Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution

Xiangyang Jia
State Key Lab of Software Engineering,Wuhan University
Luojia hill Street, 229
Wuhan, China
jxy@whu.edu.cn

Carlo Ghezzi
DEIB, Politecnico di Milano
Via Golgi, 42
Milano, Italy
carlo.ghezzi@polimi.it

Shi Ying
State Key Lab of Software Engineering,Wuhan University
Luojia hill Street, 229
Wuhan, China
yingshi@whu.edu.cn

ABSTRACT

Constraint solution reuse is an effective approach to save the time of constraint solving in symbolic execution. Most of the existing reuse approaches are based on syntactic or semantic equivalence of constraints. For example, the Green framework can reuse constraints which have different representations but are semantically equivalent, through canonizing constraints into syntactically equivalent normal forms. KLEE reuses constraints based on subset/superset querying. However, both equivalence-based approach and subset/superset-based approach cannot cover some kinds of reuse where atomic constraints are not equivalent.

Our approach, called GreenTrie, is an extension to the Green framework, which supports constraint reuse based on the logical implication relations among constraints. GreenTrie provides a component, called L-Trie, which stores constraints and solutions into tries, indexed by an implication partial order graph of constraints. L-Trie is able to carry out logical reduction and logical subset and superset querying for given constraints, to check for reuse of previously solved constraints. We report the results of an experimental assessment of GreenTrie against the original Green framework and the KLEE approach, which shows that our extension achieves better reuse of constraint solving result and saves significant symbolic execution time.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.8 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA

http://dx.doi.org/10.1145/2771783.2771806

Keywords

constraint solving, symbolic execution, cache and reuse

1. INTRODUCTION

Symbolic execution has been proposed as a program analysis technique since the 1970's [1]. It gained a lot of attention in recent years as an effective technique for generating high-coverage test cases and finding subtle errors in software applications [2, 3]. Symbolic execution works by exploring as many program paths as possible in a given time budget, creating logical formulas encoding the explored paths, using a constraint solver to check for feasible execution paths and generate test cases, as well as finding corner-case bugs such as buffer overflows, uncaught exceptions, and checking higher-level program assertions [4, 5].

In symbolic execution, constraint solving plays an important role in path feasibility checking, test inputs generation, and assertions checking. Since constraint satisfaction is a well-known NP-complete problem, not surprisingly it is always the most time-consuming task in symbolic execution. Despite significant advances in constraint solving technology during the last few years—which made symbolic execution applicable in practice—constraint solving continues to be a bottleneck in symbolic execution [4, 6]. In order to ease constraint-solving in symbolic execution, some approaches have been proposed, such as irrelevant constraint elimination [7, 8], incremental solving [9, 8], and constraint solution reuse [10, 11, 9].

The Green framework [10] is a constraint solution reuse framework which stores the solutions of constraints and reuses them across runs of the same or different programs. Green stores constraints and their solutions as key-value pairs in an in-memory database Redis [12], and queries the solutions for reuse based on string matching. To improve the matching ratio, all the constraints are sliced and canonized before they are stored and queried. *Slicing* is a process to obtain the minimal constraint required for satisfiability checking, based on graph reachability checking. *Canonization* represents each individual constraint into a normal form. Linear integer sub-constraints are converted into a normal form $ax + by + cz + ... + k\,op\,0$, where $op \in \{=, \neq, \leq\}$. In addition, canonization sorts the constraint in a lexicographic order, and renames the variables into a standard form. For example, after canonization, the constraint $x + y < z \bigwedge x = z \bigwedge x + 10 > y$ becomes $-v0 + v1 - 9 \leq 0 \bigwedge v0 + v1 - v2 + 1 \leq 0 \bigwedge v0 - v2 = 0$. As a consequence of slicing and canonization, a constraint may become syntactically equivalent to a previously evaluated constraint and thus simple string matching may detect a potential reuse.

KLEE can reuse constraint solutions based on subsets and superset querying [9]. If a constraint $x > 0 \bigwedge x < y \bigwedge y - x > 1$ is proved to be satisfiable, it can be reused to prove that the constraint $x < y \bigwedge y - x > 1$ is also satisfiable. Also if we have proved constraint $x < 0 \bigwedge x > 1$ to be unsatisfiable, then the constraint $x < 0 \bigwedge x > 1 \bigwedge x \neq 10$ is proved to be unsatisfiable by reusing this result.

However, both equivalence-based approach [10, 13, 14, 11, 15] and subsets and superset based approach like KLEE cannot cover some kinds of reuse where atomic constraints are not equivalent. Here are some examples:

- Example 1: Suppose we have proved constraint $x > 0$ to be satisfiable, with a solution{x:1}. Constraint $x > -1$ can also be proved to be satisfiable by reusing this solution.

- Example 2: Suppose we have proved constraint $x < 0 \bigwedge x > 1$ to be unsatisfiable. Constraint $x < -1 \bigwedge x > 2$ can also be proved to be unsatisfiable by reusing this result.

In this paper, we present GreenTrie, an extension to the Green framework, which supports constraint reuse based on the logical implication relations among constraints. GreenTrie provides a component, called L-Trie, which stores constraints and solutions into tries (an ordered tree data structure that is used to store a dynamic set or associative array [16]), indexed by an implication partial order graph of constraints. L-Trie is able to carry out logical reduction and logical subset and superset querying for given constraints, to check for reuse of previously solved constraints. This approach supports constraints reuse based on their logical implication relations.The contributions of this paper can be summarized as follows:

- We present a theoretical basis for checking constraint reusability based on their logical relationship, and give rules to check the implication relationship between linear integer arithmetic constraints.

- We present a constraint reduction approach to reduce the constraint into more concise form, as well as to find obviously conflicting sub-constraints.

- We describe the L-Trie data structure, which is used to cache past constraint solutions into tries indexed by implication partial order graphs.

- We give logical superset and subset checking algorithms to check the existence of reusable solutions stored in L-Trie.

- We evaluate the performance of GreenTrie in three scenarios: (1) reuse in a single run of the program, (2) reuse across runs of the same program, (3) reuse across different programs. The experiments show that, compared to the original Green framework and the KLEE approach, GreenTrie achieves better reuse of constraint solving results, and saves significant time in symbolic execution.

2. LOGICAL BASIS OF OUR APPROACH

Constraint satisfiability checking—the quintessential NP-complete problem—has been studied extensively, with strong motivations arising especially from artificial intelligence. A (finite domain) constraint satisfaction problem can be expressed in the following form: given a set of variables, together with a finite set of possible values that can be assigned to each variable, and a list of constraints, find values of the variables that satisfy every constraint[17].

In symbolic execution scenarios, the target of constraint solving is to find a solution for given constraint (always in the form of a conjunction of several sub-constraints). The *solution*, if it exists, is a valuation function mapping the set of variables of a constraint to a value set. If we substitute the variables in the constraint with the values in the solution, the constraint evaluates to TRUE. When a solution exists, the constraint is *satisfiable*; if not, it is *unsatisfiable.* In this paper we focus on *linear integer constraints*, for which satisfiability is decidable. In our future work we plan to extend our approach to also cope with other kinds of constraints, such as non-linear constraints and string constraints.

LEMMA 1. *Given two constraints C and C', (1) if C is satisfiable and has a solution V, and $C \rightarrow C'$, then $C'$is satisfiable and V is also a solution of C'. (2) if C is unsatisfiable and $C' \rightarrow C$, then C' is unsatisfiable.*

PROOF. (1) Because C is satisfiable and has a solution V, by substituting the variables in the constraint with the values in solution V, C evaluates to TRUE. Since $C \rightarrow C'$, according to the definition of logical implication, C' evaluates to TRUE for this substitution too. Therefore, V is also a solution for C' and C' is satisfiable. (2) If C is unsatisfiable, $\neg C$ will evaluate to TRUE for all valuations. Since $C' \rightarrow C$, then $\neg C \rightarrow \neg C'$ and hence C' will evaluate to FALSE for all valuations.i.e C' is unsatisfiable. □

According to Lemma 1, checking the implication relationship between constraints can be a basis for reusing constraint satisfiablity checks. In symbolic execution, constraints are mainly utilized to represent the path conditions of branches in code, and each of them is a conjunction of all the branching conditions (in terms of the program inputs) form the first branch to current location. Therefore, a constraint is always in the form $C_1 \wedge C_2 ... \wedge C_n$, and has a sub-constraint set $\{C_1, C_2 ... C_n\}$. In our approach, we will check the reusability of such constraints through querying *logical subsets* and *logical supersets* of the sub-constraint set in the solution store.

Definition 1. (Logical subset and logical superset) Given two constraint sets X and Y, if $\forall_{x \in X} \exists_{y \in Y} y \rightarrow x$, then X is a logical subset of Y and Y is a logical superset of X.

For example, if X = {x≠0, x>-1, x<2}, Y={x>1, x<2}, because $x > 1 \rightarrow x \neq 0$, $x > 1 \rightarrow x > -1$, $x < 2 \rightarrow x < 2$, then X is a logical subset of Y, and Y is a logical superset of X, even though Y has less elements than X.

THEOREM 1. *Given two constraints in conjunctive form $C = \bigwedge_{i=1}^{n} C_i$, $C' = \bigwedge_{i=1}^{m} C'_i$, where C has a sub-constraint set $S = \{C_1, C_2 ... C_n\}$, and C' has a sub-constraint set $S' = \{C'_1, C'_2 ... C'_m\}$, (1) if C is satisfiable and has a solution V, and S is a logical superset of S', then $C'$is satisfiable and V is also a solution of C'. (2) if C is unsatisfiable, and S is a logical subset of S', then $C'$is unsatisfiable.*

PROOF. (1) Since S is a logical superset of S', $\forall_{c' \in S'} \exists_{c \in S}$ $c \to c'$. Hence $C_1 \wedge C_2 ... \wedge C_n \to C'_1 \wedge C'_2 ... \wedge C'_m$, i.e. $C \to C'$. According to Lemma 1, if C is satisfiable and has a solution V, then C' is satisfiable and V is also a solutions for C'. (2) Since S is a logical subset of S', $\forall_{c \in S} \exists_{c' \in S'} c' \to c$. Hence $C'_1 \wedge C'_2 ... \wedge C'_m \to C_1 \wedge C_2 ... \wedge C_n$, i.e.$C' \to C$. According to Lemma 1, if C is unsatisfiable, then C' is unsatisfiable. □

According to Theorem 1, a constraint can be shown to be satisfiable if a logical superset can be retrieved in a storage that caches satisfiable sub-constraint sets. Likewise, a constraint can be shown to be unsatisfiable if a logical subset can be retrieved in a storage that caches unsatisfiable sub-constraint sets.

Normal form of linear integer constraint. In this paper, every atomic linear integer constraint is canonized into the form:

$$h_1 v_1 + h_2 v_2 + h_3 v_3 + ... h_n v_n + k \, op \, 0$$

where $v_1, v_2 ... v_n$ are distinct variables, the coefficients h_1, $h_2 ...$, h_n are numeric constants, k is an integer constant, $h_1 \geq 0$, and $op \in \{=, \neq, \leq, \geq\}$. The expression $h_1 v_1 + h_2 v_2 + h_3 v_3 + ... h_n v_n$, which contains all non-constant terms, is the constraint's *non-constant prefix.*

Implication Checking Rules. We define a list of rules to check for specific implication relationships between two atomic linear integer constraints. In this paper, only constraints which have the same non-constant prefix can be checked by rules. In the future, we plan to extend the rules to handle more complex situations.We compare non-constant prefixes based on string comparison and constant values based on numeric comparison, which is quite efficient. The implication checking rules are listed below. In these rules, P is a non-constant prefix and n is a constant value. The rules enable checking the implication relationship between linear integer arithmetic constraints with operators $=, \neq, \leq, \geq$.

$$(R1)\frac{}{C \to C} \qquad (R2)\frac{n \neq n'}{P + n = 0 \to P + n' \neq 0}$$

$$(R3)\frac{n \geq n'}{P + n = 0 \to P + n' \leq 0} \qquad (R4)\frac{n \leq n'}{P + n = 0 \to P + n' \geq 0}$$

$$(R5)\frac{n > n'}{P + n \leq 0 \to P + n' \neq 0} \qquad (R6)\frac{n > n'}{P + n \leq 0 \to P + n' \leq 0}$$

$$(R7)\frac{n < n'}{P + n \geq 0 \to P + n' \neq 0} \qquad (R8)\frac{n < n'}{P + n \geq 0 \to P + n' \geq 0}$$

3. OVERVIEW OF GREENTRIE

GreenTrie extends the Green framework to improve the reuse of constraint solutions. The overview architecture of GreenTrie is illustrated in Fig.1. GreenTrie includes a component named L-Trie, which replaces the Redis store of the original Green framework. L-Trie is a bipartite store used for caching satisfiable and unsatisfiable constraints, respectively, each composed of a constraint trie and its logical index. The constraint trie stores constraints in the form of sub-constraint sets, and the logical index is a partial order graph of implication relations for all the sub-constraints in the trie.

L-Trie and Green work together within GreenTrie. Any request to solve a constraint is handled by Green through the following four steps: *(1) slicing:* it removes pre-solved irrelevant sub-constraints; *(2) canonization:* it converts a constraint into normal form; *(3) reusing:* it queries the solution store for reuse; if a reusable result is not retrieved, (4) *translation:* the constraint is translated into the input format required by the chosen constraint solver (such as CVC3[18], Z3, Yices[19], or Choco), which is then invoked to solve the constraint from scratch. The result produced by the constraint solver is finally stored into either satisfiable constraint store(SCS) or unsatisfiable constraint store(UCS)(Fig.1).

L-Trie provides three interfaces to Green: constraint reduction, constraint querying, and constraint storing. These are presented in detail in the following sections. *Constraint reduction* is performed after the constraint is canonized by the Green framework; redundant sub-constraints are removed and conflicting sub-constraints are reported in this phase. *Constraint querying* handles the requests issued by Green to retrieve pre-solved constraints. Based on Theorem 1, it checks whether the constraint has a logical superset in the satisfiable constraint store or has a logical subset in the unsatisfiable constraint store. *Constraint storing* splits solved constraint into sub-constraints, puts them into the corresponding constraint trie, and the also updates the logical index.

4. CONSTRAINT REDUCTION

Symbolic execution conjoins constraints as control flow branches are traversed. This may introduce redundant sub-constraints, where a sub-constraint is implied by another. For example, if constraint $x \geq 0$ is conjoined to constraint $x \neq$-2, the latter becomes redundant and can be eliminated. It may also happen that one can easily detect that the newly added constraint conflicts with another constraints, making the whole constraint unsatisfiable; for example, consider the case where $x=0$ is conjoined with $x \geq 3$. Constraint reduction in our approach is able to recognize such situations: it can both reduce the constraint into more concise form and also find obviously-conflicted sub-constraints. As we mentioned, we only focus on the linear integer arithmetic constraints. In the future, we plan to reduce other kind of constraints based on term rewriting [20].

Our approach performs reduction as follows. The sub-constraints with same non-constant prefix are merged and reduced based on their *value interval of non-constant prefixes.* For example, considering constraint x+y+3≤0, its non-constant prefix x+y has a value interval $[MIN, -3]$, and for constraint x+y≥0, the value interval is $[0, MAX]$. As for constraint x+y+4=0, the value interval is $[4, 4]$. If the constraint is stated as an inequality, as for example x+y+6≠0, we have two value intervals $[MIN, -6)$ and $(-6, MAX]$. Equivalently, we can represent this situation by introducing the concept of an *exceptional point* (in this case, "-6").

To support reduction, firstly all sub-constraints with the same non-constant prefix are merged together, by computing the overlapping interval $[A, B]$ of these constraints, and at the same time collecting the exceptional points into a set E. For example, after computing of constraint $x + y + 3 \geq 0 \wedge x + y + 5 \geq 0 \wedge x + y - 4 \leq 0 \wedge x + y \neq 0 \wedge x + y + 6 \neq 0 \wedge x + y - 4 \neq 0$, we get an overlapping interval $[-3, 4]$ and

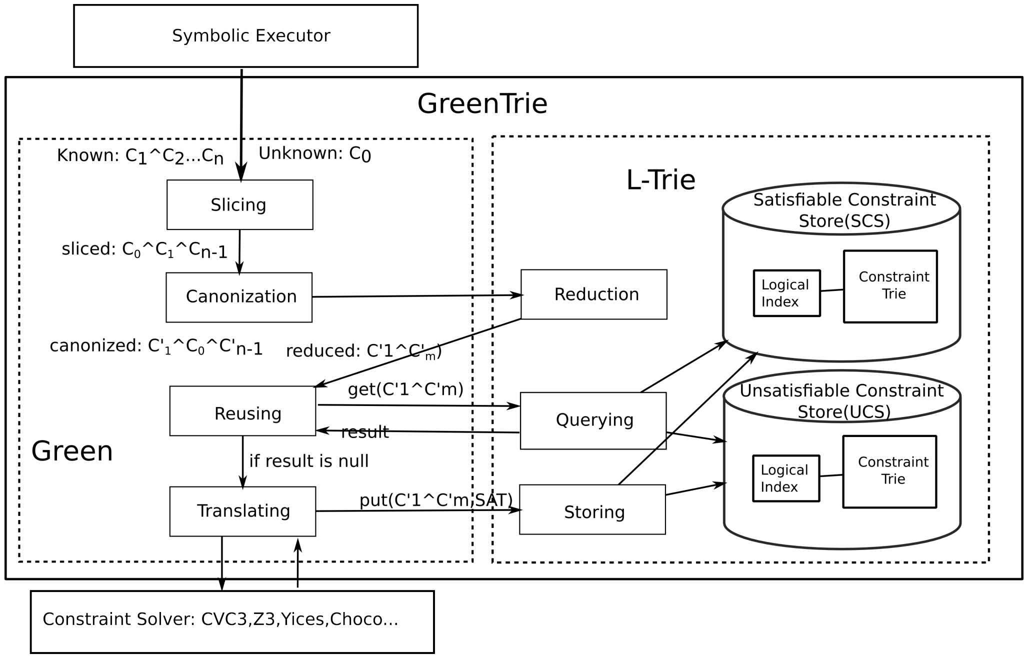

Figure 1: The overview architecture of GreenTrie

an exceptional point set $E = \{-6, 0, 4\}$. After this, we go through the following steps:

1. We discard all exceptional points that are outside the overlapping interval; in the example, the value of E becomes $\{0, 4\}$.

2. If one endpoint of the overlapping interval A (or B) belongs to E, we (repeatedly) change its value and eliminate A (or B) from E at the same time. In the example after this step the interval becomes $[-3, 3]$ and the new value of E is $\{0\}$.

3. If the overlapping interval is empty then the constraint is unsatisfiable and we report a conflict; otherwise we translate $[A, B]$ and E into a constraint in normal form. In the example, the final result of our reduction is $x + y + 3 \geq 0 \wedge x + y - 3 \leq 0 \wedge x + y \neq 0$.

5. CONSTRAINT STORING

L-Trie provides a different storage scheme that replaces the Redis store of Green:

- Unlike Redis, which stores the strings representing constraints and solutions as key-value pairs, L-Trie splits constraints into sub-constraint sets, and stores them into tries, in order to support logical subset and superset queries based on Theorem 1.

- L-Trie stores unsatisfiable and satisfiable constraints into separate areas: the *Unsatisfiable Constraint Store* (UCS) and the *Satisfiable Constraint Store* (SCS) respectively. The two areas are organized differently to efficiently support logical subset querying and logical superset querying, which pose different requirements.

- L-Trie maintains a logical index for each of the two tries, to support efficient check of the implication relations. The logical index is represented as an implication partial order graph (IPOG), whose nodes contain references to nodes in the trie.

Both UCS and SCS have the same structure (see Fig. 2).

Constraint Trie. The constraint trie is designed to store a sub-constraint set of solved constraints. The sub-constraint set is sorted in lexicographic order based on string comparison, to guarantee that sub-constraints with same non-constant prefix are kept close to each other. The labels of the constraint trie record the sub-constraints. The leaf nodes indicate the end of the constraint and are annotated with the solution (the solution is null for the leaves of the UCS trie). As shown in Fig.2, the leaf node C2 corresponds to a constraint v_0+5>=0 $\wedge v_0$+v_1<=0, which has a solution $\{v_0 : 0, v_1 : -1\}$, and its sub-constraints v_0+5>=0 and v_0+v_1<=0, are annotated as edge labels in the path.

If a constraint C is a conjunction of atomic constraints that is a prefix of another constraint C'(e.g. C is $A \wedge B$, and C' *is* $A \wedge B \wedge C$,), only one of them is kept in the trie. We keep the longer constraint in the SCS trie, while we keep the shorter in the UCS trie.

Implication Partial Order Graph (IPOG). IPOG is a graph that contains all the atomic sub-constraints appearing in its associated constraint trie, and arranges them as a graph based on the partial order defined by the implication relation. With this graph, given a constraint C, we can query the sub-constraints which imply C, as well as the sub-constraints which C implies, as we will see later. This is useful to improve the efficiency of implication checking in logical subset and superset querying. IPOG nodes are labeled by a sub-constraint and have references to all trie nodes whose input edge is labeled with exactly this sub-constraint. Through these references, it is possible to trace all the occurrences of a given sub-constraint.

Storing the constraints. Everytime a constraint is solved (or it is proved to be unsatisfiable), SCS (respec-

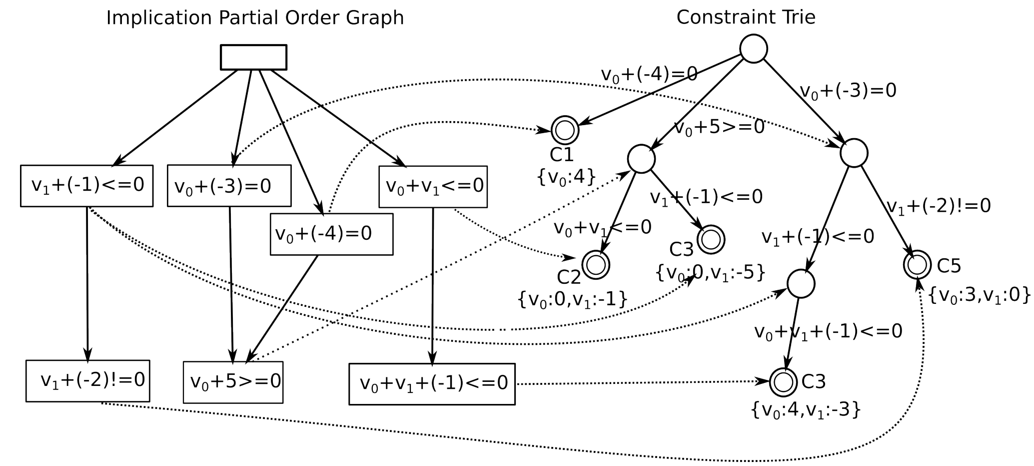

Figure 2: The structure of constraint stores in L-Trie (both UCS and SCS have the same structure).

tively, UCS) must be updated to store possibly new sub-constraints that were not found before, as we describe hereafter. Let $C = C_1 \wedge C_2 ... \wedge C_n$ be the solved constraint in canonical form. Constraint C can be represented by the set $<C_1, C_2, ...C_n>$, where each element is an atomic sub-constraints. This set is sorted by the lexicographic order that yields the canonical form. C_1 (respectively, C_n) is called the leftmost (respectively, rightmost) sub-constraint of C. The storage procedure proceeds as follows:

1. Starting from the trie root node, we consider the (possibly empty) maximal path whose labels coincide with a prefix $C_1 C_2 ... C_i$ of C [1].

   (a) If C_i labels the input edge of a leaf node, it means that we found a logical subset of C in the trie. In the case of the SCS trie, we remove the solution labeling the leaf and append to the leaf a linear subtree with edges labeled $C_{i+1}...C_n$. In the case of the UCS trie, we simply ignore constraint C, which is not saved.

   (b) If i=n and we have not reached a leaf node, it means that we found a logical superset of C in the trie. In the case of the SCS trie, we ignore constraint C and we do not save it. In the case of the UCS trie, we delete the subtree rooted C_i and the node labeled C_i becomes a leaf, which is labeled with C's solution.

   (c) Otherwise, we append a linear subtree with edges labeled $C_{i+1}...C_n$ $C_{i+1}...C_n$ to the trie node labeled $C_i$and add C's solution to the last node labeled C_n.

2. During step 1, whenever we add a new sub-constraint, it will also be stored into IPOG in a way that preserves the partial order defined by the implication relation among atomic sub-constraints.

[1]Note that this procedure ensures that the UCS trie stores the shortest of any two unsatisfiable constraints where one is a prefix of the other, while the SCS trie stores the longest.

6. CONSTRAINT QUERYING

According to Theorem 1, if we want to find a solution for a constraint which has the constraint set C, we should check if any logical subset of C exists in UCS, or if any logical superset of C exists in SCS. Since the constraints are stored in tries, checking for logical subset means that we should find a path from root to a leaf node in the UCS trie so that each constraint in the path is implied by one of the constraints in C. And checking for logical superset means that we should find a path in the SCS trie, so that each constraint in C is implied by one of the constraints in the path.

6.1 Implication Set and Reverse Implication Set

To support efficient check of implication between constraints in C and constraints in trie paths, we introduce the notions of *implication set (IS)* and *reverse implication set (RIS)* of an atomic constraint φ: $IS(\varphi)$ contains all the atomic constraints in UCS which φ implies, whereas $RIS(\varphi)$ contains all the constraints in SCS which imply φ. With the help of *IS* and *RIS*, implication checking can be reduced to checking the existence of constraints in sets.

$IS(\varphi)$ is built by searching the UCS IPOG to find all the constraints in IPOG which φ implies, and $RIS(\varphi)$ is built by searching the SCS IPOG to find all the constraints in IPOG which imply φ. Instead of visiting the whole IPOGs, we only visit the sub-graph which has the same non-constant prefix as φ, since (see Section 2) we exploit the implication relationship between two atomic constraints when they have same non-constant prefix. Because such sub-graphs are often small, the task of building these two sets is always very fast.

6.2 Logical Superset Checking Algorithm

We present an algorithm to check the logical superset of constraint set C in SCS. This algorithm (Algorithm 1) visits the trie bottom-up, from the nodes whose input edges are labeled with constraints that imply the rightmost atomic sub-constraint of C, moving up towards the root node, and checking if the constraints on the path imply the constraints in C.

Function checkSuperset has three parameters: C is a (lexicographically) sorted constraint set to be queried, IPOG

Algorithm 1 Logical superset checking algorithm

/ Check if logical superset of C exists in SCS trie; C is the constraint set to be checked. In this function, rmostRISof(L) is the last element of L, i.e. it is the RIS of the rightmost atomic sub-constraint of C; nodesInTrie(c) is the set of trie nodes referenced by c in IPOG*/*

Function boolean checkSuperset(C, IPOG, Trie)

1. L := empty list; *//the list of RIS*
2. **for each** atomic sub-constraint c in C **do**
3. S := RIS(c, IPOG);
4.   **if** $S = \emptyset$ **then return** false **else** L.add(S);
5. **for each** c in rmostRISof(L) **do**
6.   **for each** n in nodesInTrie(c) **do**
7.   **if** isSuperset(node, L) **then return** true;
8. **return** false;

/ Check if the constraints on the path is a logical superset of the constraint; n is the start node of path; L is the list of RIS */*

Function boolean isSuperset(n,L)

1. cur:=n;//current node
2. pos:=s.size-1; *//current position of L*
3. **while** $cur \neq root$ **do**
4.   **while** $cur.in \in L[pos]$ **do**
5.   pos:=pos-1;
6.   **if** $pos < 0$ **then return** true;
7.   cur:=cur.previous;
8. **return** false;

Algorithm 2 Logical subset checking algorithm

/ Check if logical subset of C exists in UCS trie; C is the constraint set to be checked */*

Function boolean checkSubset(C, IPOG,Trie)

1. S := {}; *//S represents the union of ISs*
2. **for each** atomic c in C **do**
3.   S := S ∪ IS(c, IPOG);
4. **if** $S \neq \emptyset$ **then return** hasSubset (Trie.root, S)
5.   **else return** false;

*/*Recursively check if any logical subset exists in the sub-tree; n is the root of sub-tree ; S is a union set of ISs.*/*

Function boolean hasSubset(n,S)

1. **if** n is leaf **then return** true;
2. **for each** edge in n.out **do**
3.   **if** $edge.label \in S$ **then**
4.   **if** hasSubset (n.next(edge), S) **then return** true;
5. **return** false;

and Trie are the implication partial order graph and the constraint trie in SCS. As shown in lines 1–4 of function checkSuperset, we first build the RIS for each constraints in C and put them into a list L. If one constraint's RIS is empty, then the function returns false, indicating that a logical superset cannot be found in SCS. Lines 5–7 check all the trie nodes referenced by the elements contained in the last RIS of list L; i.e., the nodes whose input edge's labeling constraints imply the rightmost sub-constraint of C. For each of these nodes, function isSuperset checks whether the constraint set on the path from the node to the root is a logical superset of C. If we find such path, then the function returns true, otherwise it returns false. Function isSuperset has two parameters: n is the start node and L is a list of RIS corresponding to each sub-constraint of C. Lines 3–7 visit the trie path from the start node upward to the root. Lines 4–6 repeatedly check if the constraint labeling the incoming edge to the current node is an element of RIS. We use a loop instead of a branch, because it is possible that one constraint on the path implies several constraints in C. Line 6 indicates that if every constraint in C is implied by constraints on the path, then a logical superset is found.

Algorithm 1 shows the benefit on performance of using IPOG as a logic index. Instead of visiting all the trie paths, it only visits a small set of paths from the nodes whose input constraints imply the rightmost sub-constraint of C.

6.3 Logical Subset Checking Algorithm

This section presents an algorithm (Algorithm 2) to check for a logical subset of constraint set C in UCS. The algorithm visits the trie top-down, starting from the root, and selects successive nodes whose input constraints are implied by constraints in C, until a leaf node is reached.

In Algorithm 2, function checkSubset has three parameters: a (lexicographically) sorted constraint set C, and the UCS IPOG and Trie. Lines 2–3 build the union set S of all ISs of atomic constraints in C. If S is not empty (Lines 4–5), function hasSubset is invoked to check if a path exists whose constraints are the logical subset of C. If S is empty, then the function returns false, indicating that no logical subset can be found in the trie. Function hasSubset is implemented as a recursive visit of the trie.

By building the union of all ISs of sub-constraints, this algorithm significantly decreases the complexity of implication checking among edge labels and sub-constraints in C and improves the performance of logical subset checking.

7. EVALUATION

This section presents an experimental evaluation of the performance of the GreenTrie framework. The assessment is performed by considering three scenarios: (1) reuse in a single run of the program, (2) reuse across runs of the same program, (3) reuse across different programs. We compare the performance of GreenTrie with the original Green framework which uses the Redis store and also with the KLEE approach which supports reuse based on simple subset/superset querying from an UBTree store[9].

All experiments were conducted on a PC with a 2.5GHz Intel processor with 4 cores and 4Gb of memory. It runs the LinuxMint 17.1 operating system. We implemented the GreenTrie framework, and integrated it into the well-known symbolic executor Symbolic Pathfinder[21, 22]. In addition, in order to evaluate the KLEE approach, we reimplemented in Java its UBTree data structure and subset/superset querying code (originally available for C++), then integrated it with Green, replacing the Redis store. Thus, whenever hereafter we refer to KLEE, we actually refer to our Java reimplementation.

The experiments that follow are based on seven programs which were used in [10], [21] and [11]:

- TriTyp implements DeMillo and Offutt's solution of Myers's triangle classification problem;
- Euclid implements Euclid's algorithm for the greatest common divisor using only addition and subtraction;
- TCAS is a Java version of the classic traffic collision avoidance system available from the SIR repository;
- BinomialHeap implements a binomial heap;
- BinTree implements a binary search tree with element insertion and deletion;
- TreeMap uses a red-black tree to implement a Java Map-like interface.
- MerArbiter is a component of the flight software for NASA JPL's Mars Exploration Rovers (MER). It has 268 classes, 553 methods, 4697 lines of code.

In all the tables that summarize our experimental results we use the following conventions:

- t_0, and n_0 denote the running time and the number of SAT solving invocations, respectively, for classical symbolic execution without any reuse;
- t_1, and n_1 denote the running time and the number of SAT solving invocations, respectively, when Green is used;
- t_2, and n_2 denote the running time and the number of SAT solving invocations, respectively, when KLEE is used;
- t_3, and n_3 denote the running time and the number of SAT solving invocations, respectively, when GreenTrie is used;
- $T' = (t_1 - t_3)/t_1$ denotes the *time improvement ratio* against Green;
- $T'' = (t_2 - t_3)/t_2$ denotes the *time improvement ratio* against KLEE;
- $R' = (n_1 - n_3)/n_1$ denotes the *reuse improvement ratio* against Green;
- $R'' = (n_2 - n_3)/n_2$ denotes the *reuse improvement ratio* against KLEE.

7.1 Reuse in a Single Run

The first experiment evaluates performance of GreenTrie in a scenario of self-reuse—the constraint solutions generated at previous states are reused for successive constraint solving within the same run. To evaluate how performance scales with the size of a symbolic execution tree, we modify the loop bound of the TreeMap, BinTree, and BinomialHeap programs, thus yielding three versions for each of these three programs. The results are shown in Table 1.

Table 1 shows that GreenTrie achieves an average reuse ratio that reaches 41.38% with respect to Green, and an average reuse improvement ratio of 6.07% compared to KLEE. In addition, GreenTrie also gets a modest improvement in running time of symbolic execution, with respect to both Green and KLEE. The experiment also shows that GreenTrie has better performance in larger scale program analysis, which has more constraints to be solved and costs more in symbolic execution time. For small scale of analysis, GreenTrie may cost a little more time than Green and KLEE, but when the scale grows GreenTrie performs better.

7.2 Reuse across Runs

This section evaluates the performance of GreenTrie in the scenario of regression verification. When a changed program is analyzed, the solution generated by previous runs can be reused in the new run. We evaluate the performance for three groups of changes: addition, deletion, and modification. These are all small changes and are generated manually in order to simulate the real situations in programming. Each group includes 4 version of programs: the first is the base version, and the others are three changed versions. *Changes by addition* are generated by adding branches to a program or adding expressions to program conditions. *Changes by deletion* just undo changes by addition. *Changes by modification* are generated by modifying operators or variable assignments. For each group of changes, we start the evaluation from empty stores, symbolically execute the base version and the three changed versions of programs one by one, and evaluate performance figures for each new version of the program.

Tables 2, 3, 4 show the evaluation results for three of the programs we examined in Section 7.1. The results show that GreenTrie achieves an average reuse improvement of 49.87%, 86.38% and 54.81% with respect to Green, i.e. GreenTrie decreases by more than half the number of evaluated constraints. The reuse improvement against KLEE varies dramatically from case to case. In the case of DEL#2 of program BinTree, it reaches 100.00%, where the number of constraint solved reduces from 599 in KLEE to 0 in GreenTrie.

Considering the average time saving ratio, we obtain values 31.28%, 22.94% and 22.27% against Green, and 36.37%, 0.29%, and 67.39% against KLEE. In the case of a very high reuse ratio, as for TCAS, GreenTrie costs almost the same running time as KLEE, but saves more than 20% running time than Green.

GreenTrie scales better than KLEE. In the case of BinTree, when more than 3000 constraints are accumulated in store, the running time of KLEE increases dramatically. One reason is that KLEE's superset querying algorithm, which searches constraints from root to leaf, performs inefficiently for a large store. GreenTrie instead searches constraints from a limited set of nodes to the root with the help of IPOG, and thus it performs better than KLEE.

Table 1: Experimental results of reuse in single run

Program	n_0	n_1	n_2	n_3	R'	R''	t_0(ms)	t_1(ms)	t_2(ms)	t_3(ms)	T'	T''
Trityp	32	28	28	28	0.00%	0.00%	1040	915	922	995	-8.74%	-7.92%
Euclid	642	552	464	464	15.94%	0.00%	5105	6503	7274	6311	2.95%	13.24%
TCAS	680	41	20	14	65.85%	30.00%	12742	3356	2182	2165	35.49%	0.78%
TreeMap1	24	24	24	24	0.00%	0.00%	871	942	947	882	6.37%	6.86%
TreeMap2	148	148	140	140	5.41%	0.00%	2918	2542	2851	2606	-2.52%	8.59%
TreeMap3	1080	956	833	806	15.69%	3.24%	21849	10729	11809	9871	8.00%	16.41%
BinTree1	84	41	25	25	39.02%	0.00%	1476	1103	1092	1027	6.89%	5.95%
BinTree2	472	238	133	118	50.42%	11.28%	4322	3648	3156	2872	21.27%	9.00%
BinTree3	3252	1654	939	873	47.22%	7.03%	36581	17197	14764	12041	29.98%	18.44%
BinomialHeap1	448	32	23	19	40.63%	17.39%	3637	2137	2046	2017	5.62%	1.42%
BinomialHeap2	3184	190	85	68	64.21%	20.00%	27165	7653	6442	6071	20.67%	5.76%
BinomialHeap3	23320	988	337	288	70.85%	14.54%	249224	28549	31892	21392	25.07%	32.92%
MerArbiter	60648	21	15	13	38.10%	13.33%	>10min	304726	290854	272813	10.47%	6.20%
total/average	94014	4913	3066	2880	41.38%	6.07%	/	390000	374012	341063	12.55%	9.35%

Table 2: Experimental results of reuse across runs (program Euclid)

Changes	n_0	n_1	n_2	n_3	R'	R''	t_1(ms)	t_2(ms)	t_3(ms)	T'	T''
ADD#1	492	432	5	3	99.54%	60.00%	3896	1375	1329	65.89%	3.35%
ADD#2	438	331	216	216	34.74%	0.00%	2830	3275	2284	19.29%	30.26%
ADD#3	220	170	32	2	98.82%	93.75%	1382	972	552	60.06%	43.21%
DEL#1	438	322	156	126	60.87%	19.23%	3428	2670	2171	36.67%	18.69%
DEL#2	492	426	350	134	68.54%	61.71%	3777	4483	2046	45.83%	54.36%
DEL#3	642	552	112	111	79.89%	0.89%	4649	2560	2049	55.93%	19.96%
MOD#1	642	552	464	463	16.12%	0.22%	4851	6899	4400	9.30%	36.22%
MOD#2	642	552	464	462	16.30%	0.43%	4765	7094	4351	8.69%	38.67%
MOD#3	642	551	442	433	21.42%	2.04%	4505	7481	4240	5.88%	43.32%
total/average	4648	3888	2241	1949	49.87%	13.03%	34083	36809	23422	31.28%	36.37%

Table 3: Experimental results of reuse across runs (program TCAS)

Changes	n_0	n_1	n_2	n_3	R'	R''	t_1(ms)	t_2(ms)	t_3(ms)	T'	T''
ADD#1	1036	9	4	2	77.78%	50.00%	1889	1535	1564	17.20%	-1.89%
ADD#2	2920	4	2	1	75.00%	50.00%	3511	2639	2652	24.47%	-0.49%
ADD#3	6730	3	0	0	100.00%	0/0	5015	3577	3576	28.69%	0.03%
DEL#1	2920	0	0	0	0/0	0/0	2675	2051	2077	22.36%	-1.27%
DEL#2	1036	0	0	0	0/0	0/0	912	727	807	11.51%	-11.00%
DEL#3	678	0	0	0	0/0	0/0	632	599	594	6.01%	0.83%
MOD#1	1406	2	2	0	100.00%	50.00%	2322	1917	1801	22.44%	6.05%
MOD#2	1406	4	2	0	100.00%	50.00%	1888	1490	1440	23.73%	3.36%
MOD#3	994	0	0	0	0/0	0/0	1020	817	797	21.86%	2.45%
total/average	19126	22	10	3	86.36%	91.36%	19864	15352	15308	22.94%	0.29%

Table 4: Experimental results of reuse across runs (program BinTree)

Changes	n_0	n_1	n_2	n_3	R'	R''	t_1(ms)	t_2(ms)	t_3(ms)	T'	T''
ADD#1	5930	1689	803	746	55.83%	7.10%	17978	20355	11889	33.87%	41.59%
ADD#2	13358	3938	2618	2556	35.09%	2.37%	35382	105190	32465	8.24%	69.14%
ADD#3	15602	540	0	0	100.00%	0/0	18106	61586	17180	5.11%	72.10%
DEL#1	13358	3149	2216	2185	30.61%	1.40%	32134	126488	31002	3.52%	75.49%
DEL#2	5930	1154	599	0	100.00%	100.00%	13565	44789	10932	19.41%	75.59%
DEL#3	3252	1682	0	0	100.00%	0/0	12945	11482	4505	65.20%	60.76%
MOD#1	3252	1682	1080	1002	40.43%	7.22%	14553	16297	10628	26.97%	34.79%
MOD#2	3252	1680	716	632	62.38%	11.73%	14147	13784	7953	43.78%	42.30%
MOD#3	8310	2377	1068	964	59.44%	9.74%	22772	32889	14593	35.92%	55.63%
total/average	72244	17891	9100	8085	54.81%	11.15%	181582	432860	141147	22.27%	67.39%

7.3 Reuse across Programs

Constraint solutions can also be reused across different programs, especially for programs with similar functionality. Our experiments compare the inter-programs reuse of Green, KLEE, and GreenTrie. We take seven programs in pairs. For each pair, we start with empty stores, and then symbolically execute one program after the other. We compute the difference between the number of solved constraints for the second program when execution starts with the empty store and the number in the case where execution starts with the store generated by the first program. This delta value represents the number of inter-programs reuse.

The results are shown in Table 5. The names of the first-run programs label the rows while the names of the second-run programs label the columns. Each cell contains three numbers: the number of reused constraints when Green is used, the number of reused constraints when KLEE is used, and the number of reused constraints when GreenTrie is used. Table 5 shows that when a program pair has a high reuse level in Green, GreenTrie has an even higher reuse level. And when two programs share almost no constraints in Green, GreenTrie has a few constraints to reuse.

Interestingly, in some cases KLEE has a little more reuse than GreenTrie, as in the case of the program pair TreeMap-BinTree. The reason is that some constraints, which reuse the solution both across programs and in same program in GreenTrie, can only reuse constraints across programs in KLEE. Such constraints are counted for KLEE but not counted for GreenTrie.

8. RELATED WORK

Our work is closely related to the Green framework, but also has some relations with other works on constraint solution reuse and constraint reduction. These are briefly discussed in this section.

8.1 Reuse of Constraint Solutions

The idea of improving the speed of constraint solving by reusing previously solved results is not new. For example, the KLEE [9] symbolic execution tool provides a constraint solving optimization approach named *counterexample caching*, which stores results into a cache that maps constraint sets to concrete variable assignments (or a special No solution flag if the constraint set is unsatisfiable). For example, $\{x + y < 10, x > 5, y \geq 0\}$ maps to $\{x = 6, y = 3\}$, and $\{i < 10, i = 10\}$ maps to No. Using these mappings, KLEE can quickly answer several types of similar queries, involving subsets and supersets of the constraint sets already cached. The subset and superset queries in KLEE are a special case of ours: our logical subset and superset queries fully cover KLEE's subset and superset queries.

Memoized symbolic execution [11] caches the symbolic execution tree into a trie, which records the constraint solving result for every branch and reuses them in new runs. When applied to regression analysis, this allows exploration of portions of the program paths to be skipped, instead of skipping calls to the solver. GreenTrie and Green could work together with this approach to provide further reuse across runs and programs and get better reuse even when the constraints are not same.

The work described in [13] proposes an approach to eliminate constraint solving for unchanged code by checking constraints using the test suite of a previous version. While in the process of exploring states, this approach compares and validates each new path condition with the solution in the test suite of the base version. If the comparison succeeds, it just adds that test case to the new test suite. The work described in [14] presents a technique to identify reusable constraint solutions for regression test cases. The technique finds variables where input values from the previous version can be reused to execute the regression test path for the new version. By comparing definitions and uses of a particular variable between the old and new versions of the application, this technique determines whether the same constraints for the variable can be (re)used. GreenTrie is complementary to these approaches, and is able to provide better reuse when constraints are not syntactically equivalent.

8.2 Constraint Reduction

Reducing the constraint into a short one is a popular optimization approach of SAT/SMT solvers and symbolic executors [9, 7, 8]. For example, KLEE [9] does some constraint reductions before solving: (1) *Expression rewriting*: These are classical techniques used by optimizing compilers: e.g., simple arithmetic simplifications ($x + 0 \Rightarrow x$), strength reduction ($x * 2^n \Rightarrow x << n$, where $<<$ is the bit shift operator), linear simplification ($2*x - x \Rightarrow x$). (2) *Constraint set simplification*: KLEE actively simplifies the constraint set when new equality constraints are added to the constraint set by substituting the value of variables into the constraints. For example, if constraint $x < 10$ is followed by a constraint $x = 5$, then the first constraint will be simplified to true and be eliminated by KLEE. (3) *Implied value concretization*: KLEE uses the concrete value of a variable to possibly simplify subsequent constraints by substituting the variable's concrete value. (4) *Constraint independence.* KLEE divides constraint sets into disjoint independent subsets based on the symbolic variables they reference. By explicitly tracking these subsets, KLEE can frequently eliminate irrelevant constraints prior to sending a query to the constraint solver.

The slicing and canonization of Green framework is also able to reduce the constraints. Constraint slicing is based on constraint independence, and eliminates irrelevant constraints in an incremental way. Canonization is able to reduce the constraint by expression rewriting with arithmetic simplifications. Our approach simplifies the constraint set based on logic relations, therefore it can reduce constraint into a simpler form after slicing and canonization by Green.

8.3 Discussion

The main difference between GreenTrie and other approaches is that it reuses constraint solving results based on the implication relationship among constraints. Green[10], memoized symbolic execution [11], the approaches presented in [13], [14], and [15] are all based on syntactic or semantic equivalence of constraint, while KLEE[9] reuses constraints based on simple implication relationships—subset and superset. GreenTrie includes the capabilities of these approaches to support reuse of constraint solutions. The benefits have been demonstrated in this paper by comparing the degree of constraint reuse achieved by GreenTrie as opposed to Green and KLEE. Other work such as Symstra[23] checks implication between constraints using Omega library and CVC Lite to support state comparison. But this approach is not suitable for constraints reuse, since it is even more expensive than solving the constraint directly.

Table 5: Experimental results of reuse across programs

Program	Trityp	Euclid	TCAS	TreeMap	BinTree	BinomialHeap	MerArbiter
Trityp	/	0, 0, 3	0, 0, 3	0, 4, 4	0, 2, 2	0, 6, 7	0, 0, 1
Euclid	0, 0, 1	/	2, 5, 5	0, 0, 0	0, 3, 4	0, 2, 2	0, 0, 2
TCAS	0, 0, 2	2, 2, 2	/	0, 0, 0	0, 2, 3	0, 3, 4	0, 3, 4
TreeMap	0, 0, 0	0, 0, 0	0, 0, 0	/	256, 326, 323	0, 0, 0	0, 0, 0
BinTree	0, 0, 0	0, 0, 0	0, 0, 0	256, 449, 470	/	0, 1, 1	0, 0, 0
BinomialHeap	2, 2, 5	2, 2, 5	2, 8, 6	0, 2, 3	1, 11, 10	/	0, 0, 0
MerArbiter	0, 1, 2	0, 2	0, 3	0, 0, 0	0, 0, 0	0, 0, 0	/

We also have shown that GreenTrie saves symbolic execution time with respect to Green and KLEE. One reason is that, because of its higher reuse ratio, it invokes the solver less times than Green. Another reason is that the logical superset and subset querying algorithm is performed as efficiently or even better than that in Green and KLEE. As shown in the experiments of Section 7.2, when both GreenTrie, Green, and KLEE all gain high reuse ratios, GreenTrie is still faster than other two approaches.

Unlike Green, which uses Redis to store and query solutions, GreenTrie saves SCS and UCS as two files on disk and loads them into memory when symbolic execution is started. GreenTrie uses almost the same memory as Green for symbolic execution. For example, in the case of Bintree-3 in Section 7.1, GreenTrie uses 284Mb memory, and Green uses 288Mb (including 5M due to the Redis process). GreenTrie also optimizes the space occupied by L-Tries: each expression is an object (a sub-constraint is also an expression composed by smaller expressions), and its occurrences in different constraints in the trie and the IPOG are all references to this object. Since the constraints in symbolic execution are always composed by the same group of expressions/sub-constraints, this optimization significantly decreases the space occupied by L-Tries. As an example, in the case of Bintree-3 the total size of SCS and UCS stores is 387 Kb for 873 cached constraints composed with 81 expressions.

GreenTrie has one limitation compared to the original Green framework: by now GreenTrie is only able to reuse the SAT solving results, and cannot reuse the model counting results (that are utilized to calculate path execution probabilities[24]) as Green instead does.

9. CONCLUSION AND FUTURE WORK

We introduced a new approach to reuse the constraint solving results in symbolic execution based on their logical relations. We presented GreenTrie, an extension to the Green framework, which stores constraints and solutions into two tries indexed by implication partial order graphs. GreenTrie is able to carry out logical reduction and logical subset and superset querying for given constraint, to check if any solutions in stores can be reused. As our experimental results show, GreenTrie not only saves considerable symbolic execution time with respect to the case where constraint evaluations are not reused, but also achieves better reuse and saves significant time with respect to Green and KLEE approach.

Our future work will extend GreenTrie to support more kinds of constraints other than linear integer constraints, through adding implication rules and extending query algorithm, as well as introducing the term rewriting technique[20] to simplify the complex constraints. We also plan to make the summaries in compositional symbolic execution[25, 26] reusable at a finer granularity, considering that the summary is a disjunctive constraint that composed by pre and post conditions of paths of target method. This work is part of our long-term efforts that aim at supporting incremental and agile verification[27, 28, 29].

10. ACKNOWLEDGMENTS

We thank Domenico Bianculli, Srdjan Krstic, Giovanni Denaro, Mauro Pezzè, Pietro Braione for comments and suggestions in various stages of this work. This work was supported by European Commission, Program IDEAS-ERC, Project 227977-SMScom, National Natural Science Foundation of China under Grant No.61272108, No.91118003 and No.61373038, and the National High Technology Research and Development Program of China, No. 2012AA011204-01.

11. REFERENCES

[1] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[2] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, May 2013.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1083–1094, Hyderabad, May 2014. ACM Press.

[4] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[5] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, August 2009.

[6] Saswat Anand. *Techniques to facilitate symbolic execution of real-world programs*. PhD thesis, Georgia Institute of Technology, 2012.

[7] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, pages 263–272, New York, USA, September 2005. ACM Press.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE:automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security*, page 322, New York, USA, October 2006. ACM Press.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 209–224. USENIX Association Berkeley, 2008.

[10] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12 (2012)*, pages 1–11. ACM Press, 2012.

[11] Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, page 144, New York, New York, USA, 2012. ACM Press.

[12] Jeremy Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

[13] Sarmad Makhdoom, Muhammad Adeel Khan, and Junaid Haroon Siddiqui. Incremental symbolic execution for automated test suite maintenance. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 271–276, New York, USA, September 2014. ACM Press.

[14] Md. Hossain, Hyunsook Do, and Ravi Eda. Regression Testing for Web Applications Using Reusable Constraint Values. In *Proceedings of 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 312–321. IEEE, March 2014.

[15] Meixian Chen. Reusing constraint proofs for scalable program analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 449–452, New York, USA, July 2014. ACM Press.

[16] F. Bodon and L. Rónyai. Trie: An alternative data structure for data mining algorithms. *Mathematical and Computer Modelling*, 38(7-9):739–751, October 2003.

[17] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[18] Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 298–302, Berlin, Germany, 2007. Springer.

[19] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Proceedings of 18th International Conference on Computer Aided Verification- CAV'2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94, Berlin, Heidelberg, August 2006. Springer Berlin Heidelberg.

[20] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, pages 411–421, New York, USA, 2013. ACM Press.

[21] Corina S. Pasareanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, February 2013.

[22] Saswat Anand, Corina S. Pasareanu, and Willem Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *the 13th international conference on Tools and algorithms for the construction and analysis of systems*, pages 134–138. Springer-Verlag, March 2007.

[23] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems-TACAS'05*, volume 3440, pages 365–381, Berlin, Heidelberg, April 2005. Springer Berlin Heidelberg.

[24] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, pages 166–176, New York, USA, July 2012. ACM Press.

[25] Patrice Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '07*, pages 47–54, New York, New York, USA, 2007. ACM.

[26] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems -TACAS '08*, pages 367–381. Springer, 2008.

[27] Carlo Ghezzi, Amir Molzam Sharifloo, and Claudio Menghi. Towards Agile Verification. In *Perspectives on the Future of Software Engineering*, chapter Towards Ag, pages 31–47. Springer, 2013.

[28] Carlo Ghezzi, Claudio Menghi, Amir Molzam Sharifloo, and Paola Spoletini. On requirement verification for evolving Statecharts specifications. *Requirements Engineering*, 19(3):231–255, 2014.

[29] Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, and Dino Mandrioli. Syntactic-semantic incrementality for agile verification. *Science of Computer Programming*, 97:47–54, January 2015.