Leveraging Control Flow Knowledge in SMT Solving of Program Verification

JIANHUI CHEN and FEI HE, School of Software, Tsinghua University and Key Laboratory for Information System Security, MoE and Beijing National Research Center for Information Science and Technology, China

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, current SMT solvers are designed for the general purpose of constraint solving. Lots of useful knowledge of programs cannot be utilized during SMT solving. As a result, the SMT solver may spend much effort to explore redundant search space. In this article, we propose a novel approach to 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 significant improvements of our approach.

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

Additional Key Words and Phrases: Program verification, satisfiability modulo theory, control-flow graph

ACM Reference format:
Jianhui Chen and Fei He. 2021. Leveraging Control Flow Knowledge in SMT Solving of Program Verification. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 41 (May 2021), 26 pages.
https://doi.org/10.1145/3446211

1 INTRODUCTION

Satisfiability modulo theories (SMT) are applied in diverse software engineering technologies, such as software verification [4, 22], static program analysis [11, 38], symbolic execution [26], test-case generation [39], and so on. A powerful SMT solver is a crucial factor in improving the efficiency of these technologies. However, SMT solvers are designed for the general purpose of constraint solving [15]. Domain-specific knowledge cannot be efficiently utilized in current SMT solvers. On the other hand, domain-specific knowledge may be quite useful for pruning the search space, and thus improving the efficiency of SMT solvers.

This work was partially funded by the NSF of China (No. 61672310 and No. 62072267), the National Key R&D Program of China (No. 2018YFB1308601), and the Guangdong Science and Technology Department (No. 2018B010107004).
Authors' address: J. Chen and F. He (corresponding author), School of Software, Tsinghua University, Key Laboratory for Information System Security, MoE, Beijing National Research Center for Information Science and Technology, Beijing, China; emails: chenjian16@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2021/05-ART41 $15.00
https://doi.org/10.1145/3446211

A control-flow graph (CFG) is the graph representation of programs. It is simple but informative. Lots of domain-specific 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. Furthermore, 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 the redundant search space, which could be pruned if the control-flow knowledge was known.

One may consider encoding the control-flow knowledge into the SMT constraints (Section 5). However, encoding this knowledge needs to explicitly specify program executions, which may yield an extremely large number of constraints, and increase the complexity of the SMT solving (Section 6.6). This article proposes a simpler but smarter solution for utilizing control-flow knowledge in program verification (Section 4). The basic idea is to use a *decision order* and a *decision value mapping* to guide the SMT solving. The decision order decides at each search step which variable should be taken for assignment, and the decision value mapping provides a prioritized value for each decision variable. They together control the search direction in the SMT solving. We propose to infer the decision order and the decision value mapping from the control-flow graph of the program.

We propose two heuristics for arranging the decision order (in Section 4.1). First, we recognize the importance of branching conditions that dominate the control flows of the program, and assign the corresponding variables (called *branching* variables) higher priorities in the decision order. Second, we follow the domination relation of the control flow graph to arrange the order of branching variables. We also devise heuristics for prioritizing the decision values of branching variables (in Section 4.2). Regarding the program verification as a searching process on the control-flow graph, the decision value of a branching variable determines which of its child nodes should be explored next. We propose two heuristics for evaluating each of its child nodes by computing the length of its shortest path and the sum length of all its paths, respectively. Finally, the inferred decision order and the prioritized decision values are all enforced in the search process of the SMT solving (in Section 4.3). In this way, we get a lightweight technique to utilize control-flow knowledge in SMT solving. It does not need a lot of implementation in the SMT solver, and can take full advantage of the built-in features of the solver.

Moreover, we propose an enhanced *conjunction normal form* (CNF) conversion procedure (in Section 4.4) to collaborate with the above approaches. After the program is encoded into the SMT formula, the if-then-else (*ite*) terms keep some 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. Moreover, with our enhanced approach, the generated CNF contains fewer clauses and fewer variables. The efficiency is further improved.

To the best of our knowledge, this is the first attempt at improving SMT solving by using domain-specific knowledge of programs. There are some works on domain knowledge-guided SAT solving [5, 42, 45, 46]. In [46], the structure information of transition systems is utilized to guide the SAT solvers. In [42], the variable dependency information of the model is utilized to guide the SAT solvers. Wang et al. [45] 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 article, we consider SMT solving

for program verification. The domain knowledge is different, and the application target is also different.

We implement a prototype of our approaches on top of *CBMC* and *Z*3. We conducted extensive experiments on programs in SV-COMP'17.[1] Totally, 948 SMT instances were generated from these programs and were used as the benchmark set. Experimental results show that the control-flow knowledge can significantly improve the efficiency of SMT solving. The average speed-up is 3.39 times for all instances, and 8.85 times for satisfiable instances.

In summary, our main contributions include:

- We propose a novel approach for leveraging control-flow knowledge in SMT solving. This approach is lightweight and can take full advantage of the built-in features of SMT solvers.
- We devise efficient heuristics for inferring the decision order and prioritizing the decision values from the control flows of the program.
- We propose an enhanced CNF conversion procedure, with which the control-flow information can be kept in the CNF formulas.
- We implement the prototype of our approach on top of *CBMC* and *Z*3, and conduct extensive experiments to evaluate its effectiveness and efficiency. Results show significant improvements of our approach.

This article is an extended and revised version of a preliminary conference paper [9]. Compared to [9], this article makes the following new contributions. First, in [9], only the decision order is proposed to guide the SMT solving. In this article, we further propose to guide the SMT solving using the decision values. Two efficient heuristics were devised for prioritizing the decision values of branching variables. The newly proposed techniques further enhance the previous control flow-guided SMT solving approach. Second, this article reports three new sets of experimental results, one for evaluating the heuristics for decision values; one for evaluating the different combinations of the proposed tactics; and the last for evaluating the explicit encoding approach.

The rest of this article is organized as follows: Section 2 introduces some background knowledge. Section 3 employs a simple example to motivate our approach. Section 4 presents our control-flow guided SMT solving approach. Section 5 presents a practical approach for encoding control-flow knowledge. Evaluation and experimental results are reported in Section 6. Related works are discussed in Section 7. Finally, Section 8 concludes this article.

2 PRELIMINARIES

2.1 Notations

In **first-order logic (FOL)**, a *term* can be a variable, a constant, or an n-ary function applied to n terms. An *atom* is $\top$, $\bot$, or an 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 formula Φ is said *satisfiable*, if there exists a model M such that $M \models \Phi$; and *valid* if $M \models \Phi$ for any model M.

A first-order theory T is defined by a signature and a set of axioms. The signature defines a set of constants, functions, and predicate symbols allowed in T, and the axioms define the intended

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

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, a jump target starts a block, and a jump ends 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*. Two branches of the branch-statement are 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 [5], static analysis [38], abstract interpretation [11], predicate abstraction [4, 22], and the like, promote program verification to a practical technique for correctness assurance of programs. Loops are the main hurdle for the program verification. There are mainly two approaches for handling loops in a program: loop invariant [41] and loop unwinding [36]. 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. The automatic generation of loop invariants has been extensively studied, but the existing techniques are still not practical enough [1]. The more popular approach for handling loops is loop 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 [16], enhance this approach by enabling the correctness proving of programs.

This article assumes all programs have been processed by the loop invariant or loop unwinding technique. Thus, the programs are free of loops. The loop-free programs are converted to their *static single assignment* (SSA) forms [12]. With the SSA form, the correctness of a program can easily be encoded as a set of FOL formulas [17]. These formulas are called the verification condition of the program with respect to the desired property. The validity of the verification condition implies the correctness of the program.

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 functions, arithmetic, arrays, bit-vectors, and so on. Nelson and Oppen [35] proposed a combination method to deal with FOL formulas in multiple theories.

DPLL(T) extends the DPLL algorithm [13] 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 Figure 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 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

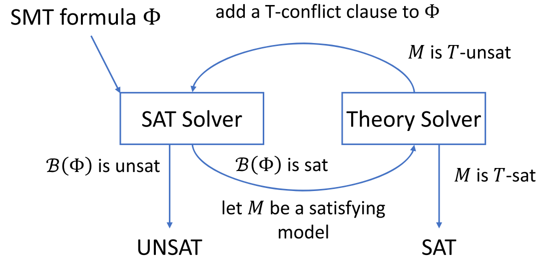


Fig. 1. DPLL(T).

ALGORITHM 1: DPLL-T(Φ)

Input: An SMT formula Φ
Output: SAT or UNSAT

1: **while** *true* **do**
2:     **while not** *propagate_and_check*() **do**
3:         **if** *decision_level* = 0 **then return** UNSAT;
4:         **else** *resolve_conflict*();
5:     **if not** *decide*() **then return** SAT;

the original formula with respect to its satisfiability. If $\mathcal{B}(\Phi)$ is unsatisfiable, so is Φ, but the reverse may not hold.

The high-level view of a practical DPLL(T) algorithm is shown in Algorithm 1. At this level, the algorithm is the same as that of a conflict-driven clause learning-based SAT solver [6]. 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 literals as many as possible. 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 adds them to the clause database. The method *decide*() decides the next unassigned Boolean variable and guesses 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 [7, 14, 32]. DPLL(T) is essentially a depth-first search algorithm. We may guide the searching process of DPLL(T) by enforcing a variable order and its values in the *decide*() method.

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 SMT solvers, but utilizing this knowledge can make great gains.

Consider a simple program shown on the left of Figure 2. Its main part is a two-tier nested if-statements. The control-flow graph of this program is shown in Figure 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 to be verified is that $x = y$ holds at the end of this program.

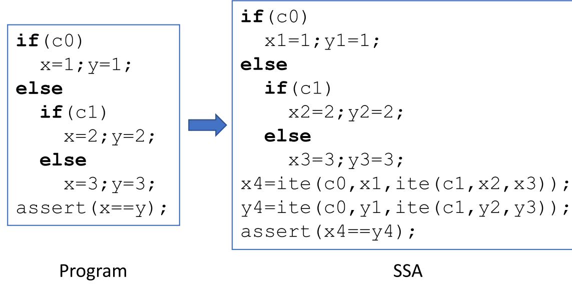


Fig. 2. Example.

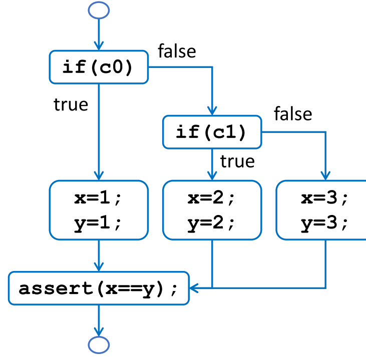


Fig. 3. Control flow graph of the example.

Verification Condition. The original program is first converted into the single static assignment (SSA) form, where for each assignment-statement, a new variable is introduced for the left-hand-side variable. After this conversion, there is at most one assignment for each variable. The SSA form of the example program is shown in the right of Figure 2. Note that $ite(bool, \cdot, \cdot)$ is a ternary function that returns its second or third argument depending on whether 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* is the correctness condition. For the motivating example, we have

$$\begin{aligned} Enc \equiv (x_1 = 1) \wedge (x_2 = 2) \wedge (x_3 = 3) \wedge (y_1 = 1) \wedge (y_2 = 2) \wedge (y_3 = 3) \\ \wedge (x_4 = ite(c_0, x_1, ite(c_1, x_2, x_3))) \wedge (y_4 = ite(c_0, y_1, ite(c_1, y_2, y_3))) \end{aligned} \tag{2}$$

and

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

Note that the above formulas can be simplified by variable elimination. The elimination is also performed by SMT solvers before the DPLL(T) procedure. For example, we can eliminate the variable x_1 in *Enc* by replacing it by 1 since we have the equation $x_1 = 1$. After being simplified, the *Enc* formula becomes

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

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 needs to be converted to *conjunctive normal form* (CNF). A conjunctive normal form is a conjunction of clauses, where each clause is a disjunction of literals. We usually write a clause as a logical implication for ease of understanding since it can be converted to a clause directly, e.g., $A_1 \wedge \cdots \wedge A_k \rightarrow B$ 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 this article, we also write a CNF as a set of clauses, and a clause as a set of literals.

Most SMT solvers adopt Tseitin's transformation method [44] to perform the CNF conversion, which adds a new variable for each subformula of the original formula. Considering the motivating example, the CNF of *Enc*, written *CNF*(*Enc*), is the conjunction of the following clauses:

$$\begin{array}{rlrl} c_0 \rightarrow x_4 = 1 & & c_0 \rightarrow y_4 = 1 \\ \neg c_0 \rightarrow x_4 = t_x & & \neg c_0 \rightarrow y_4 = t_y \\ c_1 \rightarrow t_x = 2 & & c_1 \rightarrow t_y = 2 \\ \neg c_1 \rightarrow t_x = 3 & & \neg c_1 \rightarrow t_y = 3 \end{array} \tag{3}$$

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

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

DPLL(T). 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

$$\begin{array}{rlrl} v_{c_0} \rightarrow v_{x_4=1} & & v_{c_0} \rightarrow v_{y_4=1} \\ \neg v_{c_0} \rightarrow v_{x_4=t_x} & & \neg v_{c_0} \rightarrow v_{y_4=t_y} \\ v_{c_1} \rightarrow v_{t_x=2} & & v_{c_1} \rightarrow v_{t_y=2} \\ \neg v_{c_1} \rightarrow v_{t_x=3} & & \neg v_{c_1} \rightarrow v_{t_y=3} \end{array} \tag{4}$$

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 (4), all variables are independent of 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 Figure 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 (4), the Boolean variables $v_{x_4=1}$ and $v_{y_4=1}$ must be assigned to the same Boolean value. Similarly, the Boolean variables $v_{x_4=2}$ and $v_{y_4=2}$, or the Boolean variables $v_{x_4=3}$ and $v_{y_4=3}$ must be assigned to the same Boolean values, too. This is important knowledge about the control flow of the program, which is, however, neglected by SMT solvers.

The second knowledge is about the execution of multiple blocks. The example program has three blocks (lines 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 SMT solvers. For example, if we assume that c_0 is *true*, only statements "$x = 1$", "$y = 1$", and "assert(x==y)" are executed, and the program's correctness is relevant to these three statements only. For the Boolean formula (4), the first two clauses encode the c_0 branch, and the last six clauses encode the $\neg c_0$ branch. We expect

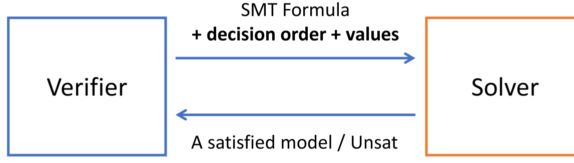


Fig. 4. Overview of our approach.

the last six clauses needn't be considered in DPLL(T) as soon as v_{c_0} is assigned to *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

Consider a basic block with n statements. Let Φ be the formula that encodes this block, and $\mathcal{B}(\Phi)$ the Boolean abstraction of Φ. Then there are at least n Boolean variables in $\mathcal{B}(\Phi)$. DPLL(T) treats all these variables as independent individuals, and needs to 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 (all *true* or all *false*) need to be explored.

Moreover, considering a program with m mutually exclusive blocks. Without the control-flow knowledge, SMT solvers need 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 sum of these m blocks, which is thus greatly reduced.

4 CONTROL FLOW-GUIDED SMT SOLVING

To utilize the control-flow knowledge, a natural idea is to encode it into the verification condition. However, specifying the control-flow knowledge may introduce a very large number of constraints and thus increase the complexity of the SMT solving (Section 5). In this section, we propose a *simpler but smarter* solution. The basic idea is to infer a variable order and a value mapping from the control-flow structure of the program, and then use the order together with the value mapping to guide the search process of DPLL(T) (more clearly, by enforcing this order and the mapped values in *decide*() of Algorithm 1).

Our approach can take full advantage of the built-in features of DPLL(T). It provides a light-weight technique for utilizing control-flow information. Figure 4 shows an overview of our approach. On the verifier side, besides the SMT formula, which encodes the verification problem, a decision order (Section 4.1) and a value mapping (Section 4.2) are also inferred. On the solver side, both the DPLL(T) (Section 4.3) and the CNF conversion (Section 4.4) are enhanced to make use of the control-flow knowledge.

4.1 Decision Order

Let Φ be a formula, $\mathcal{B}(\Phi)$ be the Boolean abstraction of Φ, and V be the set of Boolean variables in $\mathcal{B}(\Phi)$. We distinguish the Boolean variables of 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, and call it 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 $\preccurlyeq$.

ALGORITHM 2: *buildBranchingGraph*(G)

Input: A control-flow graph $G = \langle N, E \rangle$,where N are nodes and E are edges.
Output: A branching graph $G^B = \langle N^B, E^B \rangle$.

1: $G^B \leftarrow duplicate(G)$;
2: **for each** $d \in N^B \setminus \{\texttt{Entry}, \texttt{Exit}\}$ **do**
3:     **if** d is a branching node **then** ▷ keep this node
4:         $label(d) \leftarrow$ the branching condition;
5:     **else** ▷ delete this node
6:         $d' \leftarrow child(d)$;
7:         **for each** $d'' \in parent(d)$ **do**
8:             add an edge from d'' to d';
9:         delete the node d;
10: **return** G^B;

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 Figure 3, the c_0 node dominates all nodes in the graph except for `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 (4)), 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 Figure 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.

This article considers 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 statement information but keeping the branching conditions in the graph. The algorithm is presented in Algorithm 2. It traverses all nodes, except for `Entry` and `Exit`, in the CFG. Let d be the current node. We denote the statements labeled on d as $label(d)$, the parent nodes of d as $parent(d)$, and the child nodes of d as $child(d)$. If the last statement of $label(d)$ is a branch-statement, we delete all statements in $label(d)$ but keep the branching condition. If the last statement of $label(d)$ 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. In this way, we generate a branching graph of the program.

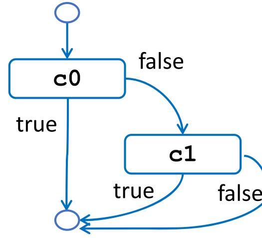


Fig. 5. Branching graph.

The resulting branching graph is a directed acyclic 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 Figure 5. Comparing the branching graph (in Figure 5) to the CFG in Figure 3, four nodes are deleted, and the labels of two nodes are changed. Note that the Boolean values associated with edges are all kept.

Apparently, the branching graph induces 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 store the branching graph implicitly. The edges of the graph are encoded 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 files need to be provided.

All branching variables are indexed by the order of their appearances in the program. Let v be a branching 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 branching variable v_{c_0}'s identifier is "0", indicating that c_0 is indexed by 0, and has no parent; the branching variable v_{c_1}'s identifier is "1_-0", indicating that c_1 is indexed by 1, its single parent is indexed by 0 (i.e., c_0), and the "-" symbol indicates that c_1 is on the *false* branch of c_0.

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

4.2 Decision Values

After a variable is chosen, we need further to decide its value (called *decision value*). A decision value is either *true* or *false*. Many heuristics [20, 25, 30, 31, 32, 40] have been proposed for prioritizing the decision values of the chosen variable. However, all these heuristics are designed for general satisfiability problems. With the control-flow knowledge, we are capable of devising more efficient heuristics for prioritizing decision values.

Let G be the control-flow graph of the program. Given a branching node d of G, we denote its *left*-child as d_l that assumes the branching condition being *true*, and its *right*-child as d_r that assumes the branching condition being *false*. Let v_d be the branching variable for d. An assignment to v_d corresponds to an outgoing edge from d. Specifically, $v_d = true$ represents the edge from d to d_l, and $v_d = false$ stands for the edge from d to d_r.

Let $V^b \subseteq V$ be the set of branching variables. We intend to infer a prioritized value for each branching variable by utilizing the control-flow knowledge. The inferring algorithm is depicted in Algorithm 3. This algorithm maintains a data structure W that records a weight for each node of G. At the beginning of the algorithm (at line 2), all weights in W are initialized to -1. Then, depending on different heuristics, one of the two procedures (*LSP* or *LAP*) is invoked for computing the

ALGORITHM 3: *inferValues*(G, heu)

Input: A control-flow graph G, and a method *heu* for computing weights.
Output: A mapping *pValue* : $V^b \rightarrow \{true, false\}$, where$V^b \subseteq V$ are branching variables.

1: let W be a mapping from nodes of G to weights;
2: **for each** node d of G **do**$W(d) \leftarrow -1$;
3: **if** $heu = 0$ **then** ▷ by heuristic in Section 4.2.1
4: *LSP*(ENTRY);
5: **else** ▷ by heuristic in Section 4.2.2
6: *LAP*(ENTRY);
7: **for each** $v \in V^b$ **do**
8: let d be the control-flow node that corresponds to v;
9: let d_l and d_r be the *left*-child and *right*-child of d, respectively;
10: $pValue(v) \leftarrow W(d_l) < W(d_r)$;
11: **return** *pValue*;

ALGORITHM 4: *LSP*(d)

Input: A control-flow node d.
Output: The weight of d.

1: **if** $W(d) \geq 0$ **then return** $W(d)$; ▷ the weight has already been computed
2: $W(d) \leftarrow MAX_VALUE$;
3: **for each** child node d' of d **do**
4: $W(d') \leftarrow LSP(d')$;
5: **if** $W(d') < W(d)$ **then** $W(d) \leftarrow W(d')$; ▷ compute the shortest length
6: $W(d) \leftarrow W(d) + sizeof(d)$;
7: **return** $W(d)$;

ALGORITHM 5: *LAP*(d)

Input: A control-flow node d.
Output: The weight of d.

1: **if** $W(d) \geq 0$ **then return** $W(d)$; ▷ the weight has already been computed
2: $W(d) \leftarrow sizeof(d)$;
3: **for each** child node d' of d **do**
4: $W(d') \leftarrow LAP(d')$;
5: $W(d) \leftarrow W(d) + W(d')$; ▷ compute the sum length
6: **return** $W(d)$;

weights (at line 4 or 6). Finally, with the computed weights, the value mapping *pValue* that assigns a prioritized value to each branching variable is computed and returned (at lines 7 to 10).

4.2.1 Shortest Path First. Program verification is essentially to check if there exists an *error path* that violates the desired property. The program verification process can be considered as the procedure of searching the error path on the control-flow graph of the program. Let d be the current node to be explored, we need to check whether or not there is a program path from d to the error state. Among the many paths starting from d, we would like to attempt the shortest first.

Formally, consider a branching node d, the shortest path from d can be divided into two parts, i.e., the statements on d and the shortest path from its child nodes. Let $sizeof(d)$ be the number of statements on d. The *length of the shortest path (LSP)* from d can be computed in a recursive way, as

$$LSP(d) = sizeof(d) + \min_{d' \in child(d)} LSP(d').$$

Especially, if d is the `Exit` node, $LSP(d) = sizeof(d) = 0$.

Heuristic 3. *For a branching variable, we prefer the value assignment that leads to the* shortest path.

Algorithm 4 depicts the algorithm for computing $LSP(d)$, which starts from a given node, traverses and computes the weights of all its offspring nodes. Note that this algorithm needs only to count the number of statements, and has nothing to do with the program semantics. Its worst-case complexity is $O(n)$, where n is the number of offspring nodes of the given node. The computed weights are all recorded in the mapping structure W, and returned to Algorithm 3. Finally, at line 10 of Algorithm 3, let v_d be the branching variable for the branching node d, we set the prioritized value $v_d = true$ if $LSP(d_l) < LSP(d_r)$, and $v_d = false$, otherwise.

For example, consider the c_0 node in Figure 3. Its left-child is the node labeled with the statements "`x=1; y=1;`", and its right-child is the c_1 node. By Algorithm 4, the weight of its left-child is 3, and the weight of its right-child is 4. Therefore, the length of the shortest path from c_0 to `Exit` (passing through its left-child) is 4, and the prioritized value for v_{c_0} is *true*.

4.2.2 Shorter Sum Length First. Different heuristics can be devised by taking different views on the search process of the program verification. The previous heuristic indeed assumes that each path has a similar probability of being an error path, and therefore suggests to take the shortest path first. From a different view, we may assume that each node in the CFG has nearly the same probability of leading to an error path.

Formally, let d be a control-flow node. Any path from d must pass through one of its child nodes. The *length of all paths* from d, written $LAP(d)$, can be computed using the following equation:

$$LAP(d) = sizeof(d) + \sum_{d' \in child(d)} LAP(d').$$

Especially, if d is the `Exit` node, $LAP(d) = sizeof(d) = 0$.

Heuristic 4. *Given a branching variable, we prefer the value assignment that leads to a* shorter sum length *of paths.*

Algorithm 5 depicts our algorithm for computing $LAP(d)$. As Algorithm 4, it is also a recursive algorithm. In the worst case, it needs to traverse all of its offspring nodes. Therefore, its worst-case complexity is also $O(n)$, where n is the number of offspring nodes of d. All computed weights are stored in the mapping structure W, and returned to Algorithm 3. Finally, at line 10 of Algorithm 3, let v_d be the corresponding branching variable for d, we set the prioritized value $v_d = true$ if $LAP(d_l) < LAP(d_r)$, and $v_d = false$, otherwise.

For example, consider the CFG in Figure 3. There are two paths from the c_1 node, each having a length of 3. Thus, the weight of c_1 is 7. Then consider the c_0 node, its left child has a weight of 3, and its right child (i.e., the c_1 node) has a weight of 7. By comparing the weights of these two child nodes, the prioritized value of v_{c_0} is *true*.

ALGORITHM 6: *decide*()

Output: An unassigned variable v and a Boolean value *value*

1: let *cur* be the current node in the branching graph
2: **if** $cur =$ `Exit` **then**
3: $(v, value) \leftarrow decide_nonbranching_variables()$;
4: **while** $var(cur)$ is an assigned variable **do**
5: let t be the assigned value to *cur*;
6: $cur \leftarrow next(cur, t)$;
7: $v \leftarrow var(cur)$; $value \leftarrow pValue(v)$;
8: **return** $(v, value)$;

4.2.3 Storing the Prioritized Values. We design a mechanism to implicitly store the prioritized values of the branching variables. As devised in Section 4.1.2, the identifier of a branching variable consists of several numbers, representing the branching node and its parent nodes, respectively. Here, we add a sign symbol "-" to the identifier to represent the prioritized value of the branching variable. Specifically, a "positive" identifier (without the sign symbol) represents the prioritized value of *true*, and a "negative" identifier (with the symbol "-") stands for the prioritized value of *false*. For example, the identifier "-1_-0" is "negative", representing the prioritized value for the branching variable v_{c_1} is *false*.

Apparently, the addition of the sign symbol does not affect the semantics of the program formula. As the decision order, the prioritized values can be restored in the parsing of the SMT formula, and then be employed to guide the DPLL(T) procedure.

4.3 Control Flow-Guided DPLL(T)

To guide the SMT solving, we enforce the decision order and the prioritized values that were inferred in the front-end of the verifier, in the *decide*() operator of DPLL(T).

Our implementation of *decide*() is shown in Algorithm 6. Assume there is a branching graph and a mapping of prioritized values to the branching variables. Let *cur* be the current node in the branching graph 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, 14]. 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 the prioritized value (computed in Section 4.2) for this variable.

Moreover, the *resolve_conflict*() method in Algorithm 1 needs to be slightly modified. In the 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.

4.4 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 when two blocks are mutually exclusive, the SMT solver cannot drop the opposite one.

ALGORITHM 7: *rewrite_ite*(Φ)

Input: A SMT formula Φ
Output: A rewritten formula Ψ without *ite* functions
1: $\Psi = \Phi$;
2: let *Ites* be the set of *ite*-terms in Φ;
3: **for each** *ite* $\in$ *Ites* **do**
4: let g, l, r be three parameters of *ite*;
5: replace *ite* in Ψ with$(g \rightarrow rewrite_ite(l)) \wedge (\neg g \rightarrow rewrite_ite(r))$;
6: **return** Ψ;

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:

$$\begin{aligned} c_0 &\rightarrow x_4 = 1 & c_0 &\rightarrow y_4 = 1 \\ \neg c_0 \wedge c_1 &\rightarrow x_4 = 2 & \neg c_0 \wedge c_1 &\rightarrow y_4 = 2 \\ \neg c_0 \wedge \neg c_1 &\rightarrow x_4 = 3 & \neg c_0 \wedge \neg c_1 &\rightarrow y_4 = 3 \end{aligned} \tag{5}$$

Compared to (3), which is obtained by the conventional approach, the block conditions are specified in (5) 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 (5), 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*, and 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 Algorithm 7. 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 needs to be applied to l and r recursively. Finally, Ψ contains no *ite*-term. Our algorithm can collaborate with 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 contain only two literals for generating *binary clauses* [43] as more as possible. Binary clauses are good for Boolean propagations. However, the main problem here is not the 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 the CNF formulas obtained by the conventional approach, in many cases, only when the theory solver is involved can the same result be deduced. 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 CONTROL-FLOW KNOWLEDGE AS CONSTRAINTS

To utilize the control-flow knowledge, a more direct idea is to encode it in the verification condition. In this section, we discuss the feasibility of this approach.

5.1 Explicit Encoding Approach

Recall the aforementioned control-flow knowledge: (1) if a block is taken by one execution, the statements in this block are all executed, and (2) mutually exclusive blocks cannot be both taken by one execution. To encode the above control-flow knowledge into the SMT formulas, we first introduce some formal notions.

Let P be a program. An *execution* σ of P is a sequence of blocks from the `ENTRY` to the `EXIT` on the CFG of P. Denote $cond(\sigma)$ the *condition* of σ, i.e., the conjunction of block conditions of all blocks on σ. An execution σ is said *feasible* if and only if $cond(\sigma)$ is satisfiable.

The encoding formula of an execution σ, called an *execution formula*, is the conjunction of encoding formulas for all blocks on σ, i.e.,

$$encExe(\sigma) \equiv \bigwedge_{\forall b \in \sigma} encBlk(b), \tag{6}$$

where the encoding formula for a block b is the conjunction of encoding formulas for all statements in b, i.e.,

$$encBlk(b) \equiv \bigwedge_{\forall st \in b} encSt(st), \tag{7}$$

where the encoding formula for a statement st is either an equation (if st is an *assignment* statement) or a logical expression (if st is an *assume* statement). Moreover, the condition formula and the encoding formula of an execution σ compose a *guarded execution formula* as:

$$cond(\sigma) \rightarrow encExe(\sigma).$$

Let Σ be the set of all executions of the program. The *program formula* is the conjunction of all guarded execution formulas, i.e.,

$$Enc' \equiv \bigwedge_{\sigma \in \Sigma} \left(cond(\sigma) \rightarrow encExe(\sigma)\right). \tag{8}$$

Note that Enc' in the above formula needs to enumerate all executions of P (analogous to symbolic execution [26]).

Consider the control-flow graph in Figure 3. There are three executions, from leftmost to rightmost, denoted as σ_0, σ_1, and σ_2, respectively. Their conditions are $p_0 \equiv c_0$, $p_1 \equiv \neg c_0 \wedge c_1$ and $p_2 \equiv \neg c_0 \wedge \neg c_1$, respectively. The encoding formula Enc' for this program is then the conjunction of the following guarded execution formulas:

$$\begin{aligned} p_0 &\rightarrow x_4 = x_1 \wedge x_1 = 1 \wedge y_4 = y_1 \wedge y_1 = 1 \\ p_1 &\rightarrow x_4 = x_2 \wedge x_2 = 2 \wedge y_4 = y_2 \wedge y_2 = 2 \\ p_2 &\rightarrow x_4 = x_3 \wedge x_3 = 3 \wedge y_4 = y_3 \wedge y_3 = 3 \end{aligned}$$

Apparently, this encoding formula Enc' is different from the Enc formula in (2). However, Enc' is *equi-satisfiable* to Enc, and the verification condition $VC' = Enc' \wedge Cor$ is also *equi-satisfiable* to the VC formula defined in (1).

5.2 Discussion

We now explain that the aforementioned control-flow knowledge has been encoded into the new formula Enc'.

Let σ be a program execution. If its condition $cond(\sigma)$ is satisfied, according to the guided execution formula

$$cond(\sigma) \rightarrow \bigwedge_{\forall b \in \sigma} encBlk(b),$$

the encoding formula $encBlk(b)$ of each block $b \in \sigma$ is propagated to be *true*, and further by the defining formula (7) of $encBlk(b)$, the encoding formula of each statement in b is propagated to be *true*. In other words, if the execution σ is taken, all blocks on this execution, and all statements on these blocks, are all executed.

Note that the execution condition is the conjunction of all block conditions on this execution. Two mutually exclusive blocks can never occur on the same execution (otherwise, this execution would be infeasible). Moreover, for any two different executions σ_1 and σ_2, they diverge on at least one branch (i.e., they take opposite values for the condition of at least one branch), thus the conjunction $cond(\sigma_1) \wedge cond(\sigma_2)$ must be *false*. In other words, the conditions of any two feasible executions cannot be simultaneously satisfied. If the execution σ is taken, its condition $cond(\sigma)$ must be satisfied, then the conditions of other feasible executions must all be unsatisfied. Then, according to the Enc' formula in (8), the guarded execution formulas for executions excluding σ are all trivially satisfied (since their premises do not hold), and no longer need to be considered in the search process of DPLL(T).

The explicit encoding approach faces a severe path-explosion problem (as the symbolic execution [26]). Recall that the Enc' formula is essentially to enumerate all executions of the program. It is exponentially large in the number of branches in the program. The overhead of processing this huge formula can easily exceed the benefits from its utilization of control-flow knowledge. In many cases, this approach may not speed up, but severely degenerate the whole performance of the verification (see Section 6).

6 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

In this section, we discuss the implementation and the evaluation of our approaches. We mainly focus on the following research questions:

RQ1. How does our approach compare to the unmodified $Z3$?

RQ2. What do the two heuristics for decision values compare with each other?

RQ3. What do the different tactics (proposed in Sections 4.1, 4.2, and 4.4, respectively) compare with each other?

How does the explicit encoding affect the efficiency of SMT solving?

6.1 Implementation

We implement our control flow-guided SMT solving approach (Section 4) on top of the open-source projects *CBMC* [27] and $Z3$ [14], where *CBMC* is a well-known bounded model checking tool for C/C++ programs, and $Z3$ is an efficient SMT solver. In our implementation, *CBMC* acts as the front-end that generates the SMT formulas of the verification conditions for the programs, and $Z3$ is the back-end that solves these SMT formulas. Our algorithms for inferring the decision order (in Section 4.1) and the decision values (in Section 4.2) are implemented in *CBMC*, since the control-flow information can be directly obtained after *CBMC* parses the programs. Our control flow-guided DPLL(T) algorithm (in Section 4.3) and the enhanced CNF conversion algorithm (in Section 4.4) are implemented in $Z3$. Specifically, two new procedures were created, one for

rebuilding the DAG of the branching graph, and another for rewriting the nested *ite*-terms. Moreover, the *decide*() procedure in *Z*3 was modified to integrate our new branching heuristics. The changed codes in *Z*3 are less than 3000 lines, and are loosely coupled with other parts of *Z*3. The SMT files transferred between *CBMC* and *Z*3 are in the SMT-LIB-v.2.0 format.

Moreover, our explicit SMT encoding approach (in Section 5) is also implemented in *CBMC*. With this approach enabled, *CBMC* generates SMT formulas that explicitly encode control-flow knowledge.

6.2 Benchmarks

All experiments are conducted on the *ReachSafety* benchmarks in SV-COMP'17. This benchmark set contains 2, 897 C programs. All these programs have already been preprocessed for verification.

We employ *CBMC* to generate SMT formulas from these benchmark programs. Note that *CBMC* handles loops in the program by unrolling them to a given bound. A program with an unrolling bound is called a *verification instance*. For each program, we ask *CBMC* to generate various SMT formulas by setting different unrolling bounds, within 300 seconds. The generating process for a program also stops if its unrolling bound reaches 2,000. Let k be the unrolling bound. For an erroneous program, let k^* be the minimal value of k such that an error trace can be revealed from the program. Then, for any SMT formula generated from this program with the bound k, it is unsatisfiable if $k < k^*$, and satisfiable otherwise. 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 1, 411 SMT instances. All of these SMT formulas are stored in plain text files of the SMT-LIB-v.2.0 format. The largest SMT files occupy hundreds of megabytes.

Note that some preamble tactics, e.g., the simplifying tactic, the eliminating tactic, and the like, are implemented in *Z*3, with which some of these SMT formulas can be immediately solved without calling the DPLL(T) procedure. Therefore, these formulas cannot be used to evaluate our enhancement techniques to the DPLL(T) procedure. Furthermore, there are also some SMT formulas that are too hard to be solved by either the baseline or our tactics within the time limit (600 seconds). We drop the above two kinds of exceptional formulas. Finally, we get 948 SMT formulas, where 314 instances are satisfiable, and 634 ones are unsatisfiable. We consider these benchmarks are sufficient to evaluate the performance of our tactics credibly.

6.3 Evaluation of Our Whole Approach

The first experiment evaluates the whole performance of our control flow-guided approach. We choose *Z*3 as the *baseline* for the following reasons:

- *Z*3 is a state-of-the-art SMT solver;
- *Z*3 has been widely applied in program verification;
- *Z*3 is well maintained, and our heuristics were implemented on top of *Z*3.

The enhancement of *Z*3 with our control flow-guided tactics, i.e., the decision order tactic, the decision value tactic, and the enhanced CNF conversion tactic, is named $Z3^+$. For the two heuristics for prioritizing decision values, we chose *LSP* here (see Section 6.4 for explanation). We follow the timeout setting in SV-COMP, i.e., setting 900 seconds as the time limit for each verification task. Since we have already used 300 seconds for generating SMT files, the time limit for solving each SMT instance is set to 600 seconds.

Results. Figure 6(a) shows the SMT solving time of the *baseline* and $Z3^+$ on all benchmarks. The horizontal axis represents the time spent by the *baseline*, and the vertical axis represents the

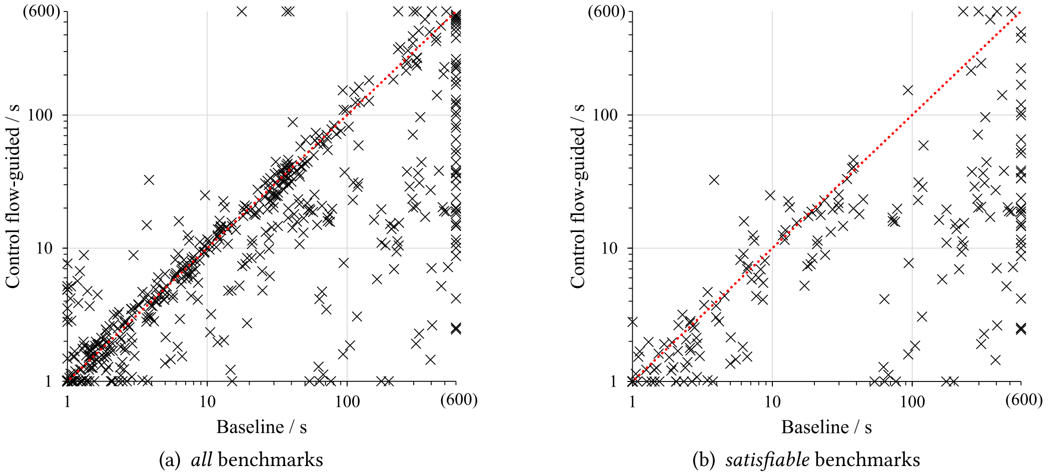


Fig. 6. SMT solving time on *all*/*sat* benchmarks.

Table 1. Experiment Results on the *baseline* and Our Approach ($Z3^+$)

Instances	CPU Time / s				Score Time / s				#Timeout	
	#Bench	*Baseline*	$Z3^+$	Ratio	#Solved	*Baseline*	$Z3^+$	Ratio	*Baseline*	$Z3^+$
sat	314	34713	7530	**4.61**	284	16713	1888	**8.85**	30	**4**
unsat	634	26201	19808	**1.32**	612	13003	9245	**1.41**	22	**4**
all	948	60915	27338	**2.23**	896	29716	8776	**3.39**	52	**8**

time spent by $Z3^+$. Each x mark represents an instance. The closer the mark is to the lower-left corner, the better $Z3^+$ is, and vice-versa. Figure 6(a) shows that $Z3^+$ is significantly superior to the *baseline* on about 65% of the benchmarks. Especially, our control flow-guided tactics speed up the SMT solver by orders of magnitude on about a quarter of the benchmarks, while there are only very few (3%) benchmarks on which $Z3^+$ is notably inferior to *baseline*. We also present the SMT solving time of both tactics on satisfiable benchmarks in Figure 6(b). On this subgroup of benchmarks, $Z3^+$ is significantly superior to *baseline* for 80% of the benchmarks. There are only a handful of benchmarks that cost $Z3^+$ more time. Besides, nearly one-third of the benchmarks are sped up by orders of magnitude.

Table 1 shows the statistic of this experiment. The *CPU time* is the total running time of each solver on the whole benchmark set (including *timeout* instance). In the comparison of the CPU time, $Z3^+$ is 2.23 times faster than *baseline* on all benchmarks and 4.61 times faster on the satisfiable benchmarks. The *score time* is the accumulated time for solving a certain quantity of benchmarks, where the quantity is the maximal number of benchmarks that the *baseline* can solve. In the comparison of the score time, our control flow-guided tactic speeds up the solver by 3.39 times on all benchmarks and 8.85 times on satisfiable benchmarks. Also, the number of timeout cases with our control flow-guided tactic is much fewer than that with *baseline*. In conclusion, our control flow-guided SMT solving tactic can significantly improve the performance of the SMT solver.

Result Analysis. The experimental results show remarkable improvements of our tactic over the baseline. The control flow-guided tactic help to prune the redundant search branches by providing the decision order and the prioritized decision values. Also, note that the improvement of $Z3^+$ on unsatisfiable benchmarks is not as significant as that on satisfiable benchmarks. In other words,

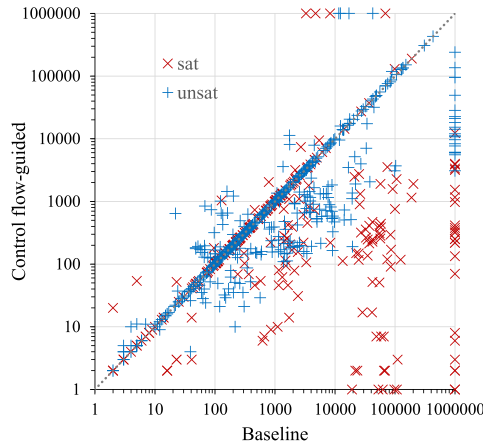


Fig. 7. #Conflict clauses learned during DPLL(T).

the utility of $Z3^+$ is limited for the unsatisfiable instances. Recall that DPLL(T) is essentially an exhaustive search procedure. For an unsatisfiable formula, no matter $Z3^+$ or the baseline, the whole search space must be explored to prove the 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 assignment 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 7 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 by $Z3^+$ is fewer than that by *baseline*, especially for the satisfiable benchmarks. These results indicate that $Z3^+$ can guide DPLL(T) to find a satisfiable solution with much fewer times to try in the search procedure.

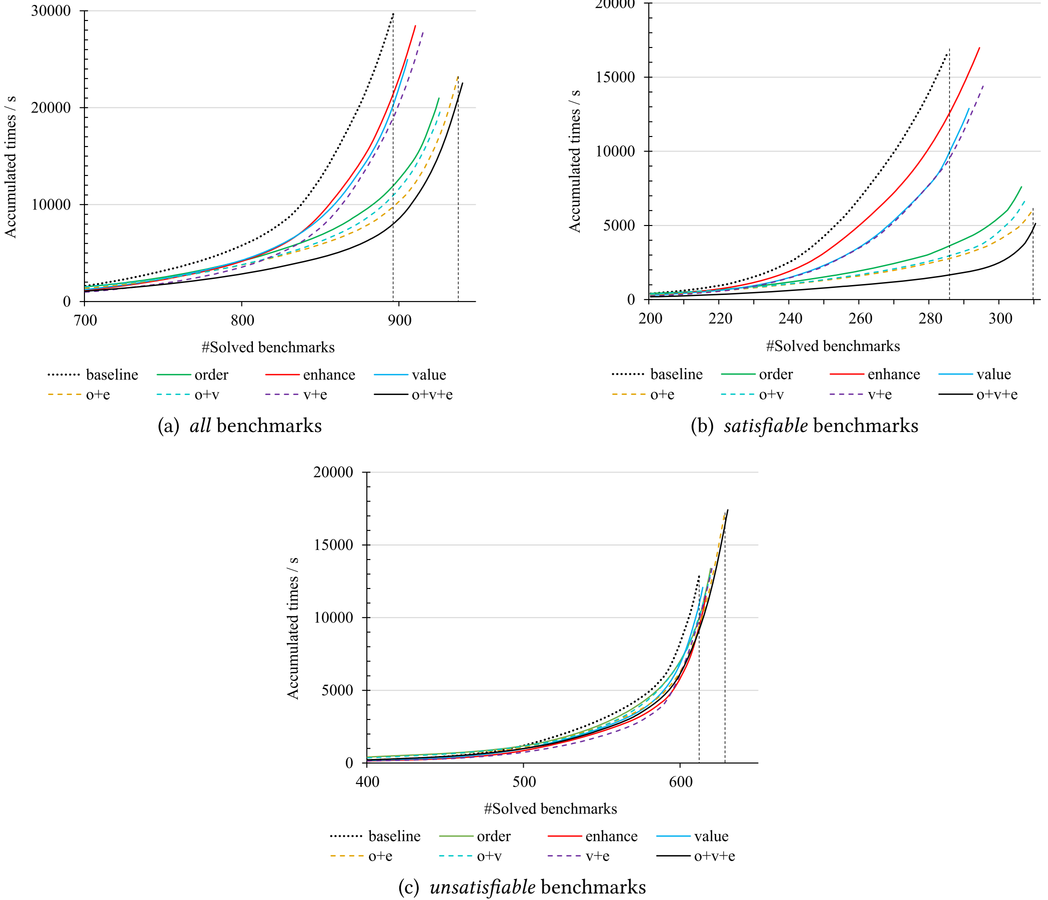
6.4 Evaluation of *LSP* and *LAP* Algorithms

The second experiment compares the two algorithms, i.e., *LSP* (in Section 4.2.1) and *LAP* (in Section 4.2.2), for prioritizing the decision values of branching variables. Recall that both *LSP* and *LAP* require to modify *CBMC* so as to compute prioritized values for the branching variables. The modified versions of *CBMC* are referred to as *CBMC* + *LSP* and *CBMC* + *LAP*, respectively. We use *CBMC*, *CBMC* + *LSP*, and *CBMC* + *LAP* to generate SMT formulas, respectively. Note that these three sets of SMT formulas differ only in the identifiers of variables. We then apply *Z*3, *Z*3 + *LSP*, and *Z*3 + *LAP* on these three sets of formulas, respectively, and compare their performance. Note that other tactics for the decision order and the enhanced CNF conversion are not applied in this experiment.

Experimental results are presented in Table 2. We observe that both *LSP* and *LAP* outperform the *baseline*, which evidences the efficiency of our control flow-guided prioritization methods. Moreover, *LSP* performs the best among all algorithms. It is 20% faster than the *baseline* on all benchmarks, and 30% faster on the category of satisfiable instances. Even for unsatisfiable instances, we still observe a 9% speed-up of *LSP* over the *baseline*. In conclusion, the experimental results show the effectiveness of our decision value prioritizing algorithms. In other experiments, we will choose *LSP* as the algorithm for prioritizing decision values.

Table 2. Comparison Results of Decision Value Prioritizing Algorithms

Instances	#	CPU Time / s			Ratio	
		Baseline	*LSP*	*LAP*	*LSP*	*LAP*
sat	314	34713	26689	28455	**1.30**	1.22
unsat	634	26201	24079	25689	**1.09**	1.02
all	948	60915	50767	54144	**1.20**	1.13



(a) *all* benchmarks

(b) *satisfiable* benchmarks

(c) *unsatisfiable* benchmarks

Fig. 8. Accumulated SMT solving time on *all*/*satisfiable*/*unsatisfiable* benchmarks.

6.5 Evaluation of Different Tactics in Our Approach

The third experiment compares the efficiency of different tactics in our approach. Recall that there are three tactics for utilizing control-flow knowledge, e.g., the decision order tactic (denoted as *order*), the decision value tactic (denoted as *value*), and the enhanced CNF conversion tactic (denoted as *enhance*). This experiment compares different combinations of these tactics, i.e. ,*baseline*, *order*, *enhance*, *value*, *order* + *enhance* (abbreviated as *o+e*), *order* + *value* (abbreviated as *o+v*), *value* + *enhance* (abbreviated as *v+e*), and *order* + *value* + *enhance* (abbreviated as *o+v+e*).

Figure 8(a) presents the accumulated SMT solving time of these eight single/combined tactics on all benchmarks. The black dotted line represents the *baseline*, and the black solid line represents *o+v+e*. The colorful solid lines represent the single tactics, respectively, i.e., *order*, *enhance*, and

Table 3. The Experimental Results of Each Configuration

Config	**Baseline**		**order**		**enhance**		**value**	
	#Solved	Time/s	#Solved	Time/s	#Solved	Time/s	#Solved	Time/s
sat	284	16713	306	7605	294	16987	291	12889
unsat	612	13003	619	13384	616	11472	614	12079
all	896	29716	925	20989	910	28458	905	24967
Config	**o+e**		**o+v**		**v+e**		**o+v+e**	
	#Solved	Time/s	#Solved	Time/s	#Solved	Time/s	#Solved	Time/s
sat	309	5969	307	6673	295	14384	310	5130
unsat	628	17211	619	13063	620	13476	630	17408
all	937	23180	926	19736	915	27860	940	22538

value. The colorful dashed lines represent the combined tactics, respectively, i.e., *o+e*, *o+v*, and *v+e*. All timeout instances are ruled out from the plot. As we can see, all the three single tactics, i.e., *order*, *value* and *enhance*, outperform the *baseline*. And among these three single tactics, *order* is the best, *value*, and *enhance* are similarly effective. We also observe that the combination of any two single tactics is better than these two tactics in isolation. Moreover, the *o+v+e* tactic achieves the best performance among all single/combined tactics.

Figure 8(b) depicts the accumulated SMT solving time of all tactics on the category of satisfiable benchmarks. Similar results as in Figure 8(a) can be observed, except that the performance improvements of our tactics over the *baseline* are more significant. Besides, Figure 8(c) shows the accumulated SMT solving time on the category of unsatisfiable benchmarks. The best tactic is again *o+v+e*. Comparing to the *baseline*, the improvements of our tactics on the category of unsatisfiable benchmarks seems to be marginal; but the best tactic *o+v+e* is still 28% faster than the *baseline*. From this plot, we also observe that the tactics combined with the *enhance* tactic outperform others, indicating that the *enhance* tactic contributes most to the unsatisfiable benchmarks.

The statistic of this experiment is listed in Table 3. Compared to the *baseline*, all of our tactics solve more benchmarks but cost less time. Especially, the *o+v+e* tactic solves 44 more benchmarks (26 *sat* and 18 *unsat*) than the *baseline*. Note that the timeout cases are not counted in the accumulated solving time. The accumulated solving time of different configurations cannot be directly compared. Considering the accumulated solving time of *o+v+e* and *baseline* for *unsat* cases, which are 17,408 and 13,003 seconds, respectively, it seems that *baseline* is faster. However, with the 18 timeout cases counted in, the accumulated solving time of *baseline* should be $13,003 + 18 \times 600 = 23,803$, greater than that of *o+v+e*.

6.6 Evaluation of Explicit Encoding Approach

The fourth experiment evaluates the efficiency of the approach that explicitly encodes the control-flow knowledge into the SMT formulas (Section 5). In the following, we refer to this approach as *explicit*, and the default SMT encoding approach in *CBMC* as the *baseline*.

Recall that the 948 SMT formulas in our benchmark set were generated by *baseline* (Section 6.2). We intend to apply *explicit* to the same set of verification instances and generate a new set of SMT formulas. If a formula is generated by *baseline* (or *explicit*), we simply call it a *baseline* (or *explicit*) formula. Note that an *explicit* formula can be much larger than the *baseline* formula. We thus increase the time limit for generating *explicit* formulas from 300 seconds to 600 seconds. However, there are still 44 instances for which the *explicit* formulas failed to be generated in the time limit. Finally, only 904 formulas were generated by *explicit*. In the following, we limit our discussion to these 904 verification instances. We will use the solving efficiency of the *explicit* and *baseline* formulas to evaluate the efficiency of these two encoding approaches.

Table 4. Results on SMT Encoding Approaches: *baseline* vs. *explicit*

Instances	#	**CPU Time / s**			**#Timeout**	
		Baseline	*Explicit*	Ratio	*Baseline*	*Explicit*
sat	306	31228	69432	**0.45**	22	**99**
unsat	598	25893	71264	**0.36**	26	**107**
all	904	57121	140696	**0.41**	38	**206**

Experimental results are presented in Table 4. Among the 904 instances, 866 *baseline* formulas can be solved, while only 698 *explicit* formulas are solvable in the time limit of 600 seconds. More specifically, there are 77 more satisfiable *explicit* formulas and 81 more unsatisfiable *explicit* formulas that cannot be solved. The total CPU time spent on solving all *explicit* and *baseline* formulas are 140, 696 and 57, 121 seconds, respectively. In summary, the total CPU time is slowed 2.46 times down by *explicit*. The main reason is due to the huge size of the *explicit* formulas. We also count the file size of all SMT files generated by both approaches. We find that the sum size of the SMT files generated by *baseline* and *explicit* are 14 and 114 gigabytes, respectively. The average size of *explicit* formulas is 8.14 times larger than that of *baseline* formulas.

6.7 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 of our tactic is credible. First, we only revised two algorithms in $Z3$, i.e., the branching heuristics (in Section 4.3) and the enhanced CNF conversion algorithm (in Section 4.4). Note that the latter algorithm (in Section 4.4) is implemented as a preamble tactic. These two algorithms are barely coupled with other modules in $Z3$. Meanwhile, they were implemented as clearly as we can. Second, the adding of the control-flow information won't affect the performance of the original solver, since we implicitly record the decision order and the prioritized values 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 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.

The *external* threat to the validity of our approach is whether our approach can be generalized to other SMT solvers than $Z3$. Actually, our tactic is based on the DPLL(T) framework. It has nothing to do with the specific features of $Z3$ solver itself. Thus, our tactic can be implemented in any DPLL(T) -based SMT solver for program analysis. One more external threat is whether our approach can be generalized to other program analysis techniques, except for bounded model checking. The answer is also yes. Actually, *CBMC* is only used to generate SMT instances in our experiments, 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 SMT formulas and then rely on an SMT solver to solve these formulas.

The main *constructive* threat to the validity of our approach is that whether our evaluation metrics are sufficient to answer the research questions. Recall that all research questions in the evaluation are about the performance comparisons of SMT solving under different approaches/heuristics/tactics. A natural metric for performance evaluation is the runtime. Therefore, we used the SMT solving time, the numbers of solved/timeout cases as the main metrics for the performance evaluations. Besides, we also use the number of learned clauses as an auxiliary metric

to measure the try times in the search procedure of DPLL(T). With these evaluation metrics, the experimental results provide sufficient views for answering the research questions.

7 RELATED WORK

There is a large body of work on improving the performance of 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 *decision ordering* and the *decision values* (also named the *branching heuristic*) of the DPLL search procedure, we discuss this aspect first.

7.1 Branching Heuristics

One of the most famous branching heuristics is VSIDS [30, 32]. It increases the weight values of each literal in a newly inferred conflict clause, and also decays these values periodically. The order and the prioritized values are based on these weight values. There are also other branching heuristics that utilize the information from the backtrack search procedure, like MOM heuristic [20, 40], Jeroslow-Wang heuristics [25], literal count heuristics [31], and the like. 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 control flow-guided heuristics are combined with the default branching heuristics (VSIDS-based), but focus on the satisfiability problems derived from the program analysis. With the domain-specific knowledge of programs, our approach significantly improves the efficiency of SMT solving.

There is some work on refining the decision ordering by the domain-specific knowledge of transition systems. The most closely related work is [46], where the decision ordering is refined for the satisfiability problem. More specifically, the transition variables are given higher priority than other variables in the decision order, and among the transition variables, they are ordered according to the transition relation of the model. Moreover, in [45], Wang et al. identify important variables from the *unsat*-core of the previous unsatisfiable BMC instances, and give priority to the same variables in the decision ordering of the current BMC instance. Shtrichman [42] suggests a static predetermining order by following the breadth-first search on the variable dependency graph of the transition system. In [23], Gupta et al. explore implications learned from the circuit structure to help the SAT solving. Since there is less structure information in a CNF formula, Ostrowski et al. [37] suggest recovering and exploiting structural knowledge by a set of equations from the initial SAT instance to eliminate clauses and variables. All these techniques are designed for SAT solvers, and the utilized domain knowledge is from the transition systems. Comparing these techniques to ours, the domain knowledge is different, and the application target is also different.

7.2 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] partition 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 partitioning the monolithic problem into several smaller problems utilizing the control-flow information. Our approach also uses this divide-and-conquer strategy in that giving priority to the branching variable is essential to split the whole program into several parts such that they can be solved separately.

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 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 works are also different from ours.

7.3 Theory-Aware Approach

The optimization of our tactic is based on the utilization of domain-specific knowledge. The existing theory-aware approaches optimize constraint solving in a similar way. These approaches discover constraints from the underlying theory by a lightweight method and utilize these constraints to prune the conflict assignments in the DPLL procedure. Berzish et al. [3] proposed a theory-aware branching heuristic that 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 the string theory literals. Goldwasser et al. [21] proposed a theory-aware branching heuristic for linear real 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. [8] utilize the structural information like equalities and arithmetic functions to reason at a higher level of abstraction within the theory of bit-vectors. Each of these theory-aware approaches focuses on a specific theory, such as string theory, linear real arithmetic theory, and bit-vector theory, and the like. Instead, our tactic is a high-level approach based on refining the branching heuristic of the DPLL procedure, which is independent of theories.

Moreover, in [28], Kuehlmann et al. use circuit-specific knowledge to guide the search of SAT solving and help the solver to reason on specific logic. In [23], Gupta et al. explore implications learned from the circuit structure to help the SAT solving. These works rely on the structural information of the BDD representation of the circuits, and can not be applied to program verification. In [18], [33], and [34], Nadel et al. employ the so-called *decision strategy* to speed up the SAT and SMT solving for routing in physical design. However, their techniques are only suitable for the specific optimization problem of circuit design, and also can not be applied to program verification. Besides the DPLL-based solver, the theory-aware approach has also been adapted to other constraint solvers. For example, Hooimeijer and Weimer [24] proposed a lazy backtracking search algorithm for solving the regular expression constraints by utilizing the domain knowledge of the constraint system. This work is specifically for solving regular expression constraints.

8 CONCLUSION

In this article, we presented a control flow-guided SMT solving approach for program verification. In this approach, a decision order and a value mapping were inferred from the control-flow structure of the program, and then employed to guide the search process of the DPLL(T). An enhanced CNF conversion algorithm was further developed to collaborate with the above tactics. With our approach, the search space of the DPLL(T) is reconstructed continuously and dynamically by the control-flow information, and thus a large number of redundant search paths can be pruned.

We implemented our control flow-guided tactics in the modern SMT solver $Z3$, and compared our tactics with the default heuristic in $Z3$. The experiments on SV-COMP'17 benchmarks showed that each of our tactics could speed up SMT solving. Especially, the combination of all tactics achieved orders of magnitude improvements on satisfiable benchmarks.

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, Robby, and Xianghua Deng. 2009. Sireum/Topi LDP: A lightweight semi-decision procedure for optimizing symbolic execution-based analyses. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2009 (ESEC/FSE 2009).* 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 *Proceedings of Formal Methods in Computer-Aided Design, 2009 (FMCAD 2009).* IEEE, 25–32.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 193–207.

[6] 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.

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

[8] 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 *Proceedings of the International Conference on Computer Aided Verification.* Springer, 547–560.

[9] Jianhui Chen and Fei He. 2018. Control flow-guided SMT solving for program verification. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 351–361.

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

[11] 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.

[12] 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.

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

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

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

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

[17] Herbert Enderton and Herbert B. Enderton. 2001. *A Mathematical Introduction to Logic.* Academic Press.

[18] Amit Erez and Alexander Nadel. 2015. Finding bounded path in graph using SMT for automatic clock routing. In *Proceedings of the International Conference on Computer Aided Verification.* Springer, 20–36.

[19] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. 2016. Guided dynamic symbolic execution using subgraph control-flow information. In *Proceedings of the 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 *Proceedings of the 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 *Proceedings of the 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 *Proceedings of the 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. Technical Report. Microsoft Research.
[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 *Proceedings of theHaifa Verification Conference*. Springer, 225–241.
[31] Joao Marques-Silva. 1999. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 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] Alexander Nadel. 2016. Routing under constraints. In *Proceedings of the2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 125–132.
[34] Alexander Nadel. 2017. A correct-by-decision solution for simultaneous place and route. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 436–452.
[35] Greg Nelson and Derek C. Oppen. 1980. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364.
[36] Alexandru Nicolau. 1988. Loop quantization: A generalized loop unwinding technique. *J. Parallel and Distrib. Comput.* 5, 5 (1988), 568–586.
[37] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. 2002. Recovering and exploiting structural knowledge from CNF formulas. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. Springer, 185–199.
[38] 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.
[39] M. Prasanna, S. Sivanandam, R. Venkatesan, and R. Sundarrajan. 2005. A survey on automatic test case generation. *Academic Open Internet Journal* 15, 6 (2005).
[40] Daniele Pretolani. 1996. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1996), 479–498.
[41] 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.
[42] Ofer Shtrichman. 2000. Tuning SAT checkers for bounded model checking. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 480–494.
[43] Niklas Sorensson and Niklas Een. 2005. Minisat v1. 13-a SAT solver with conflict-clause minimization. *SAT* 2005, 53 (2005), 1–2.
[44] G. Tseitin. 1968. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic* (1968).
[45] 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.
[46] Liangze Yin, Fei He, and Ming Gu. 2013. Optimizing the sat decision ordering of bounded model checking by structural information. In *Proceedings of the 2013 International Symposium on Theoretical Aspects of Software Engineering (TASE),*. IEEE, 23–26.

Received February 2020; revised December 2020; accepted December 2020