Model Reduction Based on Value Dependency

Hong Peng, Yassine Mokhtari and Sofiène Tahar
Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8 Canada
E-mail: {pengh,mokhtari,tahar}@ece.concordia.ca

Abstract— **Model checking is a fully automatic approach to verify large ASIC/SOC designs against their temporal specifications. However, its application is limited by the size of the system to be verified. Model reduction is one of the methods to tackle this problem. In this paper, we propose a model reduction approach which is based on syntactic analysis of programs. The value domains of the state variables are extracted and analyzed based on the control flow diagram using a satisfiability (SAT) solver. The approach is sound based on the fact that the reduced system simulates the concrete system with respect to the properties. We also made a case study in the paper to show its performance.**

I. Introduction

Model checking is a fully automatic approach to verify large ASIC/SOC designs against their temporal specifications. However, its application is limited by the size of the system to be verified. There are two main methods to tackle this problem: *compositional verification* and *model reduction*. With compositional verification one can verify each module in the system separately so as to avoid making the product of the modules in the system, then composes the local results into a global property. However, in today's multi-million-gate ASIC/SOC designs, the size of one single module usually is beyond the capability of a model checking tool. In this case, the only way to make model checking is to reduce the size of the module while the correctness of the properties is preserved. This is the so called model reduction approach. Model reduction is often done manually or needs human interactive. For large systems, these reduction methods are infeasible. Our focus is to propose an automatic model reduction approach, which is an improvement to existing approaches (see Section II for related work).

In the proposed approach, a system under verification is considered as a program for which abstract syntax and semantics are defined. The approach analyzes the syntactic structure, i.e., the *control flow diagram* of the program. Throughout the analysis, the dependencies of the state variables in the program are extracted. Then, using a satisfiability (SAT) solver SATO [10], we can partition the values of state variables in the program into *active values*, and *deactive values* according to their dependency to the properties. The deactive values then can be replaced by a typical *abstract* value, and thus the value domains of the variables are much smaller than the original ones. Accordingly, we can have a reduced program with respect to the abstracted variables. As we will see, after the above procedures are done, the state space of the reduced program is smaller than that of the original one, while the correctness of the properties are preserved. Throughout the analysis, we show that the approach is efficient in reducing the size of the data-path of a system with finite value domains. Because the reduction is based on the analysis of the control flow diagram, which is a canonical form of the program, it can be automatic.

The rest of paper is structured as follows. In Section II, we introduce the motivation and related work. In Section III, we define our system model including the abstract syntax (the control flow diagram) and semantics (Kripke structure) of a program. In Section IV, we describe the proposed reduction procedure. Section V is an industrial case study and Section VI finally concludes the paper.

II. Motivation and Related Work

Beyond compositional approaches, model reduction is the most important technique for reducing the state explosion problem. Model reduction is a general approach which allows to reduce a concrete system (M) under verification to a more abstract and smaller one (M'). Both systems M and M' are connected by an abstraction relation which is *safe* with respect to a given property φ, namely it preserves the property. This means if the property holds for the abstract system, it holds for the concrete one as well. More formally, the property φ is either *weakly* preserved if $M' \models \varphi \Rightarrow M \models \varphi$, or *strongly* preserved if $M' \models \varphi \equiv M \models \varphi$. It should be intuitively clear that the more weakly the property is preserved, the more reduction can be achieved.

Abstract interpretation [4] is a classical static program analysis approach. It has been used intensively in formal verification and model reduction. In [3], a program slicing approach is proposed with abstract interpretation to remove the redundant control structure and generate a reduced model for Java code. In [2], Clarke et al. proposed an abstract interpretation based approach to aggregate the values of variables into an equivalence class, then a counter-example is used to adjust the size of the equivalence class. In

[7], Kurshan proposed cone-of-influence (COI) reduction to reduce the redundant state variables. Our approach is similar with the above approaches based on the fact that we all remove the redundant state variables and aggregate the value domains, then construct a reduced model preserving the original properties. However, our approach extends the above through the fact that the abstract domain of a variable is generated throughout the analysis of the program which makes the reduction efficient and our concrete model is the program text. In [9], K. Yorav proposed ways to use the high level description (program text) of a system in order to build a reduced Kripke structure for the program, where states between two breakpoints in a path are abstracted. In a similar way, our approach uses the idea of the breakpoint but is focused on the dependency between values that influence the specification such that we can aggregate the value domain globally.

III. Preliminaries to the Reduction

In this section, we introduce an abstract program which includes a set of instructions and all the instructions in a program are in a forever loop.

An *abstract program* P is specified by a tuple (V, I, S), where

- $V = \{v_1, \ldots, v_n\}$ is a finite, non-empty set of variables. Each variable v_i, $i \in [1:n]$, has an associated finite domain of values, $dom(v_i)$.
- $I(V)$ is the initial condition, specified as assignments on V.
- S is a set of statements which are partitioned into the assignment statements S_a and the test statements S_t.

Similar to [5], a *control flow diagram* (CFD) of program $P = (V, I, S)$ is a directed diagram (N, E, L_e) where

- N is a finite set of nodes labeled by the program counter locations.
- $E \subseteq N \times N$ is a finite set of edges.
- L_e is the edge labeling function, $L_e : (E \to S)$, such that for every node n in the diagram either n is of out-degree 1 and the edge leaving n is labeled with assignment statement(s) or n is of out-degree 2 and the edges leaving n are labeled with test statement S_t and $\neg S_t$.

In the CFD, there are three special nodes, $Init$, ϵ, and ω. These nodes are part of the program syntax, where

- $Init \in N$ is the entry point of the program and marks the initialization block of the program, which has no predecessor.
- $\epsilon \in N$ is the beginning of the code. The edge to ϵ is marked by $I(V)$.
- $\omega \in N$, marks the end of the program text, where the code between ϵ and ω is in a forever loop.

Let $\overrightarrow{v} = \{v_1, \ldots, v_k\}$ be a vector of variables taking their values in a *universe* $U_{\overrightarrow{v}} = \bigcup_{i=1}^{k} dom(v_i)$. Semantically, we refer to the valuation of the variables $v_1 = e_1, \ldots, v_k = e_k$, where $e_i \in dom(v_i)$, as a *state* σ, and these variables v_i are often referred to as *state variables*. We use the notation $\sigma[\![v_i]\!]$ to denote the value of state variable v_i in state σ. Similarly, for an expression *expr* (over state variables in the domain of state σ), we use $\sigma[\![e]\!]$ to denote the value of e in the state σ. The special brackets '$[\![$' and '$]\!]$' emphasize the fact that the entity enclosed between them (v_i or e in the above cases) is of a syntactic nature.

The semantics of program P is defined in terms of *configurations*. A configuration of a flow program is a pair $c = \langle \sigma, \lambda \rangle$. The state σ is a visible component and assigns values to program variables. The *counter location* λ is a hidden component and its value corresponds to a current statement to be executed. Next, we define the particular transition relation "$\to$" for configurations.

Let $c = \langle \sigma, \lambda \rangle$ and $c' = \langle \sigma', \lambda' \rangle$ be two configurations. We write $c \to c'$ iff the following hold.

- If λ is the $Init$ node, then $\lambda' = \epsilon$ and $\sigma' = \sigma$.
- If λ is the ω node, then $\lambda' = \epsilon$ and $\sigma' = \sigma$.
- If λ is an assignment $S_a : \overrightarrow{v_1} = T(\overrightarrow{v_2})$, where $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are two vectors and T is a function from $U_{\overrightarrow{v_2}}$ into $U_{\overrightarrow{v_1}}$, then $\lambda' = suc(\lambda)$ and $\sigma'[\![\overrightarrow{v_1}]\!] = \sigma[\![T(\overrightarrow{v_2})]\!]$, where $suc(\lambda)$ means the successor of λ in CFD.
- If λ is a test statement $S_t : if(B(\overrightarrow{v}))$ or $while(B(\overrightarrow{v}))$, where B is a boolean function on $\overrightarrow{v}$, then $\sigma' = \sigma$, and either $\lambda' = suc^{+}(\lambda)$, in the case $\sigma[\![B(\overrightarrow{v})]\!]$ is true or $\lambda' = suc^{-}(\lambda)$ in the case $\sigma[\![\neg B(\overrightarrow{v})]\!]$ is true, where $suc^{+}(\lambda)$ and $suc^{-}(\lambda)$ mean the true or false successor of λ in a CFD, respectively.

For any natural number n, the composition of n steps of $\to$ is $\to^n$. We will also use $\to^*$ to denote the reflexive transitive closure of $\to$.

The semantics of the program $P = (V, I, S)$ is a Kripke structure $\mathcal{K}(\mathcal{P}) = (\Sigma, \langle \sigma_0, Init \rangle, \to, L)$ given by a set of configurations Σ such that $\Sigma = \{\langle \sigma', \epsilon \rangle : \langle \sigma, \epsilon \rangle \to^* \langle \sigma', \epsilon \rangle\} \cup \{\langle \sigma_0, \epsilon \rangle : \langle \sigma_0, Init \rangle \to \langle \sigma_0, \epsilon \rangle\}$, an initial configuration $\langle \sigma_0, Init \rangle \in \Sigma$ such that $\sigma[\![I(V)]\!]$ is true, a transition relation $\to$ and a labeling function $L : \Sigma \to 2^{AP}$ that associates a non-empty set of atomic propositions with every configuration $c \in \Sigma$.

Example 1. To illustrate the above definitions, we have chosen a small example. In Figure 1, we give the CFD of a program P and its semantics. In the CFD, nodes are marked by the labels $(\epsilon, L_0, L_1, L_2, \omega)$ and edges are marked by the statements in the program [1]. In the Kripke structure, states are marked by the valuation of the variables

[1] In the figure, "epsilon" means ϵ and "omega" means ω.

and the double circle is the initial state. Here, since in the Kripke structure, the value of λ is always ϵ, we omit ϵ in the label of the configurations, namely $<(0,0,0), \lambda>$ becomes $(0,0,0)$.

```
//Program P
i, j, k : {0,1,2,3};
Init:    i = 0;
         j = 0;
         k = 0;
epsilon: i = (i + 1) mod 4;
L0:      j = (j + 1) mod 4;
L1:      if (i == 3)
L2:        k = 1;
omega:
```

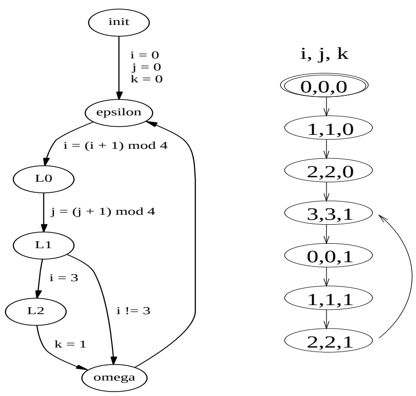

Fig. 1. Control Flow Diagram of P and its Kripke structure

IV. Proposed Model Reduction Approach

In the suggested model reduction approach, we first partition the domain values of variables by using the reachability condition (RC) and state transformation (ST) functions. Next, we define how the abstract program is constructed. Finally, we prove the soundness of our approach.

Definition 1. Let $\tau = N_0 \overset{E_1}{\rightarrow} \ldots \overset{E_k}{\rightarrow} N_k$ be a finite path in CFD of length $k+1$, for some natural number k. We first define $RC_\tau^m(\overrightarrow{v})$ and $ST_\tau^m(\overrightarrow{v})$, the corresponding characteristics of the suffix $N_m \overset{E_{m+1}}{\rightarrow} \ldots \overset{E_k}{\rightarrow} N_k$ of τ, by induction on m going down from k to 0. We use the notation $a[v \leftarrow e]$ to denote the substitution of every occurrence of v by the value e in the expression a.

• *Induction basis*
$RC_\tau^k(\overrightarrow{v}) = true$, $ST_\tau^k(\overrightarrow{v}) = \overrightarrow{v}$
In other words, being at the end of τ to traverse τ implies an identically true reachability condition and an identity state transformation.

• *Induction assumptions*
Suppose that $RC_\tau^{m+1}(\overrightarrow{v})$ and $ST_\tau^{m+1}(\overrightarrow{v})$ have already been defined for $0 \leq m < k$

• *Induction step*
We define $RC_\tau^m(\overrightarrow{v})$ and $ST_\tau^m(\overrightarrow{v})$ according to the statements at edge E_m.
– *Initialization*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v}))[\overrightarrow{v} \leftarrow I(V)]$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})[\overrightarrow{v} \leftarrow I(V)]$. Thus, the initial statement is captured by an assignment of new values to the variables.
– *Assignment*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v}))[\overrightarrow{v} \leftarrow T(V)]$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})[\overrightarrow{v} \leftarrow T(V)]$. Thus, the effect of an assignment is captured by substitution.
– *Positive test*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v})) \wedge B(\overrightarrow{v})$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})$. Thus a test does not change the state and remembers the condition.
– *Negative test*: $RC_\tau^m(\overrightarrow{v}) = (RC_\tau^{m+1}(\overrightarrow{v})) \wedge \neg B(\overrightarrow{v})$, $ST_\tau^m(\overrightarrow{v}) = ST_\tau^{m+1}(\overrightarrow{v})$. This is similar to the positive test case.
Finally, we define $RC_\tau^0(\overrightarrow{v}) = RC_\tau(\overrightarrow{v})$, and $ST_\tau^0(\overrightarrow{v}) = ST_\tau(\overrightarrow{v})$.

Let v be the variable of interest in the property, n be a node in CFD and e be the outgoing edge of n. Similar to the breakpoints in [9], We say that n is a *key node* iff $L_e(e) \triangleq v = T(\overrightarrow{v})$. Next, we compute the reachability condition and state transformation function of the path starting from the ϵ node and ending on a key node.

Let v be a variable among the dependency list of interest variables, e the value of v and τ be a path leading to a key node.
We say that the value e is *active* iff $RC_\tau[v \leftarrow e]$ is true or $ST_\tau(v) = e$. Similarity, we say that the value e is *deactive* iff $RC_\tau[v \leftarrow e]$ is false.

A practical problem in using the above concepts is how to find all the true value assignments for the reachability condition. Since SAT solvers [6] are good at finding variable assignments which make the conjunctive normal form (CNF) propositional formula true, we propose to use a SAT solver, here SATO [10], to get all the values satisfying formula $RC_\tau[v \leftarrow e] =$ **true**, which are the *active* values with respect to the RC. Generally and during computations, a variable v may have both active and deactive values. Hence, the domain values of v, i.e., $dom(v)$, is partitioned into disjoint subsets ACTIVE (v), eventually empty, of active values and a singleton DEACTIVE (v), eventually empty, of an (abstract) deactive value. However, since SAT-solvers expect their input to be a propositional formula in CNF form, we still need to convert the reachability condition into CNF. Besides, when we do the conversion, since SATO does pure literal simplification [10], we have to keep a literal and its opposite in the CNF. Hence, the abstract program is constructed as follows.

$P = (V, I, S)$ is an abstract program. The reduced program $\widehat{P} = (\widehat{V}, \widehat{I}, \widehat{S})$ is:

• $\widehat{V} = V$ is a finite, non-empty set of variables. For each $v_i \in \widehat{V}$, $dom(v_i) =$ACTIVE $(v_i) \cup$ DEACTIVE (v_i).
• $\widehat{I}(\widehat{V})$ is equal to $I(V)[e_i \leftarrow$DEACTIVE $(v_i)]$ for each v_i such that $e_i \notin$ACTIVE (v_i).
• $\widehat{S}$ is obtained as follows: (1) if or $while$ $B(\overrightarrow{v})$ statements are obtained like the initial condition. (2) An abstract assignment statement $\widehat{S_a}$ is defined nondeterministically by $v_i =$ACTIVE $(v_i) \cup$ DEACTIVE (v_i).

Also, model $\mathcal{K}(\mathcal{P})$ is simulated by model $\mathcal{K}(\widehat{\mathcal{P}})$, namely $\mathcal{K}(\mathcal{P}) \preceq \mathcal{K}(\widehat{\mathcal{P}})$. Finally, since the reduced program simulates the original program, the **ACTL** properties are preserved [2]. Namely, the reduction is conservative.

Example 2. Considering the program P of Example 1, we want to verify a property about variable k. We can first apply COI reduction to find that variable j is irrelevant and thus can be removed. We will apply our approach based on the COI reduced program.

In the CFD, node L_2 is the key node since the statement $k = 1$ changes the value of k. Consider the value 0 of i. i is 0 at node ϵ. Through the backward reachability analysis from key node L_2 to ϵ, we find that $RC_{\epsilon \to L_2}$ =“$i = 3$”. According to Definition 3 and using SATO, $RC_{\epsilon \to L_2}[i \leftarrow 0] =$ “ $0 = 3$” which is false. So, the value 0 of i is a deactive value. In a similar way, we find that value 1 and 2 do not satisfy RC, so 1 and 2 are deactive values. The value 3 of i is active since L_2 is reachable in path ϵ, L_1, L_2 ($RC_{\epsilon \to L_2}[i \leftarrow 3] = true$) while $ST_{\epsilon \to L_2}(i) = 3$. Hence, we can use abstract value domain $\{2, 3\}$ instead of its concrete value domain $\{0, 1, 2, 3\}$. Using the same approach, the abstract value domain of variable k is $\{0, 1\}$.

```
//Reduced program P̂

i : {2,3};
k : {0,1};
// Initialization
Init:    i = 2;
         k = 0;
epsilon: i = ND(2,3);
L1:      if (i == 3)
L2:        k = 1;
omega:
```

In Figure 2, the reduced program $\widehat{P}$ of P is shown on the left side, where “ND” means nondeterministic assignment. In the middle, the state space of the program P after COI reduction has 7 states. On the right side, the reduced state space of $\widehat{P}$ has three states.

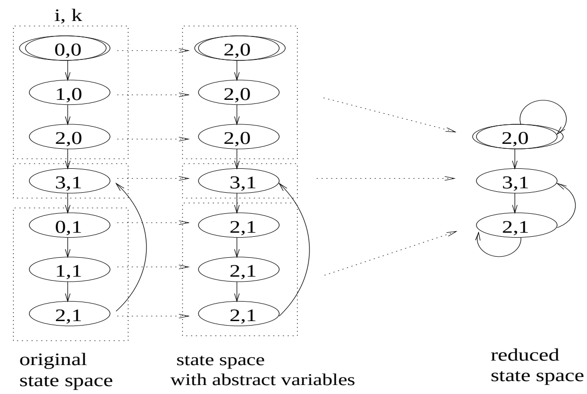

Fig. 2. Kripke structure of Reduced Program $\widehat{P}$

V. Case study: A Forwarding Table Lookup Processor

With the expansion of communication networks, fast and intelligent routers are required. Usually, these routers perform multiple lookups per packet into potentially large tables. For instance, the router of an industrial ATM (Asynchronous Transfer Mode) switch fabric translates the incoming identifier (8-bit $Item\_In1$) into the destination identifier according to the routing table $Lookup\_Table$. Only those $Item\_In1$ inputs which can find a match ($Match\_Found = 1$) in the routing table can be switched. Due to space limitation, we consider a *simplified* version of the original program.

```
module search;
 reg [7 : 0] Lookup_Table [9 : 0];
 reg Match_Found;
 integer i, Error_Count;
 reg [7:0] Item_in1;
Init:
 initialization  i=0; Match_Found = 0;
 Error_Count = 0;
epsilon: while (i <= 9) begin
L0:       if (Lookup_Table[i][7:0] == Item_In1) begin
L1:           Match_Found = 1;
L2:           i=9;
           endif
L3:        i ++;
           endwhile
L4:       if (!Match_Found)
L5:           Error_Count = Error_Count + 1;
          else
L6:           Error_Count = Error_Count - 1;
          endif
omega:
```

Assuming The values of the $Lookup\_Table$ are $\{00, 01, \ldots, 09\}$, We want to prove that “the input $Item\_in = 02$ can find a match in the table such that $Match\_Found = 1$”. The correspondent properties are φ_1 : $\mathbf{AF}((Item\_In1 = 2) \wedge (Match\_Found = 1))$ and φ_2 : $\mathbf{AG}((\neg(Item\_In1 = 2)) \rightarrow (Match\_Found = 0))$ meaning that $(Item\_In1 = 2)$ eventually can find the match and the others cannot. These properties could not be verified on the original model (with 1024 items) due to state space explosion during the verification in VIS [1] on a SUN Enterprise server with 6GB memory after more than 10 hours running. After reduction, we removed 8 items from the table, and get a reduced program. This reduced program could be model checked in VIS on the same machine. The experimental results are shown in Table I including statistic results of CPU time (in seconds) and memory usage (in bytes) of VIS verification on the reduced model, and that of SAT solver on RC_τ computation, respectively.

	Model Checking		RC_τ Computation	
Prop.	Verif. CPU	Verif. Mem	SAT CPU	SAT Mem
φ_1	11.3	8.54M	0.02	83.53K
φ_2	11.6	88.51M	0.02	83.53K

TABLE I
STATISTIC RESULT OF SAMPLE PROPERTIES

In addition to the above properties, we also verified a number of other properties (ψ_i in Table II) on the *whole* ATM switch fabric. The results are shown in Table II (a detailed description of these properties can be found in [8]). Throughout the reduction, we could verify the properties in reasonable CPU time and memory space while none ever terminates on the concrete model.

	Model Checking		RC_τ Computation	
Prop.	Verif. CPU	Verif. Mem	SAT CPU	SAT Mem
ψ_1	19.5	0.908M	0.01	83.56K
ψ_2	272.4	1.908M	0.02	83.53K
ψ_3	1.4	1.308M	0.01	32K
ψ_4	3.8	9.9M	0.02	85.90K
ψ_5	3.7	9.918M	0.02	83.53K
ψ_6	2.5	1.44M	0.01	84.55K
ψ_7	6.7	12.2M	0.02	87.24K

TABLE II
STATISTIC RESULT OF OTHER PROPERTIES

VI. CONCLUSIONS

In this paper, we proposed a model reduction approach which uses syntactic analysis and generates a smaller program compared to the original one. It preserves the properties to be verified. This model reduction approach can be used together with other state space explosion solution: compositional verification, where the size of single modules of a large ASIC/SOC system is beyond the capability of model checking. A SAT solver is used to find the true value assignment throughout the reduction.

In our approach, the solution is based on the syntactic analysis of the program source code which can be combined with other model checking or reduction tools to improve its reduction efficiency even more. Moreover, since the size of the source code is usually much less than that of its state space or other intermediate forms, this approach uses less resources (CPU time and memory space). Another advantage is that the approach can work directly on hardware description language such as Verilog via the syntactic and semantic analysis of Verilog code and this can be a feature of a Verilog compiler. The complexity of the reduction depends on that of the CFD traverse and the SAT solver. One of our future research directions is to explore an efficient SAT decision procedure to improve the efficiency. We are currently working on the implementation of the algorithms.

REFERENCES

[1] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Proceedings of Computer-aided Verification'96*, volume 1102 of *LNCS*, pages 428–432. Springer Verlag, Rutgers University, NY, USA, July 1996.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counter-example guided abstraction refinement. In *Procceding of Computer Aided Verification'00*, volume 1855 of *LNCS*, pages 154–169, Chicago, IL, USA, July 2000.

[3] J. Corbett, M. Dwyer, J. Hatcliff, and et. al. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, 1977.

[5] N. Francez. *Program Verification.* Addison-Wesley, 1992.

[6] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for satisfiability (SAT) problem: A survey. In *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem*, volume 35, pages 19–152. American Mathematical Society, 1997.

[7] R. P. Kurshan. *Computer-aided verification of coordinating processes.* Princeton University Press, 1994.

[8] H. Peng, S. Tahar, and Y. Mokhtari. Compositional verification: Case study on a Nortel ATM switch fabric. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, March 2001.

[9] K. Yorav. *Exploiting syntactic structure for automatic verification.* PhD thesis, Israel institute of technology, June 2000.

[10] H. Zhang. SATO: An efficient propositional prover. In *Proceeding of International Conference on Automated Deduction (CADE-97)*, volume 1249 of *LNAI*, pages 272–275, Berlin, German, July 1997.