativeness is computed by 3-value simulation on the whole design. We present an abstraction refinement algorithm that utilizes both *k*-controllability and *k*-cooperativeness analysis to better refine the abstract model for formally verifying safety properties.

We compared the algorithm against three automatic abstraction methods, RFN [20], GRAB [19] and the interpolation method [14], on 7 real-world properties. We found that the new method is on average 3.4x faster in runtime and 49% better in abstraction efficiency than RFN. The new method proved more properties than our implementations of the GRAB and the interpolation method. Since automatic abstraction is the key for verifying large designs, we believe that this method will make a positive impact to formal property verification.

This method may be improved by (1) more efficient *k*-controllability analysis and more accurate *k*-cooperativeness analysis and (2) using GRAB's game-theoretic formula to rank a subset of the *k*-cooperative variables in the priority queue when no *k*-controllable variable was found.

7. ACKNOWLEDGEMENTS

We thank Per Bjesse for helping us run his implementation [4] of the interpolation method and for many valuable discussions.

8. REFERENCES

[1] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Automating Modular Verification. In Proceedings of CONCUR, pp. 82-97, 1999.

[2] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Detecting Errors Before Reaching Them. In Proceedings of CAV, pp. 186-201, 2000.

[3] A. Biere, A. Cimatti, E.M. Clarke and Y. Zhu. Symbolic model checking without BDDs. In Proceedings of TACAS, pp.193-207, 1999.

[4] P. Bjesse and R. Damiano. An implementation of McMillan's interpolation algorithm, private communication and unpublished manuscript, 2003.

[5] P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith and D. Wang. Automated abstraction refinement for model checking large state space using SAT based conflict analysis. In Proceedings of FMCAD, pp.33-51, 2002.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-guided abstraction refinement. In Proceedings of CAV, pp.154-169, 2000.

[7] E.M. Clarke, A. Gupta, J. Kukula and O. Strichman. SAT based abstraction refinement using ILP and machine learning techniques. In Proceedings of CAV, pp.265-279, 2002.

[8] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer and M.Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: an industrial evaluation. In Proceedings of TACAS, pp.176-191, 2003.

[9] A. Gupta, M. Ganai, Z. Yang and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis, In Proceedings of ICCAD, pp.416-423, 2003,

[10] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In Proceedings of CAV, pp.423–427, 1996.

[11] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor and J. Long. Smart Simulation Using Collaborative Formal and Simulation Engines. In Proceedings of ICCAD, pp.120-126, 2000.

[12] O. Kupferman and M.Y. Vardi. Model checking for safety properties. Formal Methods in System Design, 19(3), pp.291-314, 2001.

[13] R.P. Kurshan. Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, 1994.

[14] K.L. McMillan. Interpolation and SAT-based model checking. In Proceedings of CAV, pp.1-13, 2003.

[15] K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. In Proceedings of CAV, pp.250-264, 2002.

[16] K.L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, 1993.

[17] K.L. McMillan and Nina Amla. Automatic abstraction without counter examples. In Proceedings of TACAS, pp.2-17, 2003.

[18] F. Somenzi. CUDD: CU Decision Diagram Package. ftp://vlsi.colorado.edu/pub/.

[19] C. Wang, B. Li, H. Jin, G.D. Hachtel, F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. In Proceedings of ICCAD, pp.408-415, 2003.

[20] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, H.-K. T. Ma and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In Proceedings of DAC, pp.35-40, 2001.