Electronic Preprint for Journal of Information Processing Vol.25

Regular Paper

Exploiting Functional Dependencies of Variables in All Solutions SAT Solvers

Takahisa Toda[1,a)] Takeru Inoue[2,b)]

Received: November 7, 2016, Accepted: April 10, 2017

Abstract: All solutions SAT (AllSAT) is the problem of generating satisfying assignments to a given conjunctive normal form (CNF) and has been a key issue commonly found in several applications of formal verification including model checking. CNF encoding, which translates original problems for AllSAT solvers, spawns many auxiliary variables and, what is worse, obscures functional dependencies over variables. AllSAT solvers consequently have to deal with unnecessarily larger CNFs, although the original problems might be much more tractable in essence. This paper proposes a novel AllSAT solver along with a CNF encoding technique; our solver extracts functional dependencies through the encoding process, and the dependence is effectively utilized to solve the CNF. Our solver is designed based on the OBDD compilation technique, which allows us to efficiently handle intractable CNFs with a number of solutions in dynamic programming manner. Our proposal is very simple but powerful; experiments with real network instances showed that our solver exhibits a great improvement.

Keywords: AllSAT, model enumeration, OBDD compiler, functional dependency, independent variable

1. Introduction

All solutions SAT (*AllSAT*) is the problem of generating satisfying assignments to a conjunctive normal form (CNF) such that they form a logically equivalent disjunctive normal form (DNF). AllSAT has many applications in the field of formal verification: e.g., model checking [6], [8], [16], predicate abstraction [5], and network verification [14], [15], [19]. Since problems in those applications are described in a domain-specific manner, they must be transformed to CNFs in order to employ an efficient AllSAT solver. This translation is usually performed in two steps: original problems are represented in propositional Boolean formulae (*the modeling part*), which are then encoded into CNFs (*the encoding part*). The modeling part is domain-specific and is beyond the scope of this paper, while the encoding part can be developed commonly.

The encoding part obscures functional dependencies over variables, which can be a clue to solve CNFs efficiently. Here, a *functional dependency* means that the value of a variable is determined only by the values of some other variables regardless of assignments, as illustrated below (also see Definition 2).

Example 1. *Consider a CNF, ψ, defined as:*

$$(\neg x \vee y) \wedge (\neg x \vee z) \wedge (x \vee \neg y \vee \neg z).$$

Since ψ is logically equivalent to $x \leftrightarrow y \wedge z$, the value of x is uniquely determined according to the values of y and z so that ψ is satisfiable.

[1] Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[2] NTT Network Innovation Laboratories, NTT Corporation, Yokosuka, Kanagawa 239–0847, Japan
[a)] todat@acm.org
[b)] inoue.takeru@lab.ntt.co.jp

Dependent variables are found in many applications; e.g., the transition relation in model checking is a conjunction of next state variables, v'_i, defined in terms of current state variables as $\bigwedge_i v'_i \leftrightarrow \psi_i(s)$ [1]. Dependent variables are also spawned as *auxiliary variables* through the modeling part; e.g., in data mining [7], the value of an auxiliary variable, z, is bound to the evaluation of a constraint, C, so that $z \leftrightarrow C$ holds. Auxiliary variables are introduced in the encoding part as well. They could make encoded CNFs much longer, even though their values are not of interest. As implied by the above observation, encoded CNFs are often much more complex than the original problems due to dependent variables.

Grumberg et al. [6] presented an AllSAT solver that focused on *important* variables, i.e., variables whose values are of interest. Given a CNF formula and dependence information about which variables are important, the solver returns all assignments for the CNF such that all important variables are assigned values and the assignments can be extended to complete solutions. Important variables must be identified outside the solver, and their work did not address how to identify them. Their solver is promising as Toda and Soh confirmed in their experiments [17]. However, since the solver finds solutions one at a time, the computation time has to depend on the number of solutions. Hence, there is a limit to the number of solutions to be generated within a realistic amount of time.

The *compilation-based AllSAT solver*, which was proposed by Huang and Darwiche [9], addresses this issue. This solver identifies equivalent subproblems in its search process and does not compute them more than once in dynamic programming manner. Thus, the computation time does not directly depend on the number of solutions. The compilation-based solver, however, deals with all variables equally and does not exploit functional depen-

dencies. Unfortunately, *ordered binary decision diagrams* (*OBDDs*), which are a data structure used to represent solutions in this solver, can be blown up with the number of variables, and this solver does not work efficiently with many dependent variables. It is not straightforward for the compilation-based AllSAT solver to efficiently utilize functional dependencies because the construction process of OBDD is tightly coupled with all the CNF variables.

This paper proposes a novel compilation-based AllSAT solver that exploits functional dependencies over variables. Our AllSAT solver is integrated with the encoding part to resolve the above issue of a compilation-based solver; given a propositional Boolean formula, our solver encodes it into a CNF while extracting functional dependencies, which is then effectively used to construct an OBDD of important variables only, where we consider variables other than extracted dependent variables important. Experiments are conducted with a real network dataset and a common model checking dataset. It turns out that CNF instances encoded from the network dataset have a large number of solutions, and compilation-based solvers are suitable for this kind of instances: they find more solutions compared to Grumberg's solver by a few orders of magnitude. Our proposed technique accelerates the original compilation-based solver further.

The paper is organized as follows. Section 2 provides necessary notions concerning binary decision diagrams, and presents the algorithm for compilation-based solvers. Section 3 proposes a novel compilation-based solver that exploits functional dependencies and Section 4 presents the experimental results. Section 5 concludes the paper.

2. Preliminaries

2.1 Binary Decision Diagrams

Binary decision diagrams (*BDDs*) are a graphical representation of Boolean functions [4]. **Figure 1** depicts an example of a BDD. Exactly one node has indegree 0, which is called the *root*. Each *branch node*, f, has a label and two children. Node labels are taken from the indices of Boolean variables. A child pointed to by a dotted arrow is called a *LO child* and a child pointed to by a solid arrow is called a *HI child*. The arc to a LO child is called a LO *arc*, and the LO arc of f means the value, 0, is assigned to the variable of f. Similarly, the HI arc means 1 is assigned to its variable. There are two *sink nodes* denoted by $\top$ and $\bot$. Paths from the root to $\top$ and $\bot$ respectively correspond to satisfying assignments and falsifying assignments. The values of variables skipped on the path are "don't care." Common prefix and suffix can be shared among paths. BDDs are *ordered* if for any node, u, with a branch node, v, as its child, the label of u is less than the label of v. In this paper, ordered BDDs (*OBDDs* for short) are not necessarily reduced [4]. Here, we remark that *each node in an OBDD is conventionally identified with the subgraph rooted by that node, which also forms an OBDD.*

2.2 Compilation-based AllSAT Solver

Algorithm 1 presents a pseudocode of compilation-based AllSAT solver [9], [17]. The algorithm integrates DPLL search and OBDD compilation, which are interleaved in the code as described below. We begin with the DPLL search which includes the three stages: decision (lines 20–22), deduction (line 5), and diagnosis (lines 8 and 10). *The procedures concerning OBDD compilation are skipped for the time being.* The DPLL search is a backtracking-based algorithm, and it searches a satisfying assignment in such a way that a solver extends a candidate assignment by assigning values to variables and if the assignment turns out to be not satisfying, the solver proceeds to the next candidate by canceling the values of some variables.

Algorithm 1: OBDD compiler on DPLL. Chronological backtracking is used for conflict resolution. k_{sat} is key computed when all variables are assigned values.

Input: a CNF formula ψ, an empty assignment ν.
Output: the OBDD for all satisfying assignments of ψ.

```
1  dl ← 0;                                              // Decision level
2  f ← ⊥;                                                        // OBDD
3  S ← {(k_sat, ⊤)};                          // Set of key-result pairs
4  while true do
5      ν ← propagate (ψ, ν);                              // Deduction stage
6      if a conflict occurs then
7          if dl ≤ 0 then return f;
8          ψ ← analyze (ψ, ν);                            // Diagnosis stage
9          S ← insertcache (dl, ν, S);
10         (ν, dl) ← backtrack (ψ, ν, dl);
11     else
12         i ← min{j | x_j is not assigned value};
13         key ← computekey (ψ, ν, i − 1);
14         if there exists a pair (key, result) in S then
15             f ← extendobdd (f, result, ν);
16             if dl ≤ 0 then return f;
17             S ← insertcache (dl, ν, S);
18             (ν, dl) ← backtrack (ψ, ν, dl);
19         else
20             dl ← dl + 1;                               // Decision stage
21             select a value v;
22             ν ← ν ∪ {(x_i, v)};
23         end
24     end
25 end
```

We will see each stage of DPLL search below. Let ψ be an input CNF formula. Let $x_1, \ldots, x_n$ be the variables that occur in ψ, which are selected in the fixed order (line 12). We begin with the decision stage (lines 20–22). A value, v, is selected, and the least unassigned variable, x_i, is assigned value v. The variable, x_i, is called a *decision variable*, and the assignment, (x_i, v), is called a *decision assignment* or simply a *decision*. The variable, dl, in the algorithm holds the number of decisions that have been made, which is called a *decision level*. The current decision level is incremented at line 20.

At the deduction stage (line 5), all implications are deduced from the current assignment, ν. The implications mean assignments to other unassigned variables that are uniquely determined by the recent assignment. An implication occurs if there is a clause, $C = \{l_1, \ldots, l_k\}$, in ψ such that all but one of the literals, say l_1 to l_{k-1}, are evaluated to 0 in the current assignment, ν, and the remaining literal, l_k, is not evaluated to 0 or 1. Clearly, l_k must be evaluated to 1 in order that ψ is satisfiable. In this case, C is called a *unit clause*, l_k a *unit literal*, the underlying variable of

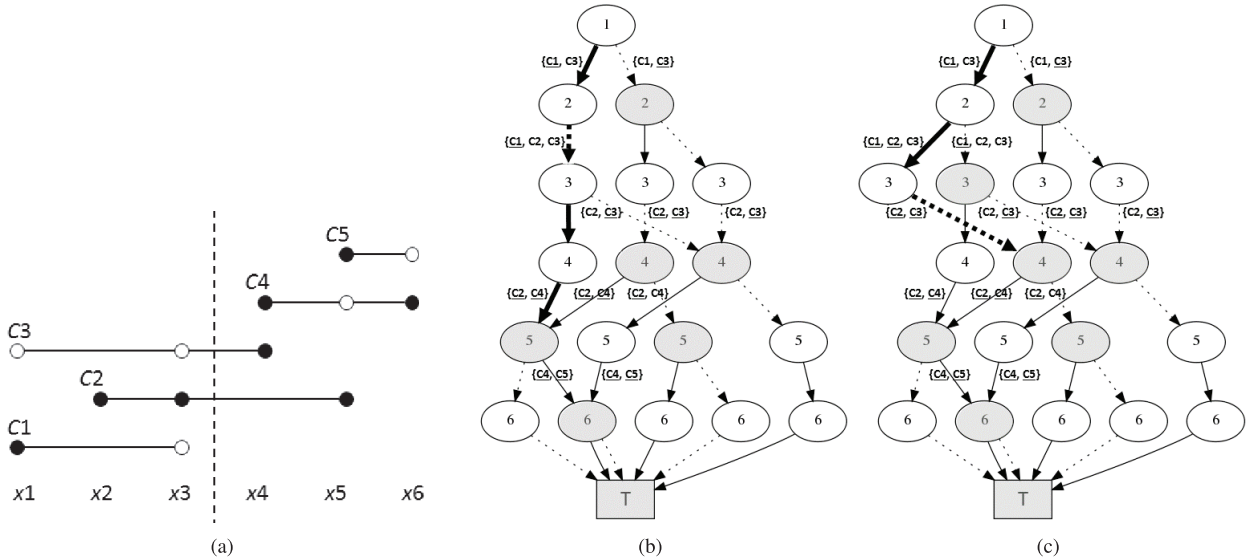

Fig. 1 (a) Illustration of CNF. Intervals represent clauses, bullets represents positive literals, and circles represent negative literals. A dashed line indicates the 3-rd cutset $\{C_2, C_3\}$. (b)–(c) Construction of OBDD from CNF. Computed states of cutsets are associated with arcs, where underlined clauses mean that they are satisfied. Gray nodes are those recorded as key-result pairs. Thick arcs in each OBDD represent the path that is going to be added.

Algorithm 2: Function, *backtrack*, that performs chronological backtracking in OBDD compiler on DPLL. The $\delta(y)$ denotes the decision levels at which k is made and $\nu(y)$ is defined.

Input: a decision level dl, an assignment ν.
Output: the updated objects ν, dl.
1 $(x, v) \leftarrow$ the decision of level dl;
2 $\nu \leftarrow \{(y, w) \in \nu \mid \delta(y) < dl\}$;
3 $dl \leftarrow dl - 1$;
4 $\nu \leftarrow \nu \cup \{(x, \bar{v})\}$, where $\bar{v}$ is the opposite value from v;

l_k an *implied variable*, and its assignment an *implied assignment*. The function, *propagate*, computes all implied assignments until there is no unit clause or there is a falsified clause. This process is called *unit propagation*. If unit propagation is terminated because of the presence of a falsified clause, this case is called a *conflict*.

If a conflict occurs, then the diagnosis stage (lines 8–10) is executed. The function, *analyze*, computes a *conflict clause* at this stage, which is a clause that is falsified by the current assignment, and the conflict clause is added to ψ, by which a solver is guided not to fall into the same conflict or relevant conflicts.

Algorithm 2 presents the pseudocode of the function, *backtrack*, used in Algorithm 1. This algorithm, for simplicity, adopts *chronological backtracking*, although our algorithm can be integrated with any backtracking method presented in Ref. [17]. This algorithm cancels all assignments of the current decision level, dl, in the current assignment, ν (line 2), decreases dl by one (line 3), and inserts a flipped decision, $(x, \bar{v})$, into ν as a non-decision assignment. Because of the presence of the flipped decision, $(x, \bar{v})$, solutions that include all decisions before backtracking occurred are blocked.

We will now shift our attention to the OBDD compilation part, which consists of the functions *computekey*, *extendobdd*, and *insertcache*. We first introduce terminology. Let ν be an assignment such that the i-th variable is the least unassigned variable. The *subinstance* in ν is the CNF induced from the original CNF, ψ, by applying the assignments up to x_{i-1} in ν. The *solution space* in ν consists of all solutions of the subinstance in ν.

The basic idea in the OBDD compilation part is to avoid the recomputation of "equivalent" subinstances by using a similar technique to dynamic programming. More concretely, since different assignments can induce subinstances with the same solution space, if all solutions of one subinstance are exhausted and they are stored in some data structure, then an exhaustive search for other equivalent subinstances do not need to be done because all solutions have already been computed.

There are two keys to realizing this idea. First, an OBDD is used as a data structure for solutions in such a way that if a solution is found, a new path corresponding to the solution is added to an OBDD. Second, whether there is a past subinstance such that it has the same solution space as the current subinstance is decided using the concept of *cutsets* we are about to introduce.

Definition 1. *Let ψ be a CNF, and let V be the set of variables in ψ. The i-th* cutset *of ψ is defined as*

$$\{C \in \psi \mid \exists l, l' \in C,\ \mathrm{id}(l) \le i < \mathrm{id}(l')\},$$

where $\mathrm{id}(l)$ *denotes the index of the underlying variable of l (see Fig. 1 (a)). The* state *of the i-th cutset in an assignment ν is a binary sequence with each entry being* 1 *if the corresponding clause in the cutset is satisfied by ν and being* 0 *otherwise. We here assume without loss of generality that the clauses in a cutset is ordered in an arbitrary fixed order so that each clause corresponds to an entry in the sequence.*

The following proposition declares that the states of cutsets provide a sound equivalence test between subinstances.

Proposition 1. *Let ψ be a CNF. Let ν and μ be assignments such that all of the i-th and less variables are assigned values. Assume that all clauses consisting only some of the $(i-2)$-nd and less*

variables are satisfied. If the state of the $(i-1)$*-st cutset in* ν *is identical to the state in* μ*, then the subinstances in* ν *and* μ *are logically equivalent.*

We are now ready to explain the pseudocode of the OBDD compilation part. Let us start with line 12 of Algorithm 1. The variable, i, holds the least index of an unassigned variable.

The function, *computekey*, is a generic function that computes some data *key*, called a *state*, that characterizes the subinstance induced by ν from an input CNF, ψ. Since states are used for deciding the equivalence of subinstances, they must provide a sound test, that is, they must ensure that if subinstances are not equivalent, then the test result must be negative. Examples of such a test include cutsets, separators [9], and a variant of cutsets [18].

We decide at line 14 whether there is a past subinstance such that it is encoded into the same state, *key* (thereby, it has the same solution space), as the current state. To do this, it suffices to simply search a pair in S with the first element, *key*, because the function, *insertcache*, maintains S so that if the current state is computed and all solutions of the current subinstance are computed as an OBDD, then the pair of that state and the root of that OBDD is inserted into S. As illustrated in Figs. 1 (b) and 1 (c), if the search succeeds [*1] and *result* holds the associated node, then we extend the current OBDD, f, by adding a new path corresponding to the current assignment, ν, so that it is connected to *result*. Since the OBDD rooted by *result* represents all solutions of the current subinstance, connecting the path of ν to *result* means registering all solutions extending ν into f. The function, *extendobdd*, performs this operation (line 15). The following expression represents how an OBDD is extended in terms of Boolean functions.

$$F_f := F_f \vee F_\nu \wedge F_{result},$$

where F_x denotes the Boolean function for x.

All that remains is to explain how to register key-result pairs into S, which is done by the function *insertcache*. This function is called whenever backtracking is performed. Since chronological backtracking cancels assignments of the highest decision level, when backtracking is performed, the subinstances induced by such assignments must be solved. The function *insertcache* thus registers all key-result pairs (*key*, *result*) such that *key* is the state made by *computekey* with such an assignment and *result* is the corresponding OBDD node, which is present in the path most recently added to an OBDD.

3. AllSAT Procedure Using Variable Dependence

This section presents an AllSAT-solving method that, given a propositional Boolean formula, extracts functional dependencies from the formula, and constructs the OBDD for satisfying assignments over important variables only, i.e., variables other than extracted dependent variables. The formal definition of functional dependencies is:

Definition 2. *Let* ψ *be a CNF and let* V *be the set of all variables in* ψ*. A variable,* $x \in V$*, is* dependent *in* ψ *if there is a nonempty subset,* S*, of* $V \setminus \{x\}$ *such that for any two total assignments* $\nu, \mu\colon V \to \{0,1\}$ *that make* ψ *evaluate to* 1*, if* $\nu(y) = \mu(y)$ *holds for all* $y \in S$*, then* $\nu(x) = \mu(x)$ *holds. In this case,* x *is* dominated *by* S*. This dependence relation is denoted by* $x \twoheadleftarrow S$*.*

We clearly distinguish the fact that $x \twoheadleftarrow S$ holds from the fact that the relation is identified in some way. Since identifying dependent variables is computationally intractable, we only compute functional dependencies that can easily be determined. We will simply call determined dependent variables *non-important variables* and the other variables *important variables*.

Our method consists of the stage of extracting functional dependencies, the stage of determining a static variable order, and the stage of exploiting functional dependencies inside compilation-based solvers.

3.1 Extracting Variable Dependence

Tseitin encodings are widely used for encoding propositional formulae into CNFs [2]. Given a propositional formula, ϕ, the Tseitin encodings introduce a new variable, x, for each subformula, α, that constitutes ϕ and generate clauses that represent the logical relation, $x \leftrightarrow \alpha$ [*2].

We present a method of extracting functional dependencies while performing the Tseitin encodings. We denote, by λ, the mapping from each subformula to the variable introduced by the Tseitin encodings. Let α be a subformula of ϕ. If α has form $\alpha_1 \circ \alpha_2$ for some binary operator $\circ$, then we extract the relation, $\lambda(\alpha) \twoheadleftarrow \{\lambda(\alpha_1), \lambda(\alpha_2)\}$. In particular, if α has form, $z \leftrightarrow \alpha_1$ or $\alpha_1 \leftrightarrow z$, where z is a variable and α_1 is not a variable, then we in addition extract the relation $\lambda(z) \twoheadleftarrow \lambda(\alpha_1)$. If α has form $\neg\alpha_1$, then we extract the relation, $\lambda(\alpha) \twoheadleftarrow \lambda(\alpha_1)$.

As was explained in Section 1, propositional formulae tend to include many subformulae of form $z \leftrightarrow \alpha$ for some variable z and subformula α. These kinds of subformulae refer to functional dependencies between original variables. Our AllSAT-solving method can make better use of such background knowledge if we consider not only constraints that are necessary for modeling, but also extra constraints that can be obtained by using some heuristic or those added manually according to domain-specific knowledge in the modeling phase.

3.2 Determining Variable Order

Here, we present a method of determining a static order over all variables in a CNF to make use of functional dependencies in compilation-based solvers. Note that compilation-based solvers do not dynamically change the order of variables due to the restrictions of OBDD.

Let us first consider a variable order, $<$, with the property (P): if a variable, x, is dominated in some extracted functional dependency, then there is a set of variables, S, such that $x \twoheadleftarrow S$ and $y < x$ hold for all $y \in S$. This property suggests that the

[*1] In the case that no variable is assigned value, i.e., $i = 1$, let *computekey* return the undefined value, k_{undef}, so that the search fails.

[*2] When satisfiability is all that matters, the subformula α can be replaced with a new variable x by adding the new constraint $x \to \alpha$ instead of the bidirectional implication $x \leftrightarrow \alpha$ when a given propositional formula is in negative normal form. In this case, x is not a depending variable. When it comes to computing all solutions, we need bidirectional implications, which ensure a one-to-one correspondence between satisfying assignments for the original formula and those for the encoded CNF.

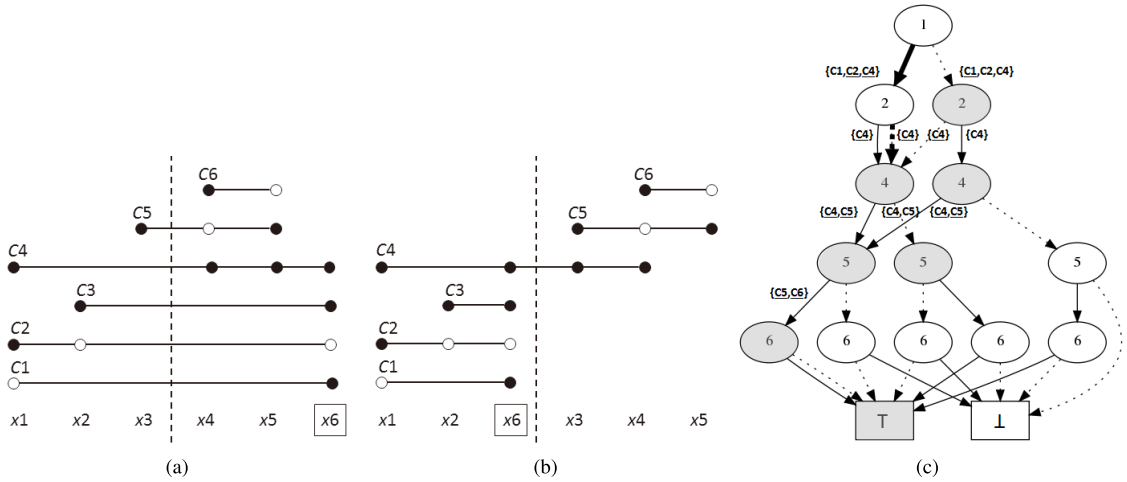

Fig. 2 (a) CNF in original variable order, where x_6 is dominated by x_1 and x_2. (b) CNF in order determined by our method. (c) Constructed OBDD with our method. Nodes of label 3 are eliminated because 3-rd variable, i.e., x_6, is dominated. Cutsets are used for equivalence test. Node labels mean a changed order of variables.

value of x is implied regardless of assignments and search cannot branch at x (see Corollary 1). One simple instance of such an ordering is to select important variables first, which is adopted in the solver of Grumberg et al. and called *important first decision procedure* [6]. This ordering, however, is not suitable for compilation-based solvers using cutsets or their variants because, as is illustrated in **Fig. 2** (a), if clauses contain both independent and dependent variables, their interval representations lengthen horizontally, and hence those clauses occur in many cutsets, by which it is more prone to fail in identifying equivalent subinstances.

The basic idea of our variable ordering is to select dependent variables as early as possible, while maintaining the property (P). To achieve this, we sort variables in a CNF by moving dependent variables to earlier positions. To do this, we introduce, for each variable x, the set, $D(x)$, of candidates for a variable, y, such that x is moved after y.

Definition 3. *Let x be a variable in a CNF, ψ. If x is not dominated in any one of the extracted functional dependencies, then define $D(x) := \{x\}$. Otherwise, define $D(x)$ as the set of all variables, y, in ψ such that there is a sequence of extracted functional dependencies, $x_1 \twoheadleftarrow S_1, \ldots, x_k \twoheadleftarrow S_k$ with $x_{i+1} \in S_i$ for all $i \in \{1, \ldots, k-1\}$ and $y \in S_k$, and y is not dominated in any one of the extracted functional dependencies.*

This is well-defined as the following proposition shows.

Proposition 2. *Our method of extraction generates no cyclic sequence of dependence relations $x_1 \twoheadleftarrow S_1, \ldots, x_k \twoheadleftarrow S_k$ with $x_{i+1} \in S_i$ for all $i \in \{1, \ldots, k-1\}$ and $x_1 \in S_k$.*

Proof. Consider the parse tree of a propositional formula, ϕ, such that internal nodes correspond to logical operators and leaves correspond to variables in ϕ. The Tseitin encodings then introduce a new variable for each node. If the relation, $x_i \twoheadleftarrow S_i$, is extracted, then the depth of the node for any variable in S_i must be greater than or equal to the depth of the node for x_i. In particular, the depths are equal, if and only if, the current subformula has form $z \leftrightarrow \alpha$, where z is a variable and α is not a variable. Since α is not a variable, the variable introduced for α must be dominated by variables of greater depth. Hence, there is no such sequence. □

Our variable order is determined as follows: for each variable x with some extracted relation $x \twoheadleftarrow S$, move x to the position immediately after the greatest variable in $D(x)$ with respect to the original order of the variables in the CNF.

Example 2. *Let us look at Fig. 2 (a). Suppose that only the relation, $x_6 \twoheadleftarrow \{x_1, x_2\}$, is extracted. We then have $D(x_6) = \{x_1, x_2\}$. Since the greatest variable in $D(x_6)$ is x_2, we move x_6 just after x_2. Figure 2 (b) outlines the CNF in the new variable order.*

Our variable order satisfies the property (P), as promised.

Theorem 1. *Let $<$ be the variable order determined by our method. If a variable, x, is dominated in some extracted functional dependency, then there is a set of variables, S, such that $x \twoheadleftarrow S$ and $y < x$ hold for all $y \in S$.*

Proof. Assume relation $x \twoheadleftarrow T$ was extracted. Let n be the maximum length of a sequence of dependence relations $x = x_1 \twoheadleftarrow S_1, \ldots, x_n \twoheadleftarrow S_n$ with $x_{i+1} \in S_i$ for all $i \in \{1, \ldots, n-1\}$. We show by induction on n that $x \twoheadleftarrow D(x)$ holds. The case of $n = 1$ is immediate because $T \subseteq D(x)$ holds. In the case of $n > 1$, any variable y in T is dominated by $D(y)$ according to induction hypothesis. Because $\bigcup_{y \in T} D(y) \subseteq D(x)$ holds, we obtain $x \twoheadleftarrow D(x)$. Therefore, $x \twoheadleftarrow D(x)$ holds for all n. All variables in $D(x)$ are selected earlier than x. □

All dominated variables mentioned in extracted relations are treated as implied variables inside solvers, as stated in the following corollary.

Corollary 1. *Let ψ be a CNF. Suppose that a variable, x, is dominated in some extracted functional dependency in the encoding of ψ. For any assignment ν such that x is unassigned and all variables in $D(x)$ are assigned values, ordinary unit propagation with ψ and ν can determine the assignment to x unless a falsified clause exists.*

We will now present an algorithm for computing the greatest variable in $D(x)$ in our ordering method presented above. We define the *dependence graph*, $G = (V, E)$, such that the vertices in V correspond to variables in a CNF, ψ, and if relation $y \twoheadleftarrow S$ is

Algorithm 3: Algorithm DFS updates d fields of all vertices in subgraph rooted by u.

```
Input: a dependence graph G, a vertex u of G.
foreach vertex v pointed to by u do
    if v.fin = 0 then
        u.fin = 1;
        v.d = u.d;
        DFS(ψ, G, v);
    end
end
```

extracted in the encoding of ψ, then there is an arc in E from each variable in S to y. Let each vertex u of G have the following three fields: $u.var$ holds the variable corresponding to u, $u.d$ is initialized to $u.var$ and updated to the greatest variable in $D(u.var)$, and $u.fin$ is initialized to 0 and updated to 1 if u is visited.

Given a dependence graph, G, our algorithm selects each root u of G in decreasing order of the associating variables, $u.var$, and performs DFS with G and u. Algorithm 3 has the pseudocode of DFS.

By definition, the roots of G correspond to variables that are not dominated in any one of the extracted functional dependencies. For each such root u, our algorithm sets the d fields of all unvisited vertices v in the subgraph rooted by u to $u.var$. Since u is selected in decreasing order, if v is already visited, then $v.d$ must be greater than $u.var$. Therefore, once v is updated, $v.d$ must be the greatest variable of $D(v.var)$ and it will not be updated afterward.

3.3 Exploiting Variable Dependence inside Solvers

This section presents a method of exploiting extracted functional dependencies inside compilation-based solvers. Our method focuses on important variables and avoids explicitly handling non-important variables inside solvers.

We improve the OBDD extension part in Algorithm 1 as follows. When a solution is found, we add a new path to an OBDD basically in the same way as stated in Section 2.2. The main difference is that we only add nodes for important variables so that the added path represents an assignment to only those variables. Recall that when the original version of Algorithm 1 extends an OBDD, the nodes in an added path have consecutive numbers as their labels; in other words, the node elimination rule of BDD is never applied to any node in the paths that lead to the sink node, $\top$. This means that if there are two nodes in an OBDD constructed with our method such that one is the child of the other and the difference in their labels is more than 1, then all eliminated nodes between them must correspond to dependent variables. Note that the assignment to dependent variables, if necessary, can easily be recovered from the assignment to the other variables and an input CNF, as presented in Corollary 1.

Let us consider that our modified OBDD extension is independent of the original mechanism for the equivalence test of subinstances, and no inconsistency occurs. Recall that Algorithm 1 only records nodes for decision variables as key-result pairs in S. For example, see the gray nodes in Fig. 1 (c). Even though multiple nodes have been eliminated in our modified OBDD extension, all of them are nodes for implied variables, and not for decision variables. Hence, they are not involved with key-result pairs to be registered.

Moreover, we do not need to change the function, *insertcache*, which registers key-result pairs to S, because as was previously mentioned, nodes to be registered are not eliminated.

Example 3. *Let us look at the OBDD outlined in Fig. 2 (c), where the solutions that extends the thick path have not yet been searched. Suppose that x_1 and x_2 are now assigned 1 and 0, respectively. Note that the equivalence test our solver then performs is not for subinstances induced by x_2, but for those induced by x_6. This means the effect of our solver effectively exploiting dependent variables; indeed, since our method of ordering moves dependent variables just after the variables dominating them, the assignments to those dependent variables are all implied, and equivalence test is skipped except for the last one of those dependent variables.*

Let us consider how an OBDD is then extended. The least unassigned variable is the 4-th variable, x_3, and the current cutset state is $\{\underline{C_4}\}$. As the same state is attached to the end of the path $1 \dashrightarrow 2 \dashrightarrow$ in Fig. 2 (c), this means that the same state was induced by the past assignment where x_1 and x_2 were both assigned 0. Hence, the current subinstance is already solved. As indicated by the thick path of Fig. 2 (c), our solver adds a new path for important variables in the current assignment. Since all solutions that extend the assignment ν: $x_1 \mapsto 1$ are exhausted, the left-most node of label 2 will be colored in gray. This means that the node associated with its cutset state will be registered as a key-result pair.

4. Experiments

Implementation. We compared the following methods.

- **Grumberg**: the AllSAT solver of Grumberg et al. [6]. We implemented this on top of MiniSat-C v.1.14.1, where the sublevel-based first UIP scheme and conflict-directed backjumping were used in conflict resolution phase, as proposed in Ref. [6].
- **bdd**: a standard compilation-based AllSAT solver based on Algorithm 1. The implementation bdd_minisat_all v1.0.0 was obtained from All Solutions SAT Repository [*3]. The decision level-based first UIP scheme and limited non-chronological backtracking were used in conflict resolution phase, as the efficiency of the combination has been confirmed [17].
- **depbdd**: our proposed method. We implemented this by modifying **bdd**.

All methods received DIMACS CNF instances with functional dependency information included. **Grumberg** and **depbdd** used dependence information inside solvers, based on important first decision procedure for the former [6] and on our proposed method for the latter, while **bdd** did not. **Grumberg** used, as important variables, those that were not dominated by any one of the functional dependencies our method extracted. **Grumberg** searched all total satisfying assignments, **bdd** constructed the OBDD for

[*3] All Solutions SAT Repository by T. Toda, http://www.sd.is.uec.ac.jp/toda/code/allsat.html. It was accessed Sep. 30, 2015.

Table 1 Used instances of LMCS-2006, where bound means the shortest bounds such that counterexamples are found.

model	property	bound	variables	ratio of important variables	clauses	cutwidth
abp4	p2false	17	7,587	0.42	22,909	5,347
abp4	ptimo	16	7,468	0.37	22,532	4,200
bc57-sensors	p0neg	103	6,284,394	0.99	18,855,596	1,091,882
bc57-sensors	p1neg	103	2,330,258	0.98	6,993,188	3,354,394
bc57-sensors	p2neg	103	2,027,654	0.97	6,085,376	2,863,913
bc57-sensors	p3	103	89,950	0.65	272,263	107,593
brp	p1neg	1	364	0.88	855	75
brp	ptimoneg	1	364	0.86	855	40
brp	ptimonegnv	24	83,751	0.90	251,013	69,568
counter	p0neg	8	66	1.00	131	13
dme2	ptimoneg	1	222	0.85	445	5
dme2	ptimonegnv	39	3,531,818	0.72	10,597,515	6,304,734
dme3	p1neg	1	331	0.72	664	7
dme3	ptimoneg	1	331	0.72	664	7
dme4	p1neg	1	448	0.83	905	10
dme4	ptimoneg	1	448	0.83	905	10
dme5	p1neg	1	559	0.69	1,129	12
dme5	ptimoneg	1	559	0.69	1,129	12
dme6	p1neg	1	672	0.69	1,360	17
dme6	ptimoneg	1	672	0.69	1,360	17
mutex	p0neg	6	368	0.45	1,087	53
production-cell	p3neg	81	4,068,468	0.99	12,205,968	413,607
production-cell	p4neg	81	5,148,273	0.01	15,445,383	419,657
ring	p0neg	7	225	1.00	616	235
short	p0neg	1	8	1.00	14	1
srg5	ptimoneg	1	35	1.00	62	6
srg5	ptimonegnv	6	2,171	0.95	6,399	2822

all satisfying assignments over all variables, and **depbdd** constructed the OBDD for all satisfying assignments over variables that were not dominated in any one of the extracted functional dependencies.

Environment. We conducted all experiments on a computer with a 2.13-GHz Xeon®E7-2830 processor and 512 GB of RAM, running CentOS 6.7 with gcc compiler 4.4.7.

Problem Instances. We modified NuSMV version 2.5.4 [*4] to be able to extract functional dependencies while creating CNF instances from NuSMV models. The basic idea of bounded model checking is to model a system to be verified as a finite state transition system and, given a number k, to determine whether there is a counterexample to a desired property for that system in k steps of transitions. Our modified NuSMV receives a model, a property, and bound k, and it generates a CNF such that satisfying assignments correspond to counterexamples [*5]. This CNF is printed out in DIMACS CNF format with extracted functional dependencies in its header.

Our method is evaluated with a real network dataset, denoted by **stanford**, obtained from the backbone network in Stanford university [*6] and a common bounded model checking dataset [*7], denoted by **LMCS-2006**.

The Stanford network has been used to evaluate network verification tools [10], [12]. It consists of 16 switches with 58 interfaces in total. Packets are forwarded by the switches based on their header bits; the forwarding decision is made with 88 bits including IP address, TCP port number, and so on. The Stanford backbone network is modeled with 94 binary variables: 88 bits for packet header and 6 bits for switch interfaces. The dataset can be converted into NuSMV models by the bundled script (nu_smv/nu_smv_generator.py).

Network verification is usually used to check conformance with operational policies, but the Stanford dataset includes no material that can be used to specify policies. In the experiments, we try to find counterexamples under a hypothetical policy — all packets should reach their destinations without being filtered out inside the network. We randomly choose 765 interface pairs and examine their reachability. The reachability properties are converted into CNF instances with the shortest bounds such that counterexamples are found; the bounds are 2 for internal interfaces, while they are 6 for external ones (note that reachability related to special addresses, e.g., 0.0.0.0, 255.255.255.255, and 224.*.*.*, are ignored, because counterexamples are not found until bound 100).

Used instances of the bounded model checking dataset LMCS-2006 and their statistics are listed in **Table 1**. These instances are pairs of NuSMV models and their properties, and they are encoded into CNFs with the shortest bounds such that counterexamples are found. The fourth and sixth columns state the number of variables and the number of clauses in each encoded CNF, respectively. The fifth column states the ratio of the number of important variables divided by the number of all variables, where important variables are those extracted by our proposing method (see Section 3.1). The other instances included in that dataset are not used because counterexamples could not be reached in several

[*4] NuSMV version 2.5.4 was obtained from http://nusmv.fbk.eu/. It was accessed May 29, 2015.
[*5] The original NuSMV only computes an equisatisfiable CNF, i.e., a CNF that is satisfiable if and only if a counterexample exists. Our modified NuSMV, on the other hand, encodes bidirectional implications between variables and their renaming subformulae to ensure a one-to-one correspondence between satisfying assignments and counterexamples.
[*6] https://bitbucket.org/peymank/hassel-public/
[*7] http://fmv.jku.at/aiger/lmcs2006-aiger-1.9-benchmarks.tar.gz

Table 2 Distribution of stanford instances according to numbers of found counterexamples.

Number of Counterexamples	Grumberg	bdd	depbdd
$[0, 10^1)$	0	1	1
$[10^1, 10^2)$	0	0	0
$[10^2, 10^3)$	0	0	0
$[10^3, 10^4)$	0	0	0
$[10^4, 10^5)$	0	0	0
$[10^5, 10^6)$	0	0	0
$[10^6, 10^7)$	0	0	0
$[10^7, 10^8)$	0	0	0
$[10^8, 10^9)$	213	0	155
$[10^9, 10^{10})$	552	0	18
$[10^{10}, 10^{11})$	0	0	199
$[10^{11}, 10^{12})$	0	0	2
$[10^{12}, 10^{13})$	0	0	2
total	765	1	377

hours.

Results. The time limit was set to 600 s and the memory limit was set to 50 GB. If the time limit is exceeded, a solver is interrupted, and it reports the progress at this time. On the other hand, since memory usage is monitored by ulimit command, if the memory limit is exceeded, a solver is forced to halt immediately without reporting any information.

Table 2 shows the distribution of stanford instances according to the numbers of found counterexamples. All instances are classified in terms of the numbers of counterexamples found by solvers, where we note that all stanford instances cannot be solved by any one of the compared solvers, i.e. not all counterexamples can be found, within the time limit. According to Table 2, **bdd** and **depbdd** run out of memory in almost all the instances and in half the instances, respectively. This shows an efficiency of **depbdd** compared to **bdd**; our method, **depbdd**, focuses on important variables only, thereby saving a large amount of memory. Furthermore, **depbdd** can find more counterexamples than **Grumberg** by a few orders of magnitude. There is a limit in the number of counterexamples that can be found in a one-by-one fashion, and instances with many counterexamples can be efficiently handled with the dynamic programming approach using BDD data structure. Our method is, thus, suitable for this kind of instances, and makes it possible to efficiently utilize the dynamic programming approach within a limited amount of memory. We remark that **bdd** and **depbdd** could not find any counterexample for one instance because the instance was very large and the majority of running time was spent in setting up the caching mechanism of BDD solvers, which is a preprocessing phase.

We conducted the same experiment with maximum node limit enabled in **bdd** and **depbdd** to avoid running out of memory. This functionality [*8] limits the maximum number of BDD nodes, and if the threshold is exceeded, the number of solutions found so far is recorded, the BDD constructed is then discarded, and the search resumes while constructing a new BDD where new solutions found afterword are stored. This does not affect the algorithmic behavior of solvers except for the deterioration of "cache hit", i.e., the ability of saving the recomputation of equivalent subproblems. We set the maximum number of BDD nodes in **bdd** and **depbdd** to 10^9.

[*8] The maximum node limit is implemented in the original compilation-based solver, **bdd**, and it is available in our solver, **depbdd**, as is.

Table 3 Distribution of stanford instances according to numbers of found counterexamples, conducted with maximum node limit.

Number of Counterexamples	Grumberg	bdd	depbdd
$[0, 10^1)$	0	1	1
$[10^1, 10^2)$	0	0	0
$[10^2, 10^3)$	0	0	0
$[10^3, 10^4)$	0	0	0
$[10^4, 10^5)$	0	0	0
$[10^5, 10^6)$	0	0	0
$[10^6, 10^7)$	0	0	0
$[10^7, 10^8)$	0	285	127
$[10^8, 10^9)$	213	46	188
$[10^9, 10^{10})$	552	396	210
$[10^{10}, 10^{11})$	0	25	223
$[10^{11}, 10^{12})$	0	10	12
$[10^{12}, 10^{13})$	0	2	4
total	765	765	765

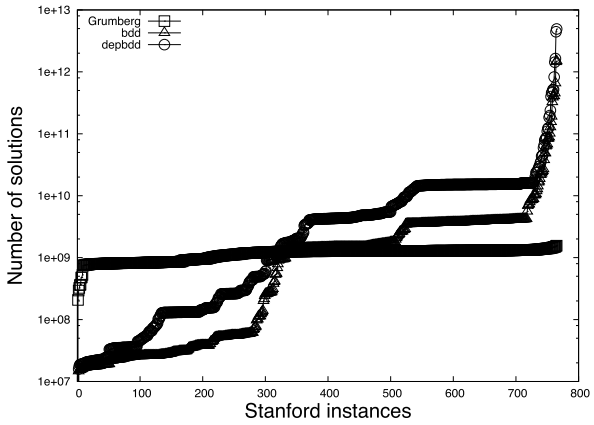

Fig. 3 Cactus plot w.r.t the numbers of found counterexamples, conducted with maximum node limit.

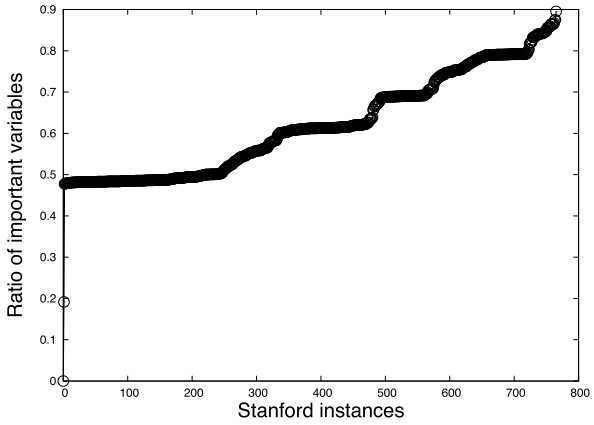

Fig. 4 Cactus plot w.r.t the ratios of important variables.

As **Table 3** shows, out of memory does not occur; **bdd** and **depbdd** find more counterexamples than **Grumberg** by a few orders of magnitude, although there are instances with fewer counterexamples found due to the deterioration of cache hit. The same result is depicted as a cactus plot in **Fig. 3**, which is given in logarithmic scale.

A comparison of OBDD sizes between **bdd** and **depbdd** is not presented because the functionality of maximum node limit refreshes an OBDD as soon as its size exceeds a threshold, and thus comparing OBDD sizes does not make sense.

Figure 4 is a cactus plot of stanford instances with respect to the ratio of the number of important variables divided by the number of all variables, which shows that the numbers of indenepdent

Table 4 Time comparison on LMCS-2006 instances, where cex indicates the number of counterexamples, - means unknown, TO indicates timeout, and OM indicates out of memory.

model	property	cex	Grumberg	bdd	depbdd
abp4	p2false	3,072	3.11	138.25	145.69
abp4	ptimo	256	3.50	112.39	89.52
bc57-sensors	p0neg	-	TO	OM	OM
bc57-sensors	p1neg	-	TO	OM	OM
bc57-sensors	p2neg	-	TO	OM	OM
bc57-sensors	p3	1,400	9.23	TO	TO
brp	p1neg	2	0.00	0.00	0.00
brp	ptimoneg	2	0.00	0.00	0.00
brp	ptimonegnv	5,328	105.72	TO	TO
counter	p0neg	1	0.00	0.00	0.00
dme2	ptimoneg	3	0.00	0.00	0.00
dme2	ptimonegnv	-	TO	OM	TO
dme3	p1neg	4	0.00	0.00	0.00
dme3	ptimoneg	4	0.00	0.00	0.00
dme4	p1neg	5	0.00	0.00	0.00
dme4	ptimoneg	5	0.00	0.00	0.00
dme5	p1neg	6	0.00	0.00	0.00
dme5	ptimoneg	6	0.00	0.00	0.00
dme6	p1neg	7	0.00	0.00	0.00
dme6	ptimoneg	7	0.00	0.00	0.00
mutex	p0neg	1	0.00	0.00	0.00
production-cell	p3neg	1	333.06	OM	OM
production-cell	p4neg	1	329.51	OM	TO
ring	p0neg	3	0.00	0.00	0.00
short	p0neg	1	0.00	0.00	0.00
srg5	ptimoneg	7	0.00	0.00	0.00
srg5	ptimonegnv	6	0.00	0.07	0.08

variables amount to 50 to 60 percent of the whole over about half of the instances.

Table 4 shows the time comparison of solvers over LMCS-2006 instances. As indicated in the third column, these instances do not have many counterexamples, and hence, **Grumberg** significantly outperforms BDD-type solvers. This is consistent with the experimental evaluation on AllSAT solvers conducted in Ref. [17]. This simply means that BDD-type solvers are not always the best choice and does not harm the effectiveness of our proposing method. The **bdd** and **depbdd** run out of memory in several instances. The reason for this is that the memory limit exceeds in a preprocessing phase prior to searching.

We also conducted a size comparison of BDD-type solvers over LMCS-2006. As **Table 5** shows, **depbdd** requires less nodes than **bdd**, which exhibits an efficiency of our method in memory usage.

5. Conclusion

The research discussed in this paper improved a state-of-the-art AllSAT solver based on OBDD compilation in terms of functional dependencies between variables. We focused on a compilation-based solver out of the several types of existing solvers available because its power in finding a large number of solutions in dynamic programming manner has recently been recognized. Our AllSAT solver was integrated with the encoding of propositional Boolean formulae into CNFs to exploit the dependence: i.e., given a propositional Boolean formula, our solver encodes it into a CNF while extracting functional dependencies; it is then effectively utilized to construct an OBDD over important variables only. It turns out that CNF instances encoded from a real network dataset have a large number of solutions, and

Table 5 BDD size comparison on LMCS-2006 instances.

model	property	bdd	depbdd
abp4	p2false	17,952,139	5,390,393
abp4	ptimo	1,839,721	632,417
bc57-sensors	p0neg	0	0
bc57-sensors	p1neg	0	0
bc57-sensors	p2neg	0	0
bc57-sensors	p3	1	1
brp	p1neg	575	503
brp	ptimoneg	539	376
brp	ptimonegnv	1	1
counter	p0neg	66	66
dme2	ptimoneg	402	349
dme2	ptimonegnv	0	0
dme3	p1neg	880	560
dme3	ptimoneg	880	560
dme4	p1neg	1,377	1,115
dme4	ptimoneg	1,377	1,115
dme5	p1neg	2,235	1,289
dme5	ptimoneg	2,235	1,289
dme6	p1neg	3,143	1,755
dme6	ptimoneg	3,143	1,755
mutex	p0neg	368	164
production-cell	p3neg	0	0
production-cell	p4neg	0	0
ring	p0neg	647	647
short	p0neg	8	8
srg5	ptimoneg	84	84
srg5	ptimonegnv	12,666	12,054

compilation-based solvers are suitable for this kind of instances: they find more solutions compared to Grumberg's solver by a few orders of magnitude. Our proposing technique accelerates the original compilation-based solver further.

In network verification applications [3], [13], [14], solutions of Boolean formulae correspond to packets that violate network policies. Since a single packet is not sufficient to identify the cause of violation, it is important to efficiently compute a packet set, i.e., a set of packets that corresponds to a subnet, a port range, and so on.

We are currently applying our AllSAT solver to model checking. Some researches have pursuit integration of SAT solvers and BDDs; however, to the best of our knowledge, all such approaches have only used BDDs as a succinct data structure for representing reached states and have not employed a dynamic programming technique such as avoiding recomputation of equivalent subproblems, as this paper focused on. As was mentioned in Section 1, a large number of dependent variables can be spawned when model checking problems are formulated. We expect that our optimized compilation-based solver will be effective for those problems.

Ivrii et al. [11] proposes algorithms for computing an *independent support*, i.e., a set of variables by which the other variables are all functionally dependent. Although our approach exploits functional dependencies that are easily obtained from propositional formulae at the syntactical level, their method incorporates with an MUS computation so that it is able to find dependent variables that are hidden deeply below the syntactical level. Their method, however, only determines that each dependent variable is dominated by the whole independent support. Since the variables in the support are selected first in our ordering method, this is nothing but important first decision procedure of Grumberg

et al. and it is thus not efficient in compilation-based solvers as discussed in this paper. We thus need to locate a smaller set of variables for each dependent variable to efficiently integrate the method of Ivrii et al. with ours, which is for future work.

Acknowledgments This work is supported by JSPS KAKENHI Grant Number 26870011 and 17K17725.

References

[1] Abdulla, P.A., Bjesse, P. and Eén, N.: Symbolic Reachability Analysis Based on SAT-Solvers, *Proc. 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, TACAS '00*, London, UK, UK, pp.411–425, Springer-Verlag (2000). (online), available from ⟨http://dl.acm.org/citation.cfm?id=646484.691763⟩

[2] Biere, A., Heule, M., van Maaren, H. and Walsh, T.: *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, IOS Press, Amsterdam, The Netherlands (2009).

[3] Bjørner, N. and Varghese, G.: Network Verification: When Hoare Meets Cerf, *ACM SIGCOMM Tutorial* (2015).

[4] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Computers*, Vol.C-35, No.8, pp.677–691 (online), DOI: 10.1109/TC.1986.1676819 (1986).

[5] Clarke, E., Kroening, D., Sharygina, N. and Yorav, K.: Predicate Abstraction of ANSI-C Programs Using SAT, *Formal Methods in System Design*, Vol.25, No.2-3, pp.105–127 (online), DOI: 10.1023/B:FORM.0000040025.89719.f3 (2004).

[6] Grumberg, O., Schuster, A. and Yadgar, A.: Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis, *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pp.275–289, Springer Berlin Heidelberg (online), DOI: 10.1007/978-3-540-30494-4_20 (2004).

[7] Guns, T., Nijssen, S. and Raedt, L.D.: Itemset mining: A constraint programming perspective, *Artificial Intelligence*, Vol.175, No.1213, pp.1951–1983 (online), DOI: http://dx.doi.org/10.1016/j.artint.2011.05.002 (2011).

[8] Gupta, A., Yang, Z., Ashar, P. and Gupta, A.: SAT-Based Image Computation with Application in Reachability Analysis, *Formal Methods in Computer-Aided Design: 3rd International Conference, FMCAD 2000, Austin, TX, USA, November 1–3, 2000, Proceedings*, pp.391–408 (online), DOI: 10.1007/3-540-40922-X_22, Springer Berlin Heidelberg (2000).

[9] Huang, J. and Darwiche, A.: Using DPLL for Efficient OBDD Construction, *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pp.157–172, Springer Berlin Heidelberg (online), DOI: 10.1007/11527695_13 (2005).

[10] Inoue, T., Chen, R., Mano, T., Mizutani, K., Nagata, H. and Akashi, O.: An Efficient Framework for Data-plane Verification with Geometric Windowing Queries, *IEEE ICNP*, pp.1–10 (2016).

[11] Ivrii, A., Malik, S., Meel, K.S. and Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting, *Constraints*, Vol.21, No.1, pp.41–58 (online), DOI: 10.1007/s10601-015-9204-z (2016).

[12] Kazemian, P., Varghese, G. and McKeown, N.: Header Space Analysis: Static Checking for Networks, *USENIX NSDI*, pp.113–126 (2012).

[13] Lopes, N., Bjorner, N., Godefroid, P. and Varghese, G.: Network Verification in the Light of Program Verification, Technical Report, Microsoft Research (2013).

[14] Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K. and Varghese, G.: Checking Beliefs in Dynamic Networks, *Proc. 12th USENIX Conference on Networked Systems Design and Implementation, NSDI '15*, pp.499–512, Berkeley, CA, USA, USENIX Association (2015). (online), available from ⟨http://dl.acm.org/citation.cfm?id=2789770.2789805⟩.

[15] Majumdar, R., Tetali, S.D. and Wang, Z.: Kuai: A Model Checker for Software-defined Networks, *Proc. 14th Conference on Formal Methods in Computer-Aided Design, FMCAD '14*, pp.27:163–27:170, Austin, TX, FMCAD Inc., (2014). (online), available from ⟨http://dl.acm.org/citation.cfm?id=2682923.2682953⟩.

[16] McMillan, K.L.: Applying SAT Methods in Unbounded Symbolic Model Checking, *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings*, pp.250–264, Springer Berlin Heidelberg (online), DOI: 10.1007/3-540-45657-0_19 (2002).

[17] Toda, T. and Soh, T.: Implementing Efficient All Solutions SAT Solvers, *J. Exp. Algorithmics*, Vol.21, pp.1.12:1–1.12:44 (online), DOI: 10.1145/2975585 (2016).

[18] Toda, T. and Tsuda, K.: BDD Construction for All Solutions SAT and Efficient Caching Mechanism, *Proc. 30th Annual ACM Symposium on Applied Computing, SAC '15*, pp.1880–1886, New York, NY, USA, ACM (online), DOI: 10.1145/2695664.2695941 (2015).

[19] Zhang, S., Malik, S. and McGeer, R.: Verification of Computer Switching Networks: An Overview, *Automated Technology for Verification and Analysis: 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012, Proceedings*, pp.1–16, Springer Berlin Heidelberg (online), DOI: 10.1007/978-3-642-33386-6_1 (2012).

Takahisa Toda received his B.E. degree in integrated human studies, and M.E. and Ph.D. degrees in human and environmental studies from Kyoto University, Japan, in 2004, 2006, and 2012, respectively. He is an Assistant Professor of graduate school of informatics and engineering, the university of electro-communications. He was an ERATO Researcher with the Japan Science and Technology Agency, from 2012 to 2014. His research interests are experimental analysis and applications of discrete algorithms. He is a member of ACM, IPSJ, JSAI and IEICE.

Takeru Inoue received his B.E. and M.E. degrees in engineering science and Ph.D. degree in information science from Kyoto University, Japan, in 1998, 2000, and 2006, respectively. He is a Distinguished Researcher with Nippon Telegraph and Telephone Corporation (NTT) Laboratories. He was an ERATO Researcher with the Japan Science and Technology Agency, from 2011 to 2013. His research interests widely cover algorithmic approaches in computer networks. He was a recipient of the Best Paper Award of the Asia-Pacific Conference on Communications in 2005. He is a member of IEEE and IEICE.