Control Flow-Guided SMT Solving for Program Verification*

Jianhui Chen
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
chenjian16@mails.tsinghua.edu.cn

Fei He†
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hefei@tsinghua.edu.cn

ABSTRACT

Satisfiability modulo theories (SMT) solvers have been widely applied as the reasoning engine for diverse software analysis and verification technologies. The efficiency of the SMT solver has significant effects on the performance of these technologies. However, the current SMT solvers are designed for the general purpose of constraint solving. Many useful knowledge of programs cannot be utilized during the SMT solving. As a result, the SMT solver may spend a lot of effort to explore redundant search space. In this paper, we propose a novel approach for utilizing control-flow knowledge in SMT solving. With this technique, the search space can be considerably reduced and the efficiency of SMT solving is observably improved. We conducted extensive experiments on credible benchmarks, the results show orders of magnitude improvements of our approach.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; • **Software and its engineering → Formal software verification**;

KEYWORDS

Program verification, Satisfiability modulo theory, Control-flow graph

ACM Reference Format:
Jianhui Chen and Fei He. 2018. Control Flow-Guided SMT Solving for Program Verification. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238218

*This work was partially funded by the NSF of China under Grant No. 61672310 and Grant No. 61527812, the National Science and Technology Major Project under Grant No. 2016ZX01038101.
†Corresponding author

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238218

1 INTRODUCTION

Satisfiability modulo theories (SMT) are applied in diverse software engineering technologies, such as software verification [4, 22], static program analysis [12, 36], symbolic execution [26], test-case generation [37] and so on. A powerful SMT solver is a crucial factor to improve the efficiency of these technologies. However, the SMT solvers are designed for the general purpose of constraint solving [16]. Domain-specific knowledge cannot be efficiently utilized in nowadays SMT solvers. On the other hand, domain knowledge may be quite useful for pruning the search space, and thus improving the efficiency of SMT solvers.

Control-flow graph (CFG) is a graph representation of programs. It is simple but informative. Many domain knowledge about programs is implied in this representation. For example, let π be a program execution, if π passes a basic block, it passes all statements in this block; reversely, if it does not pass a block, all statements in that block are not executed by π. This is straightforward from the CFG. However, after the program is encoded into the SMT formula, the SMT solver is unaware of this knowledge. Moreover, consider an if-statement, an execution can never pass both of its branches. Again, the SMT solver is unaware of this. As a result, the SMT solver may spend a lot of effort to explore redundant space, which could have been pruned if the control-flow knowledge is known.

One may consider to specify the control-flow knowledge as SMT constraints. However, specifying these knowledge may introduce additional variables and thus increases the problem complexity. We have a simpler and smarter solution. The basic idea is to use a decision order to guide the SMT solving. The decision order decides at each search step which variable should be taken for assignment. It determines the search direction in the SMT solving. We propose to infer a decision order from the control-flow graph of the program, and then use this order to guide the search process of SMT solving.

We propose two heuristics for arranging the decision order. Firstly, we recognize the importance of branching conditions that dominate the control flows of the program, and assign the corresponding variables higher priorities in the decision order. Secondly, we follow the domination relation of the control flow graph to arrange the order of branching variables. In this way, we propose a lightweight technique to utilize control-flow knowledge in SMT solving. It can take full advantage of the built-in features of the SMT solver.

Moreover, we propose an enhanced CNF conversion procedure. After the program is encoded into the SMT formula, the *ite* terms keep the control-flow information of the original program. The

conventional approach breaks the nested structures of *ite* terms, and thus lose the control-flow information. We propose a new *ite* rewriting algorithm, with which the nested structure of *ite* terms is kept in some form of CNF formulas. With our enhanced approach, the generated CNF contains fewer clauses and fewer variables. The efficiency is thus improved.

To the best of our knowledge, this is the first attempt at improving SMT solving by using domain knowledge of programs. There are some works on domain knowledge-guided SAT solving [6, 40, 43, 44]. In [44], the structure information of transition systems is utilized to guide the SAT solvers. In [40], the variable dependency information of the model is utilized to guide the SAT solvers. The paper [43] considers the iterative SAT solving for bounded model checking. The information from the previous unsatisfiability core is utilized to refine the decision ordering for the current SAT instance. All these techniques are designed for SAT solvers, and the domain knowledge comes from the model checking. In this paper, we consider SMT solving for program verification. The domain knowledge is different, and the application target is also different.

We realized a prototype of our approach on top of CBMC and Z3. We conducted extensive experiments on 2897 credible benchmarks. All benchmarks are obtained from SV-COMP'17[1]. Up to 1411 SMT instances are generated. Experimental results show that the control-flow knowledge can significantly improve the efficiency of SMT solving. The average speedup is 3.34 times for all instances, and 8.22 times for satisfiable instances.

In summary, our main contributions include:

- We propose a novel approach for utilizing control flow knowledge in SMT solving. This approach is lightweight, and can take full advantage of the built-in features of SMT solvers.
- We propose an enhanced CNF conversion procedure, with which the control-flow information can be kept in the CNF formulas.
- We implement our approach on the top of CBMC and Z3, and conduct extensive experiments to evaluate its effectiveness and efficiency. Results show promising performance of our approach.

The rest of this paper is organized as follows. Section 2 introduces some background knowledge. Section 3 uses a simple example to motivate our approach. Section 4 presents our control-flow guided SMT solving approach. Experiments are reported in Section 5, related works are discussed in Section 6. Finally, Section 7 concludes this paper.

2 PRELIMINARIES

2.1 Notations

In first-order logic (FOL), a *term* can be a variable, a constant, or a n-ary function applied to n terms. An *atom* is $\top$, $\bot$, or a n-ary predicate applied to n terms. A *literal* is an atom or its negation. A FOL formula is built from literals using the Boolean connectives and quantifiers. An interpretation (or model) M consists of a non-empty set of objects called the domain of M, written $dom(M)$; a map from each variable and each constant, respectively, to an object in $dom(M)$; and an interpretation for each function symbol and each predicate symbol, respectively. Given a formula Φ, we say M satisfies Φ, written $M \models \Phi$, if Φ is true in the model M.

A first-order theory T is defined by a signature and a set of axioms, where the signature defines a set of constant, function and predicate symbols allowed in T, and the axioms define the intended meanings. A T-model is a model that satisfies all axioms of T. A formula Φ is T-satisfiable if there exists a T-model M such that $M \models \Phi$. A formula Φ is T-valid if $M \models F$ for all T-models M.

2.2 Control Flow Graph

A *control flow graph* (CFG) is a graphical representation of computation and control flow in the program. Let a *basic block* be a straight-line sequence of instructions without any jumps or jump targets. Especially, jump targets start a block and jumps end a block. A control flow graph (CFG) is a directed graph, where nodes are basic blocks and edges represent jumps in the control flow. Two specially designated blocks, `ENTRY` and `EXIT`, may be used in a CFG to represent the entry and exit points of the program.

Let c be the condition of a branch-statement. The c is called a *branching condition*. And two branches of the branch-statement is called c branch and $\neg c$ branch, respectively. Given a basic block, its *block condition* is a predicate such that the block is executable iff the block condition is satisfiable.

2.3 Program Verification

Recent advances in model checking [6], static analysis [36], abstract interpretation [12] predicate abstraction [4, 22], etc, promoted program verification to a practical technique for correctness assurance of programs. Loops are the main hurdle for program verification. There are mainly two approaches for handling loops in a program: loop invariant [39] and loop unwinding [34]. The former approach uses a loop invariant, which holds at the beginning of each iteration of the loop, to represent the behaviors of the loop. However, it relies on the user to provide the loop invariant. Automatic generation of loop invariants have been extensively studied, but the existing techniques are still not practical enough [1]. The more popular approach for handling loops is by unwinding. With this technique, each loop is unwound to a predefined depth. As a result, the loops are replaced by nested if-statements. This technique is good at bug finding. Other techniques, like k-induction [17], enhances this approach by enabling the correctness proving of programs.

Many of state-of-the-art program verification techniques are based on the satisfiability modulo theories (SMT) solvers [5]. These solvers are used as the reasoning engine. In this paper, we assume all programs have been processed by the loop invariant or loop unwinding technique, and are thus free of loops. The loop-free programs are converted to their static single assignment (SSA) forms [13]. With the SSA form, the correctness of a program can easily be encoded as a set of SMT formulas [18]. These formulas are called the verification condition of the program with respect to the property. The validity of the verification condition implies the correctness of the program.

[1] https://sv-comp.sosy-lab.org/2017/

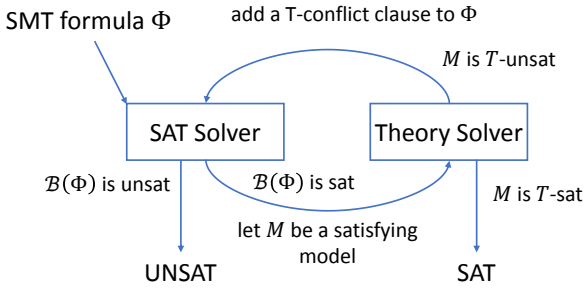

Figure 1: DPLL(T)

2.4 Satisfiability Modulo Theories

Satisfiability modulo theories (SMT) extends SAT with the ability to reason with first-order theories. We assume all theories discussed in this paper are decidable, and for each theory T, there is a T-solver that can check the T-satisfiability of conjunctions of literals in T. In practice, theories are not isolated. For example, software verification needs theories of uninterpreted function, arithmetic, array, bitvectors, and so on. Nelson-Oppen proposed a combination method [33] to deal with FOL formulas in multiple theories.

DPLL(T) extends DPLL algorithm [14] to incorporate reasoning about theories. It uses an SAT solver to cope with the Boolean structure and theory solvers for deciding satisfiability in background theories. The basic idea of DPLL(T) is illustrated in Fig. 1. Given an SMT formula Φ, each of its atoms is first replaced with a fresh Boolean variable, called the *Boolean abstraction*. Denote the resulting formula as $\mathcal{B}(\Phi)$. The Boolean abstraction gives us a *lazy* way to solve the SMT formulas. DPLL(T) uses an SAT solver to find assignments for $\mathcal{B}(\Phi)$ and then uses a theory solver to check the T-satisfiability of the found assignments. The Boolean abstraction is an over-approximation of the original formula with respect to its satisfiability. If $\mathcal{B}(\Phi)$ is unsatisfiable, then so is Φ, but the reverse may not hold.

The high-level view of a practical DPLL(T) algorithm is shown in Alg. 1. At this level, the algorithm is the same as that of a conflict-driven clause learning-based SAT solver [7]. The differences lie in the implementations of *propagate_and_check*(), *resolve_conflict*() and *decide*(). The method *propagate_and_check*() repeatedly applies unit propagation and theory propagation to force values to as many as possible literals. It also checks the T-consistency of the current model. The method returns 0 if it encounters a conflict or T-inconsistency, and 1 otherwise. In case of conflict or T-inconsistency, the method *resolve_conflict*() is invoked to learn conflict clauses and add them to the clause database. The method *decide*() decides the next unassigned Boolean variable and guess its value. If there is no unassigned variable, the current model is complete, and is thus a satisfying model.

Many heuristics are developed for selecting the next unassigned variable and deciding its value. A commonly used branching heuristic is the VSIDS branching heuristic, which is employed as the default heuristic in many well-known solvers [8, 15, 32]. DPLL(T) is essentially a depth-first search algorithm. We may guide the searching process of DPLL(T) by enforcing a variable order in the *decide*() method.

Algorithm 1 DPLL(T)

Input: An SMT formula Φ
Output: SAT or UNSAT

```
1: while true do
2:     while !propagate_and_check() do
3:         if decision_level == 0 then return UNSAT;
4:         else resolve_conflict();
5:     if !decide() then return SAT;
```

3 MOTIVATIONS

In this section, we use a simple example to motivate our approach. We show that some important control-flow knowledge is neglected by the SMT solver, and utilizing this knowledge can make great gains.

Consider a simple program shown in the left of Fig. 2. Its main part is a two-tier nested if-statements. Control-flow graph of this program is shown in Fig. 3, where ellipse nodes represent `Entry` and `Exit`, and rectangle nodes represent blocks. We ignore the detailed forms of the branching conditions in the program, and simply represent them as c_0 and c_1, respectively. The property is to verify that $x = y$ holds at the end of this program.

Verification Condition. The original program is firstly converted into the single static assignment (SSA) form, where for each assignment-statement, a new variable is introduced for the right-hand-side variable. After this conversion, there is at most one assignment for each variable. The SSA form of the program example is shown in the right of Fig. 2. Note that $ite(bool, \cdot, \cdot)$ is a ternary function that returns its second or third argument depending on if its first argument is true or not. Also note that the nested structure of the original if-statement is kept in the formula as a nested *ite* structure.

The verification condition (*VC*) is

$$VC \triangleq Enc \wedge Cor \tag{1}$$

where *Enc* is the encoding of the program, and *Cor* the correctness condition. For the motivating example, we have

$$Enc \equiv (x_4 = ite(c_0, 1, ite(c_1, 2, 3))) \wedge (y_4 = ite(c_0, 1, ite(c_1, 2, 3)))$$

and

$$Cor \equiv (x_4 \neq y_4)$$

The program is *correct* with respect to the property if and only if *VC* is *unsatisfiable*.

Conjunctive Normal Form. We rely on an SMT solver to check the satisfiability of *VC*. The formula need be converted to *conjunctive normal form* (CNF). A conjunctive normal form is a conjunction of clauses, and each clause is a disjunction of literals. For ease of understanding, we usually write a clause as a logical entailment, since it can be converted to a clause directly, i.e., $A_1 \wedge \cdots \wedge A_k \rightarrow B$ can be converted to $\neg A_1 \vee \cdots \vee \neg A_k \vee B$, where $A_1, \ldots, A_k$ and B are literals. In the remainder of the paper, we also write a CNF as a set of clauses, and a clause as a set of literals.

Most SMT solvers adopt the Tseitin's transformation method [42] to perform the CNF conversion, which adds a new variable for each subformula of the original formula. Consider the motivating

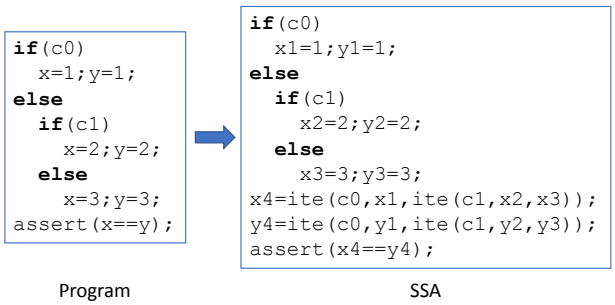

Figure 2: Example

example, the CNF of *Enc*, written *CNF*(*Enc*), is the conjunction of the following clauses:

$$\begin{array}{ll} c_0 \rightarrow x_4 = 1 & c_0 \rightarrow y_4 = 1 \\ \neg c_0 \rightarrow x_4 = t_x & \neg c_0 \rightarrow y_4 = t_y \\ c_1 \rightarrow t_x = 2 & c_1 \rightarrow t_y = 2 \\ \neg c_1 \rightarrow t_x = 3 & \neg c_1 \rightarrow t_y = 3 \end{array} \quad (2)$$

Note that the auxiliary variables t_x and t_y are introduced for the inner *ite* function of VC_{x_4} and VC_{y_4}, respectively. Moreover, $CNF(VC) \equiv CNF(Enc) \wedge CNF(Cor)$, where $CNF(Cor) \equiv x_4 \neq y_4$.

The Tseitin's method guarantees that the converted formula is equi-satisfiable to the original formula. In other words, the verification condition *VC* is satisfiable iff *CNF*(*VC*) is satisfiable.

DPLL(T). DPLL(T) is the standard technique underlying modern SMT solvers. It starts by replacing each atom of the formula with a fresh Boolean variable, called *Boolean abstraction*. In the following, we use v_l to represent the Boolean variable for the atom l. The Boolean abstraction of *CNF*(*Enc*), written $\mathcal{B}(CNF(Enc))$, is

$$\begin{array}{ll} v_{c_0} \rightarrow v_{x_4=1} & v_{c_0} \rightarrow v_{y_4=1} \\ \neg v_{c_0} \rightarrow v_{x_4=t_x} & \neg v_{c_0} \rightarrow v_{y_4=t_y} \\ v_{c_1} \rightarrow v_{t_x=2} & v_{c_1} \rightarrow v_{t_y=2} \\ \neg v_{c_1} \rightarrow v_{t_x=3} & \neg v_{c_1} \rightarrow v_{t_y=3} \end{array} \quad (3)$$

Moreover, $\mathcal{B}(CNF(VC)) \equiv \mathcal{B}(CNF(Enc)) \wedge \mathcal{B}(CNF(Cor))$, where $\mathcal{B}(CNF(Cor)) \equiv \neg v_{x_4=y_4}$.

3.1 Control-Flow Knowledge is Neglected

The first knowledge is about the execution of statements in the same block. Considering the Boolean formula (3), all variables are independent each other. The variables $v_{x_4=1}$ and $v_{y_4=1}$ can be assigned with different values by DPLL(T). However, if we look at the original program in Fig. 2, clearly the statements "x=1" and "y=1" are in the same basic block. For any execution, either these two statements are both executed, or neither. Corresponding to the Boolean formula (3), the Boolean variables $v_{x_4=1}$ and $v_{y_4=1}$ must be assigned with the same Boolean value. Similarly, the Boolean variables $v_{x_4=2}$ and $v_{y_4=2}$, the Boolean variables $v_{x_4=3}$ and $v_{y_4=3}$ must be assigned with the same Boolean values, too. This is an important knowledge about the control flow of the program, which is, however, neglected by the SMT solver.

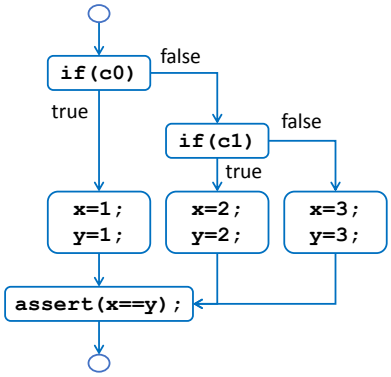

Figure 3: Control flow graph of the example

The second knowledge is about the execution of multiple blocks. The example program has three blocks (at line 2, 5 and 7), with conditions of c_0, $\neg c_0 \wedge c_1$, and $\neg c_0 \wedge \neg c_1$, respectively. Apparently, their conditions are mutually exclusive, hence only one of these blocks can be executed in one program execution. This important knowledge is also neglected by the SMT solver. For example, assume the c_0 is *true*, only statements "x=1", "y=1", and "assert(x==y)" are executed, the program's correctness is relevant to these three statements only. For the Boolean formula (3), the first two clauses encode the c_0 branch, and the last six clauses encode the $\neg c_0$ branch. We expect the last six clauses need not be considered in DPLL(T), as soon as v_{c_0} is assigned *true*. The actual situation is that the third and fourth clauses can be dropped from DPLL(T) (since they are satisfied), whereas the last four clauses cannot. The SMT solver still makes some efforts to consider different assignments of Boolean variables in these clauses (i.e., to explore the corresponding blocks) in the subsequent search process.

3.2 Applying Control-Flow Knowledge Makes Great Gains

Considering a basic block with n statements, let Φ be the formula that encodes this block, and $\mathcal{B}(\Phi)$ the Boolean abstraction of Φ, there are at least n Boolean variables in $\mathcal{B}(\Phi)$. DPLL(T) treats all these variables as independent individuals, and needs explore their 2^n assignments. In contrast, with the control-flow knowledge about the execution of statements in the same block, only 2 assignments of these n variables (both *true* or both *false*) need be explored.

Moreover, considering a program with m mutually exclusive blocks. Without the control-flow knowledge, the SMT solver needs to explore all combinations of these m blocks. The search space, in this case, is the Cartesian product of these m blocks. In contrast, with the control-flow knowledge about the execution of multiple blocks, each time we explore one block only. The search space is the addition of these m blocks. The search space is thus greatly reduced.

4 CONTROL FLOW-GUIDED SMT SOLVING

To utilize the control-flow knowledge, one may consider adding constraints to the original formula. However, specifying these constraints may introduce additional variables and thus increases the

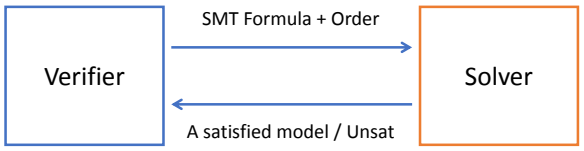

Figure 4: Overview of our approach

problem complexity. We have a simpler and smarter solution. The basic idea is to infer a variable order from the control-flow of the program, and then use this order to guide the search process of DPLL(T) (more clearly, by enforcing this order in *decide*() of Alg. 1).

Our approach can take full advantage of the built-in features of DPLL(T). It provides a lightweight technique for utilizing control-flow information. Fig. 4 shows an overview of our approach. At the verifier side, except the SMT formula that encodes the verification problem, a decision order on the variables is also generated (Section 4.1). This order records the control-flow information of the program. An SMT solving involves two steps: CNF conversion and DPLL(T) . We propose techniques to enhance both steps: a technique for guiding the DPLL(T) with an order (Section 4.2), and an enhanced CNF conversion technique (Section 4.3).

4.1 Decision Order

Let Φ be a formula, $\mathcal{B}(\Phi)$ the Boolean abstraction of Φ, and V the set of Boolean variables in $\mathcal{B}(\Phi)$. We distinguish the Boolean variables for branching conditions, and call them *branching variables*. Let $V^b \subseteq V$ be the set of branching variables in $\mathcal{B}(\Phi)$. We intend to define a *partial order* $\preccurlyeq$ over V. We call this order the *decision order*. Let v_1, v_2 be two variables in V, if $v_1 \preccurlyeq v_2$, we say v_1 is *prior to* v_2 in $\preccurlyeq$.

In the CFG of a program, a node d_1 is said to *dominate* another node d_2, if every path from `Entry` to d_2 must go through d_1. Considering the CFG in Fig. 3, the c_0 node dominates all nodes in the graph except of `Entry` and itself; the c_1 node dominates two of its successor nodes.

Remark that the branching conditions dominate the control flow of the program. We have the following heuristic.

Heuristic 1. *Branching variables are* prior to *all other variables in* V*, i.e.,* $\forall v_1 \in V^b, v_2 \in V \setminus V^b . v_1 \preccurlyeq v_2$.

Recall the problem discussed in Section 3.1: the statements "x=1" and "y=1" are in the same block, while the Boolean variables $v_{x_4=1}$ and $v_{y_4=1}$ can be assigned to different values. With Heuristic 1, this is not going to happen. If $v_{c_0} = true$, by Boolean propagation (on the first two clauses of the formula (3)), both $v_{x_4=1}$ and $v_{y_4=1}$ are forced to be *true*; if $v_{c_0} = false$, the first two clauses are trivially satisfied, and the Boolean variables $v_{x_4=1}$ and $v_{y_4=1}$ need not be considered in the subsequent process of DPLL(T).

To further define the order among branching variables, we have the following heuristic:

Heuristic 2. *Given two branching variables* $v_1, v_2 \in V^b$, v_1 *is* prior to v_2*, if* v_1 dominates v_2.

The underlying principle of Heuristic 2 is straightforward. Considering the CFG in Fig. 3, if c_0 is *true*, the whole *false* branch of c_0, no matter c_1 holding or not, can be excluded from the consideration. In contrast, deciding a value of c_1 cannot drop any branch of c_0.

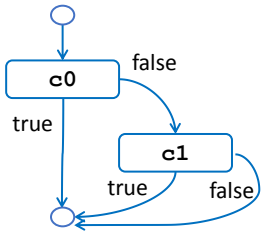

Figure 5: Branching graph

This paper consider loop-free programs only. Thus there can never be two nodes d_1 and d_2 in the CFG, such that both $d_1 \preccurlyeq d_2$ and $d_2 \preccurlyeq d_1$. In other words, the induced order by Heuristic 2 is a partial order.

4.1.1 Inferring the Order. The control-flow graph has all the information that we need. Inferring the decision order from the CFG of a given program is quite easy.

Given a CFG, we construct a so-called *branching graph* by eliminating all statements information but keeping the branching conditions in the graph. Let d be a node, except of `Entry` and `Exit`, of the CFG, we use $label(d)$ to denote the statements labeled on d, $parent(d)$ the parent nodes of d, and $child(d)$ the child nodes of d. If the last statement of $label(d)$ is a branch-statement, we delete all information in $label(d)$ but keep the branching condition. If the last statement in $label(d)$ is not a branch-statement, indicating that d has only one child node, we connect all its parent nodes to this single child node, and then delete d.

The resulting branching graph is a directed graph where nodes are branching conditions, and edges are control flows. All control flows are kept in the branching graph. The branching graph of the motivating example is shown in Fig. 5. Comparing the branching graph (in Fig. 5) to the CFG (in Fig. 3), four nodes are deleted, the labels of two nodes are changed. Note that the Boolean values associated with edges are all kept.

Apparently, the branching graph induce a partial order that agrees with Heuristic 2. We use this graph to guide the search of DPLL(T).

4.1.2 Storing the Order. We design a mechanism to implicitly store the branching graph. The edges of the graph are recorded in the identifiers of variables, which won't affect the semantics of the formula. As a result, when an SMT solver is invoked, the only input is the SMT formula file (in SMT-Lib-v.2.0 format). No additional file need be provided.

All condition variables are indexed by the order of their appearances in the program. Let v be a condition variable. The identifier of v is a series of numbers, separated by "_", with the first number being the index of v, and others being indexes of v's parents in the branching graph. For example, the condition variable v_{c_0}'s identifier is "0", indicating that c_0 is indexed by 0, and has no parent; the condition variable v_{c_1}'s identifier is "1_0", indicating that c_1 is indexed by 1, and its single parent is indexed by 0 (i.e. c_0).

Algorithm 2 *decide*()

Input: The current node *cur* in the branching graph
Output: An unassigned variable v and a Boolean value *value*

```
1: if cur == Exit then
2:     (v, value) = decide_nonbranching_variables();
3: while var(cur) is an assigned variable do
4:     let t be the assigned value to cur;
5:     cur = next(cur, t);
6: (v, value) = (var(cur), rand(0, 1));
7: return (v, value);
```

Algorithm 3 *rewrite_ite*()

Input: A SMT formula Φ
Output: A rewritten formula Ψ without *ite* functions

```
1: Ψ = Φ;
2: let ites be the set of ite-terms in Φ;
3: for each ite ∈ ites do
4:     let g, l, r be three real parameters of ite;
5:     replace ite in Ψ with (g → rewrite_ite(l)) ∧ (¬g → rewrite_ite(r));
6: return Ψ;
```

When an SMT solver is invoked, it first parses the SMT formula, and restore the branching graph from the variables' identifiers. Then it uses this branching graph to guide its DPLL(T) procedure.

4.2 Control Flow-Guided DPLL(T)

To guide the SMT solving, we enforce a decision order in the *decide*() operator of DPLL(T).

Our implementation of *decide*() is shown in Alg. 2. Assume there is a branching graph, and *cur* is its current node that represents the last assigned branching variable. Let d be a node, and t a Boolean value, we provide two methods for manipulating the branching graph: $var(d)$ returns the branching variable that d represents, and $next(d, t)$ returns the next node following the t-edge of d. If *cur* is the `Exit` node, indicating that all of the branching variables have been assigned to a value, the algorithm relies on the method *decide_nonbranching_variables*() to select the next variable and decide its value, which implements the default branching heuristics in conventional SMT solver [3, 15]. Otherwise, if the variable that *cur* represents has been assigned a value, the algorithm follows its value and move to the next node. This moving process repeats until we get an unassigned branching variable. Then we return this variable and a randomly decided value.

Moreover, the *resolve_conflict*() method in Alg. 1 needs be slightly modified. In case of backtracking, if the target variable is a branching variable, we need to correspondingly modify *cur* to the node that represents the backtracked variable and flip its assigned value.

4.3 Enhanced CNF Conversion

Recall the problem about the execution of mutually exclusive blocks (see Section 3.1). After the CNF conversion, the conditions of blocks are divided into parts. Even two blocks are mutually exclusive, the SMT solver cannot drop the opposite one.

To solve this problem, we need to enhance the CNF conversion procedure. For the motivating example, the $CNF(Enc)$ is expected to be the conjunction of the following clauses:

$$\begin{array}{ll} c_0 \rightarrow x_4 = 1 & c_0 \rightarrow y_4 = 1 \\ \neg c_0 \wedge c_1 \rightarrow x_4 = 2 & \neg c_0 \wedge c_1 \rightarrow y_4 = 2 \\ \neg c_0 \wedge \neg c_1 \rightarrow x_4 = 3 & \neg c_0 \wedge \neg c_1 \rightarrow y_4 = 3 \end{array} \quad (4)$$

Compared to (2), which is obtained by the conventional approach, the block conditions are specified in (4) as the premises of clauses. As a result, if the condition predicate of one block is *true*, by mutual exclusion of block conditions, the clauses corresponding to other blocks are trivially satisfied. In the above CNF formula (4), the first two clauses represent the c_0 branch, and the last four clauses represent the $\neg c_0$ branch. Assume v_{c_0} is *true*, thus $\neg v_{c_0} \wedge c_1$ and $\neg v_{c_0} \wedge \neg c_1$ are *false*, the last four clauses are hence trivially satisfied. The atoms in these four clauses, for instance, $x_4 = 2$, $y_4 = 2$ and so on, need not be considered again in DPLL(T). In this way, we avoid DPLL(T) to explore the $\neg c_0$ branch.

The key to getting the above CNF is the *ite* rewriting procedure, which rewrites *ite* terms into those with only Boolean connectives. Remark that all branching conditions of the verification condition formula are stored in the *ite* terms. The conventional rewriting algorithm breaks the nested structures of *ite* terms, and thus splits the block conditions into parts. We want a new *ite* rewriting algorithm that keeps the block condition as a whole.

Our new *ite* rewriting algorithm is shown in Alg. 3. The algorithm finds all *ite* terms in Φ. Each *ite* term is a ternary function with three parameters. We store these three parameters in g, l and r, respectively. The semantics of the *ite* function is that if g is true, it returns l, and otherwise returns r, which is equivalent to $(g \rightarrow l) \wedge (\neg g \rightarrow r)$. Note that l and r may also contain *ite* terms. The algorithm need be applied to l and r recursively. Finally, Ψ contains no *ite* term. Our algorithm can collaborate with the Tseitin's algorithm directly since the rewritten *ite* formulas are already in the form of a conjunction of clauses (Section 3).

Comparing our *ite* rewriting algorithm to the conventional one, the conventional rewriting algorithms introduce an auxiliary variable for each *ite*-term. The most benefit of the conventional approach is that all generated clauses contains only two literals for generating *binary clauses* [41] as more as possible. Binary clauses are good for Boolean propagations. However, the main problem here is not Boolean propagation but the theory propagation. With our CNF formula, there are some results that can be forced by applying Boolean propagation only. However, With their CNF formula, in many cases, must the theory solver be involved can we deduce the same result. Moreover, with our enhanced approach, the number of clauses is fewer. Besides, our algorithm needs not to introduce auxiliary variables for each *ite* term. The total number of variables is also reduced.

5 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

In this section, we discuss the implementation and evaluation of our control flow-guided SMT solving tactic. During the evaluation, we mainly consider the following research questions:

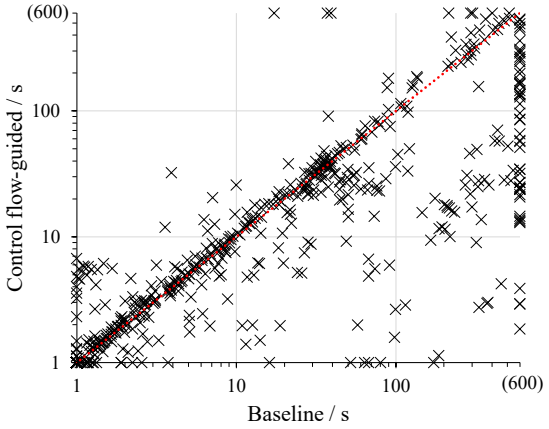

Figure 6: SMT solving time on all benchmarks.

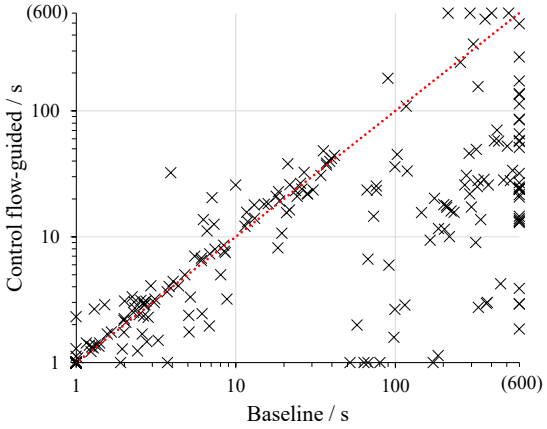

Figure 7: SMT solving time on *satisfiable* benchmarks.

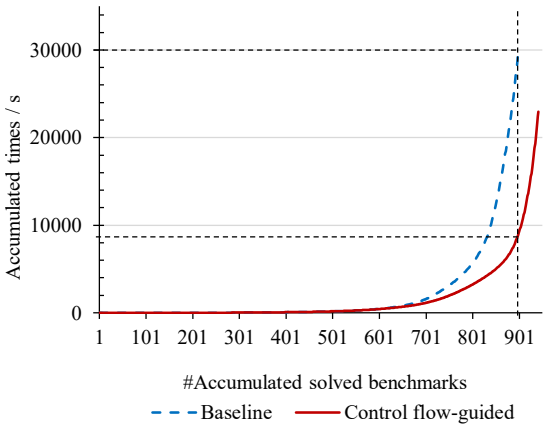

Figure 8: Accumulated SMT solving time on all benchmarks.

RQ1. How can we evaluate the performance of our tactic credibly?

RQ2. How does the performance vary between our tactic and the original method on credible benchmarks?

RQ3. Does our tactic affect the performance of the SMT solver in a way as we expected?

5.1 Implementation

We implement our control flow-guided SMT solving tactic (presented in section 4) in CBMC [11, 27] and Z3 [15]. CBMC is a well-known bounded model checking tool for C/C++ programs and Z3 is an efficient SMT solver. Both CBMC and Z3 are open-source projects. In our implementation, CBMC is responsible for generating SMT formulae that encode the verification conditions of the programs. Our decision order inferring algorithm is implemented in CBMC, since the control-flow information can be directly obtained after CBMC parses the programs. Our control flow-guided DPLL(T) algorithm and the enhanced CNF conversion algorithm are implemented in Z3. The SMT files transferred between CBMC and Z3 are in the SMT-LIB-v.2.0 format.

5.2 Experiment Setup

All experiments are conducted on the ReachSafety benchmarks in SV-COMP'17. This benchmark set contains up to 2897 C programs. All these programs have already been preprocessed for verification.

We use CBMC to generate SMT formulas from the benchmark programs. Let k be the unroll bound of CBMC. With different values of k, CBMC generates different SMT formulas. We ask the CBMC to generate as many as possible SMT formulas within 5 minutes. If the unroll bound reaches 2000, the generation process also stops. Consider a falsified program, let k^* be the minimal value of k such that a counterexample can be revealed from the program. Then, for all generated SMT formulas for this program, they are unsatisfiable for $k < k^*$, and satisfiable for $k \geq k^*$. In order to balance the number of satisfiable and unsatisfiable instances, we drop some SMT instances with a rather small k. Finally, excluding the exceptional cases (timeout, internal error, etc), there are totally 1411 programs with which CBMC successfully generates SMT instances. All of these SMT files are plain text files in SMT-LIB-v.2.0 format. The largest SMT files occupy hundreds of MB.

We compare the performance of our control flow-guided SMT solving tactic to that of the default heuristic in Z3. The original Z3 with the default heuristic is considered as the baseline. The plus of Z3 and our tactic is called Z3+. Timeout for each instance is set to 10 minutes. Note that some preamble tactics (for example, the simplifying tactic, the variable eliminating tactic, etc) are realized in Z3, with which some of these SMT instances can be immediately solved, without calling the DPLL(T) procedure. There are also some instances that are too hard to be solved by either Z3+ or the baseline in the time limit. We drop the above two kinds of exceptional benchmarks. Finally, we got 948 SMT instances, where 314 instances are satisfiable and 634 ones are unsatisfiable. We believe these benchmarks are sufficient to evaluate the performance of our tactic credibly.

5.3 Experimental Results

Figure 6 show the SMT solving time of the baseline and our control flow-guided tactic on all benchmarks. The figure demonstrates that our tactic is superior to the baseline on the majority of the

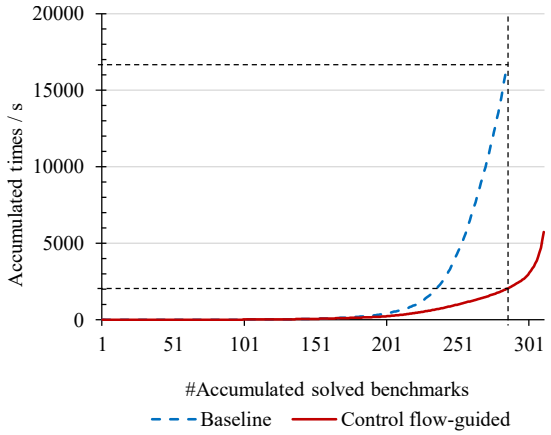

Figure 9: Accumulated SMT solving time on *satisfiable* benchmarks.

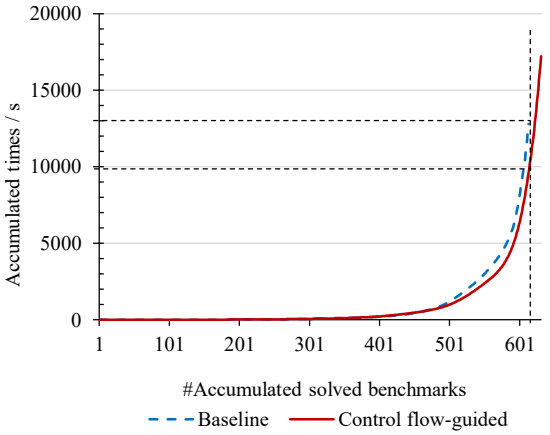

Figure 10: Accumulated SMT solving time on *unsatisfiable* benchmarks.

benchmarks. Especially, our tactic can speed up the SMT solver by orders of magnitude on about a quarter of the benchmarks, and there are a very little of benchmarks on which our tactic is notably inferior to the baseline. The accumulated SMT solving time for solved benchmarks is shown in Figure 8. In general, our tactic uses only 30.0 percents of time of the baseline to resolve the same number of the benchmarks.

We also present the SMT solving time of the satisfiable benchmarks in Figure 7. On satisfiable benchmarks, our tactic is superior to the baseline for most of the benchmarks. There are only a handful of benchmarks that cost our tactic more time. Besides, nearly half of the benchmarks are sped up by orders of magnitude. The accumulated SMT solving time for all satisfiable benchmarks is shown in Figure 9. Our tactic costs only 12.2 percents of time of the baseline to resolve the same benchmark group.

The accumulated SMT solving time for solved unsatisfiable benchmarks is shown in Figure 10. The curves for both the baseline and our tactic are substantially identical. Our tactic saved about 25.2 percents of the SMT solving time for this group of benchmarks.

Summary of the experimental results for SMT solving time is presented in Table 1. The *CPU time* is the total running time of each tool on the whole benchmark set. Our tactic costs half the *CPU time* to finish all the benchmarks and only spends a quarter of *CPU time* to finish the satisfiable benchmarks compared to the baseline. The *score time* is the accumulated time for solving a certain quantity of benchmarks, where the quantity is equal to the maximal number of benchmark that the baseline can solve. All of the *score times* are indicated by the straight dash line in the curve diagrams like Figure 8. The ratio of the *score time* of the baseline and our tactic indicates the speed-up times of our tactic for solving the same group of benchmarks. To summarize, our tactic speeds up the solver by 3.34 times for all benchmarks and 8.22 times for satisfiable benchmarks. Also, the number of timeout cases with our tactic is much fewer than that with the baseline. In conclusion, our control flow-guided SMT solving tactic can significantly improve the performance of the SMT solver.

Table 1: Summary of experimental result

Sub	CPU Time / s			Score Time / s			#Timeout	
	z3	z3+	Lift	z3	z3+	Lift	z3	z3+
sat	34573	8135	**4.25**	16573	2016	**8.22**	30	**4**
unsat	25994	19619	**1.32**	12794	9567	**1.33**	22	**4**
all	60568	27754	**2.18**	29368	8803	**3.34**	52	**8**

5.4 Result Analysis

The experimental results illustrate remarkable improvements of our tactic over the baseline. Also note that the improvement of our tactic on unsatisfiable benchmarks is not as significant as that on satisfiable benchmarks. In other words, the utility of our tactic is limited for the unsatisfiable instances. Recall that DPLL(T) is basically an exhaustive search procedure. For an unsatisfiable formula, no matter our tactic or the baseline they have to explore the whole search space to prove its unsatisfiability. The efficiency of the control flow-guided approach is thus greatly limited. In fact, the unsatisfiable cases are hard to be optimized for most of the existing heuristic methods.

Furthermore, during the DPLL(T) search procedure, the algorithm stops as long as a satisfiable variable assignments is found. If the current variable assignments conflict, a conflict clause is learned and the search procedure backtracks. To a certain extent, the number of conflict clauses indicates the try times in the search procedure of DPLL(T) . Figure 11 shows the number of conflict clauses learned during the DPLL(T) . As we can see, on the majority of the benchmarks, the number of the conflict clauses learned with our tactic is fewer than that with the baseline, especially for the satisfiable benchmarks. These results indicate that our tactic can guide DPLL(T) to find a satisfiable solution with fewer try times in the search procedure.

5.5 Threats to Validity

The main *internal* threats to the validity of our approach are that whether the performance improvements are mainly due to our control flow-guided tactic, and that whether the implementation

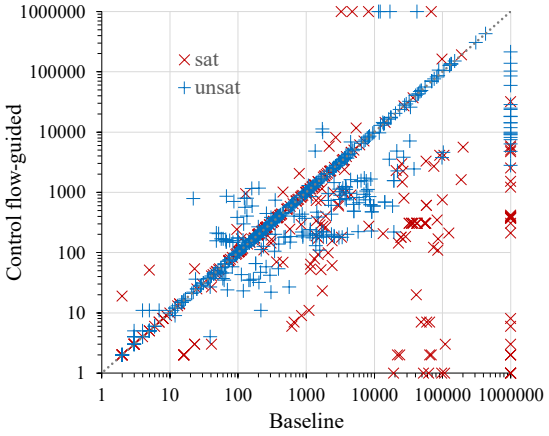

Figure 11: #Conflict clauses learned during DPLL(T) .

of our tactic is credible. Firstly, we only implemented two algorithms in Z3, i.e., the branching heuristic and the enhanced CNF conversion algorithm. They are simple and barely coupled with other modules. Meanwhile, we have implemented our algorithm as clearly as we can. Secondly, the adding of the control-flow information won't affect the performance of the original solver, since we implicitly record the decision order in the SMT file by naming the SMT variables in a special fashion, which doesn't have any effect on the semantics of the SMT formula. Last but not the least, we conduct experiments on a large number of benchmarks from SV-COMP'17. The experimental results show the remarkable performance of our technique. Additionally, the analysis of the number of conflict clauses further demonstrates that our tactic does accelerate the search procedure, which conforms to our expectations. We are thus confident in the effectiveness of our tactic.

Another threat to the validity of our approach is that whether our approach can be generalized to other solvers other than Z3. Actually, our tactic is based on the DPLL(T) framework, it has nothing to do with the specific features of the Z3 solver itself. Thus, our tactic can be implemented in any DPLL(T) -based SMT solver for program analysis.

One more threat is that whether our approach can be generalized to other program analysis techniques, except BMC. The answer is also yes. Actually, we use CBMC in our experiments only to generate SMT instances, and nothing more. Our tactic can be easily applied to other program analysis techniques only if they are SMT-based, i.e., they encode the verification conditions as the SMT formulas and then rely on an SMT solver to solve these formulas.

6 RELATED WORK

There is a large body of work on improving the performance of the DPLL-based constraint solvers, like SAT and SMT solvers. Below, we compare our approach with the most closely related works. Since our approach is based on the branching heuristic (also named the *decision ordering*) of the DPLL backtrack search procedure, we discuss this aspect first.

6.1 Branching Heuristics

One of the most famous branching heuristics is VSIDS [30, 32]. It increases the ranking values of each literal in a newly inferred conflict clause, and also decays these values periodically. The ordering is based on these ranking values. There are also other branching heuristics utilize the information from the backtrack search procedure, like MOM heuristic [20, 38], Jeroslow-Wang heuristics [25], literal count heuristics [31], etc. However, all of these branching heuristics are designed for the general satisfiability problems, which can only utilize the information from the DPLL procedure, and almost do not care about the domain-specific knowledge from original problems modeled by the satisfiability problems. Our semantic branching heuristic is combined with one of the most efficient heuristics (VSIDS-based), but focus on the satisfiability problems derived from the program analysis problem. With the domain-specific knowledge of program analysis, our heuristic can lead to more efficient constraint solving.

There is also some work on refining the decision ordering by the domain-specific knowledge of circuits or models. The most closely related work is [44], where Yin et al. refine the decision ordering for the satisfiability problem from transition system by giving the higher priority to the transition variables over other variables. Besides, all of the transition variables are ordered by the transition relations in the models. As we discussed in section 1, the program is more complex than the general transition systems. Furthermore, this heuristic can hardly work since all of the program variables are transition variables and the DPLL decision procedure is performed on the predicates over the program variables instead of on the program variables directly. Moreover, there are some other related work refining the decision ordering by the knowledge of the transition systems. In [43], Wang et al. identify the important variables in the *unsat*-core form the previous unsatisfiable BMC instances and give priority to these important variables in decision ordering on the current instance. Shtrichman and Ofer [40] suggests a static predetermining order following a forward or backward breadth-first search on the variable dependency graph of the transition system. In [23], Gupta et al. explore implications learned from the circuit structure represented by BDDs to help the SAT solving. Since there is less structure information in a CNF formula, Ostrowski et al. [35] suggests recovering and exploiting structural knowledge by a set of equations form initial SAT instance to eliminate clauses and variables. All of the work are based on the domain-specific knowledge of circuits or models, instead of the programs.

6.2 Utilizing Control-Flow Information

Our tactic is based on utilizing the control-flow information of the program. We also compare our approach with the related work on utilizing the control-flow information for program analysis. In [29], Leino et al. split the monolithic VC of a program analysis procedure into the conjunction of several smaller VCs according to the control-flow information. The partition results in several smaller SMT query which could be solved more efficiently than the original one. Cimatti et al. [10] partitions the abstraction problem into the combination of several smaller abstraction problems by exploiting the structure of the program. These approaches improve the performance by the heuristic of partition the monolithic problem into several smaller

problems utilizing the control-flow information. Our tactic also utilized this heuristic since giving priority to the condition variable is similar to splitting the whole program into several program paths and then solving the path formula one by one.

Symbolic execution generates a verification condition for each path of the program. For different paths (of the control flow of the program), it generates different SMT formulas. In this respect, one may say that the symbolic execution also utilizes the control flow information of programs. However, symbolic execution utilizes this information at the level of VC generation, while ours does at the level of the SMT solver. With our approach, only one SMT formula that encodes verification conditions of all paths of the program is generated. Our tactic can use the many built-in features of the SMT solver, including the conflict clause learning, value propagation and so on. As a result, the intermediate results among verifying the many paths of the program can be easily and automatically shared. Belt et al. [2] and Feist et al. [19] proposed to utilize the control-flow information and the domain-specific knowledge of symbolic execution to reduce the times of the SMT query, respectively. For the same reason, these work are also different from ours.

6.3 Theory-Aware Approach

The optimization of our tactic is based on pruning the conflict assignments early. Some of the related work based on the theory-aware approach also optimize the constraint solving procedure in this way. The theory-aware approaches discover the constraints from the underlying theory by a lightweight method and utilize these constraints to help to prune the conflict assignments in DPLL procedure. Berzish et al. [3] proposed a theory-aware branching heuristic which prioritizes simpler branches over more complex ones in string solvers. They also proposed a theory-aware case-split to circumvent mutually exclusive assignments by the structure of string theory literals. Goldwasser et al. [21] proposed a theory-aware branching heuristic for linear reals arithmetic based on a geometric analysis over the linear constraints. The heuristic suggests the values, which is consistent with the current partial assignment, for the unassigned predicates of linear constraints. Bruttomesso et al. [9] utilize the structural information like equalities and arithmetic functions to help to reason at a higher level of abstraction within the theory of bit-vectors. All of these theory-aware approaches focus on the specific theory, such as string theory, linear reals arithmetic theory, and bit-vector theory. However, our tactic is a high-level approach based on refining branching heuristic of DPLL procedure, which is independent of theories. In [28], Kuehlmann et al. use circuit specific knowledge to guide the search of SAT solving and help the solver reasoning on specific logic. This work is only suitable for circuit problems. Besides the DPLL-based solver, the other constraint solver can also utilize the theory-aware approach to improve the performance significantly. For example, Hooimeijer et al. [24] proposed a lazy backtracking search algorithm for solving the regular expression constraints efficiently by a *follow graph* and a constraint-path map generated from the constraint system. This work is specifically for solving the regular expression constraints.

7 CONCLUSION AND FUTURE WORK

In this paper, we present control flow-guided SMT solving tactic utilizing the control-flow information of the programs to refine the DPLL(T) search procedure in SMT solvers. In our tactic, the search space is reconstructed continuously and dynamically by the control-flow information and thus a large number of redundant search paths can be pruned. We implement our tactic in the modern SMT solver Z3, and compare our tactic with the default heuristic in Z3. The experiments on SV-COMP'17 benchmarks have shown that our control flow-guided SMT solving tactic can significantly speed up the SMT solving. Especially, our tactic achieves orders of magnitude improvements on satisfiable benchmarks.

In algorithm 2, after a decision variable is selected, its value is randomly determined. There also exist some strategies for deciding a variable's value in the existing SAT/SMT solvers. Again, these strategies are not domain specific. We are planning to study to use the program information to guide the SMT solver for determining the variables' values.

REFERENCES

[1] Mike Barnett, Manuel Fähndrich, K Rustan M Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and verification: the Spec# experience. *Commun. ACM* 54, 6 (2011), 81–91.

[2] Jason Belt, Xianghua Deng, et al. 2009. Sireum/Topi LDP: a lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 355–364.

[3] Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. 2017. Z3str3: A string solver with theory-aware branching. *arXiv preprint arXiv:1704.07935* (2017).

[4] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009.* IEEE, 25–32.

[5] Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2018. A unifying view on SMT-based software verification. *Journal of Automated Reasoning* 60, 3 (2018), 299–335.

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 193–207.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2009. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), 131–153.

[8] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Van Rossum, Stephan Schulz, and Roberto Sebastiani. 2005. The mathsat 3 system. In *International Conference on Automated Deduction*. Springer, 315–321.

[9] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. 2007. A Lazy and Layered SMT ($\mathcal{BV}$) Solver for Hard Industrial Verification Problems. In *International Conference on Computer Aided Verification*. Springer, 547–560.

[10] Alessandro Cimatti, Jori Dubrovin, Tommi Junttila, and Marco Roveri. 2009. Structure-aware computation of predicate abstraction. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009.* IEEE, 9–16.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 238–252.

[13] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 25–35.

[14] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[16] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

[17] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software verification using k-induction. In *International Static Analysis Symposium*. Springer, 351–368.

[18] Herbert Enderton and Herbert B Enderton. 2001. *A mathematical introduction to logic*. Academic press.

[19] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2016. Guided dynamic symbolic execution using subgraph control-flow information. In *International Conference on Software Engineering and Formal Methods*. Springer, 76–81.

[20] Jon William Freeman. 1995. *Improvements to propositional satisfiability search algorithms*. Ph.D. Dissertation. University of Pennsylvania Philadelphia, PA.

[21] Dan Goldwasser, Ofer Strichman, and Shai Fine. 2008. A theory-based decision heuristic for DPLL (T). In *Formal Methods in Computer-Aided Design, 2008. FMCAD'08*. IEEE, 1–8.

[22] Susanne Graf and Hassen Saïdi. 1997. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification*. Springer, 72–83.

[23] Aarti Gupta, Malay Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. 2003. Learning from BDDs in SAT-based bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*. ACM, 824–829.

[24] Pieter Hooimeijer and Westley Weimer. 2010. Solving string constraints lazily. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 377–386.

[25] Robert G Jeroslow and Jinchang Wang. 1990. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence* 1, 1-4 (1990), 167–187.

[26] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[27] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

[28] Andreas Kuehlmann, Malay K Ganai, and Viresh Paruthi. 2001. Circuit-based Boolean reasoning. In *Proceedings of the 38th annual Design Automation Conference*. ACM, 232–237.

[29] K Rustan M Leino, Michał Moskal, and Wolfram Schulte. 2008. Verification condition splitting. *Submitted manuscript, September* (2008).

[30] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. 2015. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *Haifa Verification Conference*. Springer, 225–241.

[31] Joao Marques-Silva. 1999. The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*. Springer, 62–74.

[32] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*. ACM, 530–535.

[33] Greg Nelson and Derek C Oppen. 1980. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364.

[34] Alexandru Nicolau. 1988. Loop quantization: A generalized loop unwinding technique. *J. Parallel and Distrib. Comput.* 5, 5 (1988), 568–586.

[35] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. 2002. Recovering and exploiting structural knowledge from CNF formulas. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 185–199.

[36] Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. 2007. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal* 46, 2 (2007), 265–288.

[37] M Prasanna, S Sivanandam, R Venkatesan, and R Sundarrajan. 2005. A survey on automatic test case generation. *Academic Open Internet Journal* 15, 6 (2005).

[38] Daniele Pretolani. 1996. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1996), 479–498.

[39] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using Gröbner bases. *ACM SIGPLAN Notices* 39, 1 (2004), 318–329.

[40] Ofer Shtrichman. 2000. Tuning SAT checkers for bounded model checking. In *International Conference on Computer Aided Verification*. Springer, 480–494.

[41] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.

[42] G Tseitin. 1968. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic* (1968).

[43] Chao Wang, HoonSang Jin, Gary D Hachtel, and Fabio Somenzi. 2004. Refining the SAT decision ordering for bounded model checking. In *Proceedings of the 41st annual Design Automation Conference*. ACM, 535–538.

[44] Liangze Yin, Fei He, and Ming Gu. 2013. Optimizing the sat decision ordering of bounded model checking by structural information. In *Theoretical aspects of software engineering (tase), 2013 international symposium on*. IEEE, 23–26.