2010

A Slice-based Decision Procedure for Type-based Partial Orders TR-UNL-CSE-2010-0004

Elena Sherman
University of Nebraska-Lincoln, esherman@cse.unl.edu

Brady J. Garvin
University of Nebraska-Lincoln, bgarvin@cse.unl.edu

M Dwyer
University of Nebraska-Lincoln, dwyer@cse.unl.edu

A Slice-based Decision Procedure for Type-based Partial Orders

TR-UNL-CSE-2010-0004

Elena Sherman, Brady J. Garvin, and Matthew B. Dwyer

Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE 68588-0115
{esherman,bgarvin,dwyer}@cse.unl.edu

Abstract. Automated software verification and path-sensitive program analysis require the ability to distinguish executable program paths from those that are infeasible. To achieve this, program paths are encoded symbolically as a conjunction of constraints and submitted to an SMT solver; satisfiable path constraints are then analyzed further.
In this paper, we study *type*-related constraints that arise in path-sensitive analysis of object-oriented programs with forms of multiple inheritance. The dynamic type of a value is critical in determining program branching related to dynamic dispatch, type casting, and explicit type tests. We develop a custom decision procedure for queries in a theory of *type-based partial orders* and show that the procedure is sound and complete, has low complexity, and is amenable to integration into an SMT framework. We present an empirical evaluation that demonstrates the speed and robustness of our procedure relative to Z3.

1 Introduction

Recent years have witnessed an explosion in research on path-sensitive program analysis, e.g., [1–4]. These approaches have the potential to achieve significantly greater precision than more traditional program flow analyses. Increased precision is possible because the analyses are able to ignore program behavior on *infeasible paths*—sequences of program statements that are not executable in any run of the program—and thus more accurately reflect the program's semantics. Analyses avoid infeasible paths by calculating a symbolic characterization of constraints on input values that govern the execution of a path. This characterization is referred to as the *path condition*.

A path condition includes a constraint for each conditional branch in the program. For example, an integer constraint would be generated both for explicit branches like `if(x > 0){ ... }` and implicit branches such as those embedded in array bounds and divide-by-zero checks. Constraints over a variety of domains, and theories, are needed to encode path conditions for non-trivial programs. In addition to the theory of linear integer arithmetic, the theories of uninterpreted

functions, extensional arrays and fixed-size bit vectors are commonly used [2, 3]. This diversity of constraints makes Satisfiability Modulo Theory (SMT) solvers, such as CVC3 [5] and Z3 [6], particularly well suited to reason about path-sensitive program behavior.

The theories supported by modern SMT solvers are somewhat limited, and mapping the data types in modern programming languages onto those theories can lead to inefficiency and imprecision. Consequently, there is significant interest in enriching the theories supported by SMT solvers to better match the needs of program analysis clients. For example, last year alone there were several papers reporting on decision procedures for theories of strings and on the reductions in cost and improvements in precision that arise from using those theories in reasoning about programs [7–9].

In this paper, we identify a fragment of the theory of partial orders, *type-based partial orders* (TPO), that is sufficient for reasoning about type constraints arising in object-oriented programs. In particular, it comprises the constraints due to dynamic dispatch, explicit subtyping tests (such as Java's `instanceof`), and typecasts. Furthermore, in contrast to the general theory of partial orders, under this smaller fragment we are able to adapt existing approaches for evaluating type tests at runtime to implement a standalone decision procedure for TPO, TPO-DP. It is amenable to inclusion in the DPLL(T) framework for SMT solvers [10] because it is incremental, restartable, and capable of calculating equalities for propagation as well as unsatisfiable cores.

We have evaluated TPO-DP on a set of challenging benchmarks that are generated from a characterization of the type constraints encountered in path-sensitive program analyses. The results of the evaluation demonstrate that TPO-DP performs significantly better than Z3 using the theory of uninterpreted functions and an axiomatization of TPO.

In Sect. 2 we provide background on the nature of the type constraints that require support, discuss decision procedures that are capable of reasoning about partial orders, and describe existing approaches to efficiently evaluating type tests at runtime. We present a TPO-DP in Sect. 3; proofs of soundness, completeness and time and space complexity are included. An evaluation of TPO-DP and a discussion of our findings are presented in Sect. 4. Sect. 5 concludes with a discussion of several approaches that might be taken to further extend TPO-DP.

2 Background and Related Work

Figure 1 illustrates how type constraints arise when performing a path-sensitive analysis. Consider a modular analysis of method `A.f()` that begins on entry to the method. Initially, it can be inferred that the dynamic type of the implicit receiver object, `this`, is a subtype of `A`, but is not `A` itself because there can be no instances of an abstract class. The right side of the figure illustrates the constraints on the type of `this`, denoted t, that arise during the analysis; $t \preceq A$ means that t is a subtype of `A`. The root of the tree expresses the constraints arising from the definition of the type hierarchy. The edge from the root corre-

```
abstract class A {
  void m() {...};
  void n() {...};
  void f() { m(); n(); }
}
class B extends A {
  void m() {...}; }
class C extends A {
  void n() {...}; }
```

Fig. 1. Simple dynamic dispatch example (left) and type-related branching in symbolic execution tree (right)

sponds to the constraints on entry. At the call site to method `m()` there are two possibilities: either `A.m()` or `B.m()` is invoked depending on the type of `this`; the type constraints describing these situations label the second level of tree edges. Similarly at the call site to method `n()` the receiver type determines the invocation of `A.n()` or `C.n()`.

An analysis that does not consider type constraints must consider all four of the inter-procedural paths through `A.f()`. This can be costly in terms of both analysis time and precision since, for example, the analysis of the sequence `B.m()` followed by `C.n()` may produce results that are not reflective of executable program behavior. In this example, two of the four paths, marked with $\times$ in the figure, are infeasible and can be eliminated from analysis, thereby increasing both analysis speed and precision.

2.1 A Fragment of the Theory of Partial Orders

A type hierarchy is a partial order $(T, \preceq)$ where T is the set of types and $t_0 \preceq t_1$ holds if and only if t_0 is a subtype of t_1. The relevant semantics of the subtype relation are completely captured by the definition of a partial order: $\preceq$ is reflexive (every type is a subtype of itself), transitive (a subtype of a subtype is itself a subtype), and antisymmetric (mutual subtypes must be identical).

Let x be the dynamic type of a given value on a given path through the program. The domain of x is T. An analysis seeks to determine whether a path permits a constraint-satisfying assignment to x; otherwise the path must be infeasible.

Since most widely used object-oriented languages, such as Java and C++, do not support computing with types directly[1], constraints of the form $x \preceq y$, $x = y$, or $t_0 \preceq x$ (where y is another type variable) will not arise during analysis. TPO is the fragment of partial orders that is free of such constraints.

[1] Language support for type reflection can provide a form of type based computation, but we defer its treatment to future work.

Within this fragment, we distinguish two classes of constraints. Constraints of the form $t_0 \preceq t_1$ and $t_0 \not\preceq t_1$ are used to encode the type hierarchy. These constraints are known ahead of time and do not directly arise during path-sensitive analyses. Constraints that do arise in path conditions may take on four forms: $x \neq t_0$ (e.g., no dynamic type is abstract), $x = t_0$ (e.g., a newly constructed object's type is known exactly), $x \preceq t_0$ (e.g., when successful dynamic type checks imply one of the object's supertypes), or $x \not\preceq t_0$ (e.g., when failed dynamic type checks eliminate a possible supertype). We refer to constraints with negations as *negative* and all others as *positive*.

When encoding a tree of TPO constraints, as in Fig. 1, we observe that the type hierarchy constraints can only appear in the root of the tree. This fact can be leveraged by incremental TPO decision procedures. The lack of constraints between type variables in TPO can also be leveraged since a TPO query can be decomposed into separate per-variable sub-queries that are each solved independently.

2.2 Deciding Partial Order Queries

Techniques for deciding partial order queries are readily available. There have been a rich body of work and also tool development related to the Bernays-Schönfinkel class of formulae, which subsumes partial order queries. Tools such as iProver [11] and Darwin [12] have been shown to be quite effective on such formulae. But in our setting, these tools have the disadvantage of not supporting incremental query checking, a common occurrence in path-sensitive analyses. While we could, in principle, compare TPO-DP to these tools, the comparison would be unfairly and severely biased in our favor. Therefore we chose a different approach for comparing our procedure to the state of the art.

Researchers have also explored support for the Bernays-Schönfinkel class of formulae in DPLL(T) solvers [13], but to the best of our knowledge no widely available solver implements those techniques. However, Z3 does offer heuristic quantifier instantiation, the underlying technique exploited by, for example, iProver. TPO queries can be encoded in Z3's theory of uninterpreted functions with an appropriate axiomatization of partial orders. Moreover, using Z3's support for the SMT-LIB Command language [14], complex TPO queries can be solved faster than on other tools we experimented with. In Sect. 4 we provide more detail on exactly how Z3 was configured for our evaluation.

2.3 Efficient Type Tests

As with runtime type tests, the key to an efficient TPO decision procedure is the encoding of the type hierarchy T. Two obvious alternatives are the transitive closure of $\preceq$ represented as a Boolean matrix and the transitive reduction of $\preceq$ as a linked structure. The former offers constant time subtyping tests, but it requires $\Theta(|T|^2)$ space and as much time to setup. Thus it can be costly on realistically sized hierarchies, e.g., Java 6 has 8865 classes in its standard library

[15]. On the other hand, representing the transitive reduction as a linked structure needs only $\Theta(|T| + r)$ space, where r is the size of the transitive reduction, though subtype tests take $\Theta(h)$ time, where h is the height of the hierarchy.

There has been a significant body of research on finding novel encodings to improve this space/time trade-off, especially on the memory front [16–19]. But viewed as solving a satisfiability problem, these runtime tests never consider anything except a conjunction of two constraints in the pattern $(x = t_0) \wedge (x \preceq t_1)$ where t_0 is the object's known type, and t_1 is the type it is being checked against. As modular symbolic execution doesn't always have exact type information, we concentrated on encodings where longer conjunctions of constraints could be supported. This requirement led us to use the Type Slicing (TS) encoding [17] as the basis for TPO-DP.

TS constructs a compact representation of a type hierarchy by partitioning T and ordering the types within each partition. Let $T = \bigcup_{i=1\ldots k} \mathcal{T}_i$, where all $\mathcal{T}_i$ are disjoint; each individual $\mathcal{T}_i$ is called a *slice*, and the partitioning is termed a *slicing*. Further define $D_i(t)$ to be the descendants of t in slice i, namely $\{t' \mid t' \preceq t\} \cap \mathcal{T}_i$. Then we denote the ordered elements of a slice with square brackets, e.g., $[\mathcal{T}_i]$. Similarly, $[D_i(t)]$ designates the elements of $D_i(t)$ in the order given for $\mathcal{T}_i$. The essence of TS is the requirement that every $[D_i(t)]$ be a substring of the corresponding $\mathcal{T}_i$. In other words, the descendants of every type must be ordered contiguously in every slice. Once this property is established, determining whether one type t_0 is a subtype of another, t_1, is a two-step process. First we must locate t_0 in the slicing. Then we must compare its position to the bounds of the interval occupied by $[D_i(t_1)]$ in the same slice. The operation is constant time.

The TS encoding uses two integers per type to store that location information: one index indicating which slice it occupies and another giving its position in that slice's order. Additionally, for every type/slice pair there must be an entry to track the upper and lower bounds of $[D_i(t)]$. Hence, the space complexity is in $\Theta(k\,|T|)$ where, recall, k is the number of slices. As we show later in Sect. 3.2, for TPO-DP we also want to minimize k.

Algorithms for constructing a minimal number of slices are exponential, but greedy algorithms have proven effective on real type hierarchies. They operate by building a slice by repeatedly adding a single type. That type is added by attempting to insert it into each existing slice in succession. If insertion into an existing slice is not possible without violating the contiguity requirement, a new slice is created with t as the sole member.

Even though in the worst case $k \in \Theta(|T|)$, the number of slices k is usually very small compared to $|T|$. For instance the Java 6 hierarchy of `java.*` and `javax.*` packages contains $5,632$ types which can be partitioned into 12 slices. Table 3 in [19] provides more k values for different Java releases using different slicing encodings. That data confirms that k is at least two orders of magnitude smaller than $|T|$ in practice. While the TS encoding is not in that table, we found it better in practice than the ESE encoding that is mentioned.

We illustrate the TS encoding and type test evaluation approach on the simple type hierarchy from Fig. 1. Assume that A, B and C types are divided (sub-optimally) into 2 slices: $[A, B]$ and $[C]$. According to TS, the encoding is $A = (1, 1, ([1 \dots 2], [1 \dots 1]))$, $B = (1, 2, ([2 \dots 2], []))$ and $C = (2, 1, ([], [1 \dots 1]))$. The first element of each tuple is the slice the type belongs to, the second is the type's position in that slice's order, and the third is the per-slice descendant intervals. To evaluate $C \preceq A$, we first determine that C lies in slice 2 at index 1. A's descendant interval for slice 2 is $[1 \dots 1]$, which includes C's position. So $C \preceq A$ holds. Checking $C \preceq B$, C's index cannot lie in the second descendant interval of B because that interval is empty; $C \not\preceq B$.

3 Decision Procedure for Type Partial Orders (TPO-DP)

We determine the set of assignments that satisfy a conjunction of literals by computing the set of assignments that satisfy each literal on its own and then taking the intersection of these sets. A formula is satisfiable if this intersection is nonempty.

Under the slicing used in TS, a set of assignments can be expressed as the union of a set of intervals in the slices' orderings. Consider the example encoding at the end of the previous section. The constraint $x \preceq A$ would have its assignments encoded as $\{[1 \dots 2]\}$ for the first slice and $\{[1 \dots 1]\}$ for the second. Because of the contiguity of descendants (the contiguity of equal types is trivial), assignments for a positive constraint will form at most one interval in each slice. Similarly, negative constraints are satisfied by the complement set of assignments, which can be expressed in at most two intervals per slice. In contrast the general theory of partial orders affords no such guarantees.

To support pushing and popping constraints, we keep an explicit stack of these sets for each slice. Take, for instance, the same example encoding and two operations: pushing $x \neq A$ and $x \not\preceq C$. The first would push the interval set $\{[2 \dots 2]\}$ on the first slice's stack and $\{[1 \dots 1]\}$ on the second's. $x \not\preceq C$ would be encoded as $\{[1 \dots 2]\}$ and $\varnothing$, so the intersections of the respective interval unions would be $\{[2 \dots 2]\}$ and $\varnothing$; these sets become the new tops of the stacks when $x \not\preceq C$ is pushed.

3.1 Soundness and Completeness

Because we are computing intersections of sets of satisfying assignments, soundness and completeness depend on the correctness of the TS slicing algorithm: for all t, the set $\bigcup_{i=1 \dots k} D_i(t)$ as computed from the slicing must be exactly its set of descendants. Unfortunately, the TS paper does not contain a formal demonstration of correctness. We provide one here with a summary of the algorithm.

Data Structures To represent a slice $[\mathcal{T}_i]$, the TS slicing algorithm uses a data structure called an ordered list, which maintains a total order on a nonempty set of records. That set is initially a singleton containing a special record that

we will designate $\star$. More records can be added by specifying the element to add and the element before which it should appear, and records can be removed provided that the set remains nonempty. An ordered list is also able to answer queries about which of two records comes earlier in the total order. We say $x \lhd y$ to mean that x and y are in the ordered list, and x precedes y. Similarly, $x \trianglelefteq y$ also indicates that both are in the list, but y does not precede x. It should be clear from the context which ordered list we are referring to when we use the symbols $\star$, $\lhd$, and $\trianglelefteq$.

We will also assume that we can iterate over a subrange of an ordered list and obtain the list's first element with a function called head$(\cdot)$. Neither behavior is required by the ordered list interface, but at the very least we can mirror the contents of each $[\mathcal{T}_i]$ in another structure.

The other data structures used by the slicing algorithm are T, which is the set of types, l_i and r_i, which encode the left and right endpoints of each $[D_i]$ by mapping types to types, and λ_i, which maps types to non-negative integers for bookkeeping purposes. We use parentheses to denote a map look-up; e.g. $\lambda_4(t_8)$ represents the natural number corresponding to the key t_8 in the fourth λ map. For simplicity, we will assume that $\star$ may be used in place of a type, and that if there is no corresponding value for a key, the special constant "invalid" is returned.

Invariants The TS algorithm encodes types by looping over them in topological order (ancestors first), repeatedly invoking a method called insert. As the base case to our inductive proof, we will show that invariants below hold when $T = \varnothing$, and as the inductive step we will demonstrate that calls to insert preserve their truth.

1. All types in T appear in exactly one $[\mathcal{T}_i]$.
2. No types outside of T appear in any $[\mathcal{T}_i]$.
3. For every type $t \in T$, $l_i(a) \trianglelefteq t \lhd r_i(a)$ is satisfied if and only if a is an ancestor of t.
4. For any $t \in T$, $\lambda_i(t)$ is exactly the number of types u for which $l_i(u) \lhd t \lhd r_i(u)$. Note that the first relation is $\lhd$, not $\trianglelefteq$, so $\lambda_i(t)$ is not necessarily the count of t's ancestors.

The first three invariants amount to the correctness of the slicing.

The Base Case Before any function calls, T is empty, so the first, third, and fourth invariants vacuously hold. Likewise all of the $[\mathcal{T}_i]$ are empty, implying the second invariant.

The Function insert The function insert appears in Fig. 2 taking three arguments: the current type t and its set of ancestors A (recall that $t \in A$ should hold). Line 1 adds t to the set T, and then lines 2–8 iterate through the slices looking for a place to put it. The call at line 3 finds a position in the i^{th} slice

insert(t, A)

```
1  let T ← T ∪ {t};
2  for i ∈ [1 ... k] do
3  |  let s ← getValidCandidate(i, A);
4  |  if s ≠ invalid then
5  |  |  insertInExistingSlice(i, t, s, A);
6  |  |  return;
7  |  end
8  end
9  insertInNewSlice(t, A);
```

Fig. 2. Inserting a New Type into a Slicing

insertInNewSlice(t, A)

```
10 let k ← k + 1;
11 insert t before ⋆ in [T_k];
12 for a ∈ A do
13 |  let l_k(a) ← t;
14 |  let r_k(a) ← ⋆;
15 end
16 let λ_k(t) ← 0;
17 let λ_k(⋆) ← 0;
```

Fig. 3. Inserting a New Type into a New Slice

where t can be inserted; if such a location exists, the slicing is updated on line 5 and the processing of t finishes. But if there are no viable locations in any of the slices, line 9 allocates a new slice for t.

T is never modified by other functions, and by the inductive hypothesis every type in $T \setminus \{t\}$ is accounted for in the slicing. As types are never removed from a slice, insert's only option under the first two invariants is to place t exactly once into some $[\mathcal{T}_i]$. Here we will point out that after a call to either insertInExistingSlice or insertInNewSlice, insert immediately returns; it is enough to demonstrate that both of these subroutines make a single insertion of t.

The control structure of insert also lays out our proof for the third and fourth invariants: we will show that insertInNewSlice always preserves them, and that insertInExistingSlice maintains their truth when s is the value returned from getValidCandidate.

The Function insertInNewSlice The code for the simpler case, insertInNewSlice, is given in Fig. 3. Line 10 increments the number of slices and places t in the newly allocated $[\mathcal{T}_k]$. The loop at line 11 sets new left and right boundaries for the descendants in this interval of each ancestor, and finally lines 16 and 17 setup the map λ_k.

insertInExistingSlice(i, t, s, A)

```
18 insert t before s in [T_i];
19 let λ_i(t) ← λ_i(s) + |{a ∈ A | r_i(a) = s}|;
20 let λ_i(s) ← λ_i(s) + |{a ∈ A | l_i(a) = s}|;
21 for a ∈ A do
22 |  if l_i(a) = invalid then
23 |  |  let l_i(a) ← t;
24 |  |  let r_i(a) ← s;
25 |  else if l_i(a) = s then let l_i(a) ← t;
26 end
27 for n ∈ (T \ A) do
28 |  if r_i(n) = s then let r_i(n) ← t;
29 end
```

Fig. 4. Inserting a New Type into an Existing Slice

Clearly line 11 must be executed exactly once; the first two invariants are preserved.

Next we note that line 10 is the only line of the algorithm that modifies k, and none of l_i, r_i, λ_i, or $\mathcal{T}_i$ are touched when $i > k$. Thus, immediately after Line 10, l_k, r_k, and λ_k are all empty while $\mathcal{T}_k$ contains only $\star$.

By the inductive hypothesis, the third invariant already holds for types in slices other than $\mathcal{T}_k$, i.e. every type except t. Because l_k and r_k do not have t's non-ancestors as keys, it also holds for t in the non-ancestor case. The remaining case, that each ancestor a of t satisfy $l_i(a) \trianglelefteq t \lhd r_i(a)$, is covered by lines 12–15: $t \trianglelefteq t \lhd \star$.

Looking at the fourth invariant, lines 16 and 17 are sufficient if there is no type u for which $l_i(u) \lhd \star \lhd r_i(u)$ or $l_i(u) \lhd t \lhd r_i(u)$. Indeed, if there were such a u then $[\mathcal{T}_k]$, which on exit contains only $\star$ and t, would in contradiction hold three distinct elements.

The Function insertInExistingSlice Figure 4 lists the function insertInExistingSlice. Line 18 puts t into the slice $[\mathcal{T}_i]$, the parameter s determining where. Lines 19 and 20 update the map λ_i, and the bounds on the descendant intervals of t's ancestors are selectively updated by the loop on lines 21–26. Those of t's non-ancestors are corrected by lines 27–29.

Like line 11, line 18 is executed once per call, finishing the proof of the first and second invariants.

Immediately after line 18, the types u for which $l_i(u) \lhd t \lhd r_i(u)$ holds are exactly the types u such that, in one case, $l_i(u) \lhd s \lhd r_i(u)$ or, in the other, $l_i(u) \lhd s = r_i(u)$. Those in the former case are counted by $\lambda_i(s)$. Because $l_i(u) \lhd r_i(u)$ whenever l_i and r_i contain the key u, the latter are the types that have $r_i(u) = s$. Hence, after line 19, $\lambda_i(t)$ is too small by $|\{n \in (T \setminus A) \mid r_i(n) = s\}|$. This error is corrected by the final loop in lines 27–29.

The types u satisfying $l_i(u) \lhd s \lhd r_i(u)$ are unchanged until the loop at line 21. There, every ancestor a having $l_i(a) = s$ is remapped by line 25 to t. As t immediately precedes s in $[\mathcal{T}_i]$, the other side of the if statement has no effect on the map λ_i, so $\lambda_i(s)$ is also consistent with $[\mathcal{T}_i]$, and the fourth invariant is kept.

For the third invariant we proceed by cases, dividing the types according to whether they are ancestors of s and t.

First, take a common ancestor a of s and t. It is certainly true that $s \lhd r_i(a)$. If $l_i(a) \neq s$ then $l_i(a) \lhd s$, so the insertion already guarantees that $l_i(a) \trianglelefteq t \lhd r_i(a)$. Otherwise, the conditional on line 25 applies for the same conclusion.

Next, consider a type n that is an ancestor of neither s nor t. The type t will only be put into n's interval if $r_i(n) = s$ on entry to the function, a case that is caught and corrected by line 28.

Now let u be an ancestor of t that is not an ancestor of s. If u has no interval in slice i beforehand, neither line 25 nor line 28 can apply, but lines 23 and 24 will cause its new interval to enclose just t, which is correct. Or, if $r_i(u) = s$ then the insertion correctly places t in u's interval and none of the subsequent conditional apply. The behavior is incorrect if $r_i(u)$ is neither invalid nor s.

Finally, take v to be an ancestor of s but not of t. When $l_i(v) = s$, the insertion correctly does not place t as a descendant. Otherwise the result is wrong.

Therefore, to establish the third invariant we need getValidCandidate to ensure two conditions: every ancestor of t that is not an ancestor of s must have r_i map it to invalid or s, and every ancestor of s that is not an ancestor of t must have l_i map it to s.

The Function getValidCandidate The purpose of getValidCandidate, in Fig. 5 is to find a type s such that insertInExistingSlice can insert t before s. Lines 30–39 find the minimum (with respect to $\lhd$) valid $l_i(a)$, maximum valid $r_i(a)$, and the number of $l_i(a)$ values (and hence $r_i(a)$ values) that are valid. If $r \lhd l$, line 40 gives up; otherwise lines 41–45 test every s satisfying $l \trianglelefteq s \trianglelefteq r$. The function can also give up if all of these tests fail, on line 46.

Though a behavior to be avoided for efficiency's sake, giving up is always correct. Thus, to finish our proof of the third invariant we will assume that $l \trianglelefteq r$ and discuss what happens when execution returns from lines 41, 42, and 44.

We start by observing that if the return value s is not equal to r, every ancestor of t is either an ancestor of s or mapped by l_i and r_i to invalid. And if $s = r$, the interval for any ancestor having descendants in the slice must begin before s and end no earlier than s. Thus, in either case, every ancestor of t that is not an ancestor of s has r_i map it to invalid or s. The first condition is fulfilled.

Next, we point out that, by the fourth invariant, $\lambda_i(s)$ is the number of ancestors of s that l_i does not map to s. Thus, the second condition amounts to all of the ancestors counted by λ_i being ancestors of t. On lines 41 and 42 we explicitly check that this is the case. On line 44, we already know that every

getValidCandidate(i, A)

```
30 let l ← head([T_i]);
31 let r ← ⋆;
32 let c ← 0;
33 for a ∈ A do
34 |  if l_i(a) ≠ invalid then
35 |  |  let c ← c + 1;
36 |  |  if l ◁ l_i(a) then let l ← l_i(a);
37 |  |  if r_i(a) ◁ r then let r ← r_i(a);
38 |  end
39 end
40 if r ◁ l then return invalid;
41 if λ_i(l) = |{a ∈ A | l_i(a) ◁ l ◁ r_i(a)}| then return l;
42 if λ_i(r) = |{a ∈ A | l_i(a) ◁ r ◁ r_i(a)}| then return r;
43 for m ∈ [T_i] | l ◁ m ◁ r do
44 |  if λ_i(m) = c then return m;
45 end
46 return invalid;
```

Fig. 5. Determining Where in an Existing Slice a New Type can be Inserted

ancestor a mapped by l_i and r_i, those counted by c, satisfies $l_i(a) \lhd m \lhd r_i(a)$. Thus, the third invariant holds and the proof is complete.

3.2 Time and Space Complexity

Because TPO-DP uses the TS encoding of T we separate the description of time and space complexity into the TS slicing complexity and the decision procedure complexity.

The time complexity of greedily creating slices is determined by the cost of trying to insert each type into a slice, multiplied by the number of types $|T|$ and the number of slices k. The TS insertion attempts take $O(|A(t)|)$ time where $A(t)$ designates the type's ancestors [17]. So the worst case for preprocessing is certainly in $O(k\,|T|^2)$. Note that this is only an upper bound—we have not developed a tighter bound. Also recall that k grows very slowly with respect to $|T|$, as discussed in Sect. 2.3.

The space complexity is dominated by the encoding at $\Theta(k\,|T|)$.

As for the online decision procedure, time and space complexity are closely coupled. We proceed by considering two cases: the cost per slice of pushing a positive constraint and the same cost for pushing a negative one.

For a positive constraint we must take a union of intervals already on the stack and retain only the elements within the interval that correspond to the new constraint. In processing each interval we test whether it should be kept, narrowed, or discarded, so the operation is in the worst case linear in the number of intervals in the union. Note that positive constraints have only one interval per slice, and it is impossible for the push to divide a single interval into many.

Similarly, when a negative constraint is added, each interval in the slice's union may be kept, narrowed, or discarded, but it is also possible to split one of the intervals into two if it encloses the constraint-violating assignments. Thus, the complexity of adding a negative constraint is the same as for a positive constraint—linear in the number of intervals in the union—but in the worst case it may increase this count by one for the next push. The growth in the number of intervals across all slices is bounded by $|T|/2$ though; the maximum is reached when the types in every slice are alternately included and excluded, and all slices have even length.

So let c_p be the number of positive constraints and c_n the number that are negative. In the worst case, the negative constraints come first, amassing as many as $\sum_{i=1}^{c_n} k(i+1)$ intervals on the stack when $k(c_n+1) \leq |T|/2$ and the $|T|/2$ bound is not encountered. If the bound is struck, at the b^{th} constraint, the number of intervals is $\sum_{i=1}^{b} k(i+1)$ for the first b pushes and $\sum_{i=b+1}^{c_n} k(b+1)$ for those that come later. After that, the worst that the positive constraints can do is eliminate no intervals from the tops of the stacks. There will be an additional $\sum_{i=1}^{c_p} k(c_n+1)$ intervals in the first case and $\sum_{i=1}^{c_p} k(b+1)$ in the second.

In total, the first case creates $\Theta(kc_p + kc_nc_p + kc_n^2)$ intervals and the latter builds $\Theta(c_p|T| + c_n|T| - |T|^2)$. Each interval is processed exactly once, so these expressions give the time complexity. It also follows that the worst-case memory complexity is the sum of these counts and the $\Theta(k|T|)$ space taken by the slicing.

Designating the total number of constraints as c, we observe that c_n controls where the complexity of the algorithm lies between $\Theta(kc)$ and $\Theta(|T|c)$. Therefore we believe it may be fruitful to explore transformations that eliminate negative constraints. We propose one such transformation in Sect. 5.

3.3 Incrementality, Restartability, and Unsatisfiable Cores

For a decision procedure to be incorporated into the DPLL(T) framework it should be incremental, restartable, able to propagate equalities, and able to generate explanations for UNSAT cases, i.e. unsatisfiable cores. The TPO-DP meets each of these requirements.

TPO-DP is inherently incremental thanks to the explicit stack. Restarting is also straightforward: the procedure is reset by clearing its stack. For propagating equalities we must merely check after each push if there is exactly one possible assignment.

We have developed two approaches for calculating unsatisfiable cores. Each approach has its own advantages in terms of complexity and core size. However neither approach is completely incremental in nature or guaranteed to produce minimum cores. Below we present these two approaches.

Let u be the first constraint on variable x to make the top of every slice's stack empty. Then the constraints up through u constitute an (likely large) unsatisfiable core U. All sub-cores of U must contain u, because the formula was SAT before u was pushed. Furthermore, if we define a sub-domain T' of T that includes exactly those assignments to x that satisfy u, $U \setminus \{u\}$ is UNSAT on T'.

```
simpleUnsatCore([T_i], J)
1  let U ← ⊤;
2  let t ← head([T_i]);
3  while t ≠ ⋆ do
4  |  let u ← invalid;
5  |  for j ∈ J do
6  |  |  if (x = t) ⇒ ¬j then
7  |  |  |  if (u = invalid) ∨ (e(u) ◁ e(j)) then
8  |  |  |  |  let u ← j;
9  |  |  |  end
10 |  |  end
11 |  end
12 |  let U ← U ∧ {u};
13 |  let t ← e(u);
14 end
15 return U;
```

Fig. 6. Computing the Minimum UNSAT Core for a Single Slice when every Literal Invalidates a Single Interval

Therefore, we mark u and push it onto a new solver instance (effectively limiting the domain to T'), followed by the constraints that preceded it in order, until we obtain u' in the same manner that we obtained u. If u' is unmarked, as it will likely be the first time, we can repeat the procedure with a chance to eliminate more constraints from the core. Otherwise, we terminate, returning the set of constraints pushed on the last solver instance.

The complexity of finding unsatisfiable cores by this process is the cost of checking the formula at most c times, c being the total number of constraints as defined in Sect. 3.2.

For our alternative approach, we present an algorithm to compute a minimum unsatisfiable core on one slice and explain how it can be extended to find a small core for multiple slices, though that core will not necessarily be a minimum or even a minimal core.

We begin by restricting ourselves to a single slice. Suppose for the moment that every conjunct's negation has its satisfying assignments in just one interval, not two. Then we can find a minimum core by applying the algorithm in Fig. 6, where the slice is $[\mathcal{T}_i]$ and the set of literals on the stack is J.

The function simpleUnsatCore processes the slice from left to right, adding literals to the conjunction U so that the types satisfying U and the types not satisfying U are partitioned into the right and the left sides of the slice, respectively. U is initialized to the empty conjunction on line 1, and line 2 sets t, the left-most type satisfying U, to the left-most type in the slice. The loop on lines 3–14 repeatedly selects a literal (with lines 4–11), adds it to U (line 12), and updates t (line 13) until U becomes unsatisfiable and t advances to the position after all types, $\star$ in the notation from Sect. 3.1.

The selection on lines 4–11 takes advantage of the fact that a minimum core must contain a conjunct such that $x = t$ implies the conjunct's negation; otherwise t would be a satisfying assignment to x. Thus, one of the values of j that passes the test on line 6 must be included. We use the notation $e(j)$ to indicate the type immediately after the rightmost type not satisfying j, so that ties are broken on line 7 by selecting the conjunct that invalidates the largest number of assignments to the right of t. If there is a minimum core containing the conjuncts currently in U but using a different literal to invalidate t, the core will still be a minimum core if we substitute the algorithm's selection for the alternative; this tie-breaker is always a safe choice.

We note that some conjunct in J must invalidate t, so u certainly is a legal literal at line 12, despite the initialization at line 4. Furthermore, $\neg u$ is satisfied by a single interval on $[\mathcal{T}_i]$ containing t. Therefore, after line 13, every type to the left of t is either an invalid assignment to a prior conjunct or an invalid assignment to u. Finally, at least one type is eliminated in each iteration, so the algorithm must terminate by exhausting the slice.

However, the algorithm of Fig. 6 is not applicable when some constraints invalidate two intervals; if we are to use it, we must establish a correspondence between constraints and single intervals. The process is straightforward provided that we initialize U not to the empty conjunction, but to a conjunct that is known to appear in some minimum core and that has a set of satisfying types that is not a superset of the another literal's (weaker conditions exist, but are unnecessary for our argument). To match this initialization, we rotate the elements of the slice so that the i^{th} element appears in position i plus some offset, modulo the size of the slice, where we choose the offset so that types satisfying the chosen conjunct are on the right side.

After the rotation, the types invalidated by a literal can only form two intervals if the one interval extends to the rotated slice's left edge and the other reaches the right edge. If the left interval is a subset of the interval invalidated by the chosen constraint, it can safely be ignored—those types are already eliminated by the core. If it is not, it must include all of the types from the slice's left-most type past the right-most type that is invalidated by the conjunct we chose. But then the satisfying assignments to the literal are a subset of the assignments that satisfy the initial conjunct, a contradiction. In summary, the rotation causes every conjunct to correspond to exactly one interval.

Assuming that the core is requested as soon as the stack of conjunctions becomes unsatisfiable, a suitable starting conjunct is readily available: the top-most literal. Because the conjunction was satisfiable before that literal, it must be essential to the core, and moreover, the set of assignments satisfying it cannot be a superset of the satisfying assignments to any other conjunct.

For multiple slices, we find a core for each slice as soon as it runs out of satisfying assignments, and return the union of these cores. Some further optimizations are possible, for instance at lines 2 and 13 of Fig. 6 by advancing t past the types that are already excluded by other slices' cores.

3.4 Implementation Choices

We implemented the TPO-DP in Java without any significant effort to optimize its performance. To perform a fair comparison with Z3, we implemented a parser for SMT-LIB command syntax that prepares inputs for our decision procedure; this allows our DP and Z3 to use exactly the same input.

After initial evaluation, we identified two optimizations that yield a performance benefit. First, when multiple type variables are present, we produce independent instances of the TPO solver for each variable; a problem is SAT if and only if it is SAT in each solver instance. Second, we cache the results of satisfiability check which allows us to avoid processing redundant queries. Such queries can be common in path-sensitive analysis of real programs. For instance, [21] reports on an analysis that caches queries outside of the decision procedure and observes hit-rates above 80%.

4 Evaluation

Our primary research question centers on performance: *How does the performance of the TPO-DP compare to state-of-the-art solvers on queries arising in path-sensitive program analyses?* We begin with an assessment of path-sensitive analysis techniques and their use of decision procedures.

4.1 Categories of Path-sensitive Analyses

A path-sensitive analysis generates a set of closely related queries where longer queries are extended from shorter ones by conjoining additional clauses. There are multiple ways to extend a query, e.g., producing two queries where one conjoins a constraint for a branch predicate and another that conjoins its negation. Such an analysis gives rise to a *query tree* where nodes correspond to calls to check for satisfiability and edges in the tree encode the assertion of clause sets.

Path-sensitive analyses are expensive to apply to real programs. Consequently, *intra-procedural* path-sensitive analyses, i.e., analyses limited to individual methods, were the first to be explored by researchers. It is natural that papers on these techniques report results on leaf methods – methods that do not call other methods. For example, red-black tree implementations can be analyzed to detect complex corner cases in their logic [2]; when tree height is no greater than six such an analysis generates a tree with a few thousand queries. In intra-procedural analysis, dynamic dispatch is not an issue, since calls from the method will not be analyzed, so there is, arguably, little need for a TPO-DP.

Recently, researchers have begun to develop *inter-procedural* path-sensitive analyses and apply them to larger portions of programs. This strategy has been applied, for example, to produce crash-inducing inputs [22] and to detect security faults [21]. Such analyses consider larger portions of programs whose behavior involves many method calls and, consequently, there is a need to reason about non-trivial type constraints related to dynamic dispatch. The authors of [21]

Table 1. Type Constraint-related Observations from Program Trace Data

Program	$\|T\|$	Ex.	k	#Trace	#Obj.	Len.	DD	CC	IN	Type
NanoXML[23]	79	106	3	5	366	41.0	68k	5.5k	0	41%
Weka[24]	611	893	6	14	2.8k	35.5	134k	1.2M	383	36%
Soot[25]	3259	4019	7	1	32k	191.7	458k	1.6M	4.1M	69%

applied their technique to 3 large programs that required from 22 to 188 thousand queries. We note that these query counts were taken after significant optimization of query checks were performed, e.g., caching of query results outside of the decision procedure.

Spurred by advances in automated decision procedures research in path-sensitive program analysis appears to be accelerating. We conjecture that the *next-generation* of path-sensitive analyses will continue to scale to larger portions of program behavior.

4.2 A Population of TPO Queries

Since we do not have access to a next-generation path-sensitive analysis, we performed a pilot study to characterize the size and diversity of type queries across a set of three open-source Java programs of varying size and complexity. For each program, we instrumented its implementation to record the total number of branches taken and, for each object, the sequence of type related branches evaluated and the nature of the predicate being tested. The instrumented programs were executed on a subset of their test suites, and the recorded data were analyzed to produce the summary in Table 1.

We measured the number of classes and interfaces used in each program including all of the application classes' super-types and super-interfaces. This number is shown under the second ($|T|$) column in Table 1. The third column (Ex.) corresponds to the number of subtype, i.e., `extends` declarations, or interface implementation declarations, i.e., `implements`. The number of type slices for each program (k) illustrate the significant compression achieved by the TS encoding. The recorded program traces (#Trace) are partitioned into sub-traces for each object allocated in the program. (#Obj.) reports the average number of objects in a trace that are involved in type constraints and (Len.) the average length of the per-object sub-traces. Across all of the traces for a program we recorded the number of branches related to dynamic dispatch (DD), class casts (CC), and instanceof tests (IN). Finally, we report the percentage of all branches across the traces that involved type tests (Type); these data clearly indicate the need to support TPO constraints. We also note that gathering fine-grain trace data is extremely expensive, and to reduce the cost of our study we ran Soot just once and only collected information for objects of Soot-defined type. Even then the data collection took many hours.

To characterize the per-variable constraints, we processed the per-object trace data to discard type tests that occurred in the same method where the

object was instantiated, since they could be decided trivially with the equality constraints introduced by object allocation. In this way we retained only the tests that modular path-sensitive analysis might see.

From those raw traces we built a summary in the form of a Markov model where each state corresponded to exactly one type test. The training could then proceed deterministically by frequency counting. We used no prior distribution.

Finally, we built the benchmarks directly from the Markov models for each program. For each variable our generator kept a stack of states. Then, to push constraints, it chose a variable randomly and treated its state stack as the prefix of a path in the Markov model; the next state was chosen according to the transition probabilities from the topmost state. This state and one of the corresponding sets of constraints were pushed. As for popping, the generator removed the most recently pushed state, along with its constraints. Moreover, to better mimic the constraints generated by a path-sensitive analysis, if a popped state still had other alternatives to explore, one of these was subsequently pushed.

The generator can be parameterized by the number of objects involved in a trace and the depth of the query tree, thereby allowing us to produce a population of TPO query trees that resemble the three programs but are scaled in several dimensions. Query trees are emitted in the SMT-LIB command format using push/pop to encode the query tree edges[2].

4.3 Comparing to a state-of-the-art SMT solver

We selected Z3 as a point of comparison since it is known to be efficient, supports incremental solving, and can answer TPO queries by instantiating quantifiers in the partial order axioms. Initially our use of Z3 for checking TPO queries resulted in poor performance. It is widely understood that the appropriate selection of solver heuristics is crucial to using a tool like Z3 effectively. To be as fair as possible in our evaluation we contacted the developers of Z3 asking for their advice on configuring the tool and modifying the input files to maximize performance. The developers were very supportive and supplied us with a modified input file that was better suited for Z3's quantifier instantiation techniques [26]. They also suggested that for our queries we should run Z3 with `AUTO_CONFIG=false` to disable heuristics that were unnecessary for TPO problems and, in fact, were hurting performance significantly. Finally, they informed us that for TPO problems Z3 is "effectively" sound and complete, thus when the solver returns an UNKNOWN result it can be interpreted as SAT; the results of our evaluation confirmed this to be the case.

4.4 Results

We decomposed the population of TPO query trees into three groupings based on the number of queries in the tree. The data from [21] combined with the fact

[2] The data and benchmarks used in this study are available at `http://esquared.unl.edu/wikka.php?wakka=TpoDp`

Table 2. Solver performance data across size-based problem categories

Category	Num. Trees	Avg. Size	Avg. SAT	Avg. UNSAT	TPO-DP all	TPO-DP non-TO	Z3 non-TO	Z3 TO
Intra	388	857.6	535.4	322.1	10.2	2.8	464.1	119
Inter	46	22359.1	11665.4	10693.7	12.3	2.4	2233.1	21
Next-Gen.	16	124576.7	64668.6	59908.1	21.1	3.2	831.6	10

that between 36% and 69% of the queries in our pilot studies were related to type tests led us to the following breakdown. Inter-procedural (Inter) analyses perform between a few thousand up to several tens-of-thousands of type-related queries; for our TPO data we selected 4096 as the lower and 65535 as the upper bound for defining this category. Queries trees that were smaller were classified as those that intra-procedural (Intra) analyses could produce, and those that were larger we consider the province of next-generation analyses.

Table 2 reports performance data of TPO-DP and Z3 on the population of TPO queries broken down by grouping. The number of query trees (Num. Trees) and the average number of queries (Avg. Size) in a tree are listed for each grouping. In addition, we report the average number of SAT and UNSAT problems per tree; TPO-DP and Z3 agreed on this in every case—when Z3's UNKNOWN is interpreted as SAT. We ran all jobs under Linux on a 2.4GHz Opteron 250 with 4 Gigabytes of RAM. With a timeout of four hours, Z3 failed to complete some of the larger query trees, and we give the number aborted (Z3 TO); TPO-DP never took more than 72 seconds and completed all problems. We provide the average time to solve a TPO query tree across each category for TPO-DP in seconds (TPO-DP all). For the problems on which Z3 completed, we report the average TPO query tree solve times for TPO-DP (TPO-DP non-TO) and for Z3 (Z3 non-TO) in seconds of user time.

We conjectured that other factors might influence the relative effectiveness of TPO-DP and Z3. Specifically, the diversity in programs that informed our generation strategy was significant and had a non-trivial influence on the degree of branching in the query tree. In addition the number of free variables in the constraints and the depth of the query tree (i.e. the number of the conjuncts) could cause the two techniques to perform differently. Figures 7 and 8 present log-scale plots of the cumulative cost of running the queries while varying the two parameters of our generator. This time we break the data down by program, where N, W, and S stand for nanoXML, Weka, and Soot, respectively. In addition, we indicate the mean run times with the dashed line.

Note that these plots only reflect benchmarks solved by both implementations—the problems which Z3 could complete in four hours. In particular all of the Soot benchmarks with more than two variables timed out, and the bar for Soot at two variables represents a single data point. Similarly, at the depths of 10 and 15 only one Z3 run with the Soot hierarchy completed. In contrast, the data for the nanoXML and Weka benchmarks was not significantly hampered by timeouts.

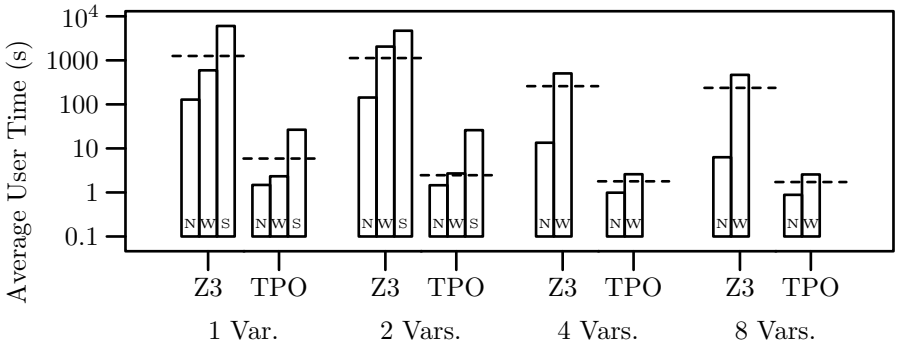

Fig. 7. Z3 and TPO User Times versus Variable Count

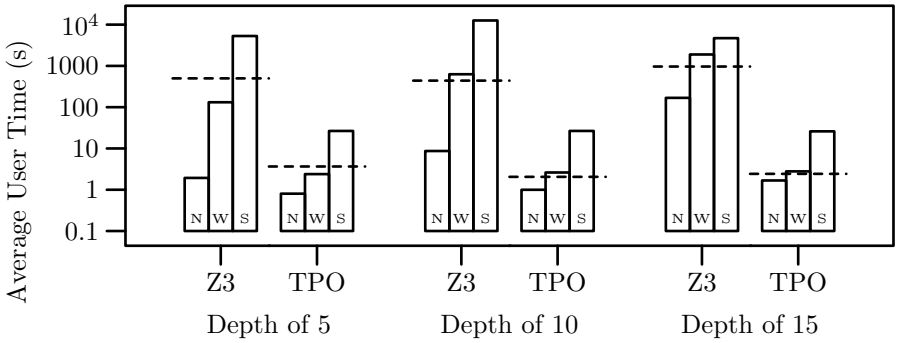

Fig. 8. Z3 and TPO User Times versus Stack Depth

4.5 Discussion

We believe that these results strongly suggest that in advanced path-sensitive program analyses there is a need for custom decision procedures for TPO queries. Such queries arise frequently, and when they do, even state-of-the-art solvers such as Z3 struggle to scale to the size of problems that are characteristic of advanced analyses.

Across all of the different decompositions of the data we collected, TPO-DP outperformed Z3 by a wide margin. Varying the program, the number of variables, or tree depth seemed to have only a modest effect on TPO-DP's performance, except when the number of variables grows large. We believe this latter observation to be more a property of the generated problems, because when the number of variables grows large, for a fixed depth of query tree, the number of constraints that simultaneously involve a single variable will likely decrease. TPO-DP appears to scale well with query tree size, it is relatively stable when moving from the intra-procedural to inter-procedural categories, a 26-fold increase in size on average. This appears to reflect a true benefit of TPO-DP, since Z3's run time increases almost 5-fold across those same categories. We be-

lieve that the performance of TPO-DP in moving from the inter-procedural to next-generation categories also suggests good scalability, a 5.6-fold increase in size gives rise to only a 70% increase in solver time. The analogous data for Z3 is not informative since it times out on 10 of the 16 problems.

5 Conclusions and Future Work

As path-sensitive analyses scale to consider larger portions of object-oriented programs they will invariably encounter large numbers of type-related constraints. Existing methods for solving those constraints are not well-integrated with SMT solvers that support the integer, array, string, and bit-vector theories needed for program reasoning. We developed TPO-DP—a custom decision procedure that leverages results from efficient language runtime systems to efficiently process constraints related to type-based partial orders. We designed and conducted a significant evaluation producing problems representative of those that would be generated by path-sensitive analyses and took care in conducting this evaluation that we did not bias the results in favor of our tool. In this context, TPO-DP outperformed Z3 by a wide margin.

In future work, we plan to explore additional optimizations to our method that will compute type slices for commonly used libraries ahead of time. In addition we will investigate hierarchy transformations that will allow us to introduce a concreteness pseudo-type and thereby convert the many negative constraints for abstract classes into one positive subtype constraint. Our fine-grained complexity characterization suggests that this may further reduce solver time. Finally, we have been approached by the developers of Kiasan [2] who wish to integrate TPO-DP into their analysis engine to study its effectiveness on a variety of analysis problems. In support of that work, we plan to explore extensions of TPO and TPO-DP that support dynamic loading of types and type reflection.

Acknowledgements

The authors would like to thank Robby for bringing the need for a TPO decision procedure to our attention and Xiao Qu for working on a early prototype. Cesare Tinelli, Aaron Stump, and Leonardo de Moura provided helpful comments on an early version of this work and suggestions for developing it further, and we thank them. We could not have conducted a fair comparative evaluation without the help of Nikolaj Bjørner, and we thank him for sharing his insights with us.

This work was supported in part by the National Science Foundation through awards CCF-0541263 and CCF-0747009, the National Aeronautics and Space Administration under grant number NNX08AV20A, the Air Force Office of Scientific Research through award FA9550-09-1-0129, and the Army Research Office through DURIP award W91NF-04-1-0104.

References

1. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11** (2009) 339–353
2. Deng, X., Lee, J., Robby: Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: Proceedings of ASE. (2006) 157–166
3. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Proceedings of TACAS. Volume 4963 of LNCS. (2008) 367–381
4. Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., Levin, M.Y.: Automating software testing using program analysis. IEEE Software **25** (2008) 30–37
5. Barrett, C., Tinelli, C.: CVC3. In: Proceedings of CAV. Volume 4590 of LNCS. (2007) 298–302
6. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of TACAS. Volume 4963 of LNCS. (2008) 337–340
7. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Proceedings of TACAS. Volume 5505 of LNCS. (2009) 307–321
8. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: Combining string analysis and size analysis. In: Proceedings of TACAS. (2009) 322–336
9. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Proceedings of PLDI. (2009) 188–198
10. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (2006) 937–977
11. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Proceedings of IJCAR. Volume 5195 of LNCS. (2008) 292–298
12. Baumgartner, P., Tinelli, C.: The model evolution calculus as a first-order DPLL method. Artif. Intell. **172** (2008) 591–632
13. de Moura, L., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. In: Proceedings of IJCAR. Volume 5195 of LNCS. (2008) 410–425
14. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Technical Report BarST-RR-10, Department of Computer Science, The University of Iowa (2010) Available at www.SMT-LIB.org.
15. Java: Package java.lang, Java^{TM}platform standard ed. 6. (http://java.sun.com/javase/6/docs/api/java/lang/package-summary.html)
16. Zibin, Y., Gil, J.Y.: Efficient subtyping tests with PQ-encoding. In: Proceedings of OOPSLA. (2001) 96–107
17. Zibin, Y., Gil, J.Y.: Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. Proceedings of OOPSLA (2002) 142–160
18. Baehni, S., Barreto, J., Eugster, P., Guerraoui, R.: Efficient distributed subtyping tests. In: Proceedings of DEBS. (2007) 214–225
19. Alavi, H.S., Gilbert, S., Guerraoui, R.: Extensible encoding of type hierarchies. In: Proceedings of POPL. (2008) 349–358
20. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57). North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (2004)

21. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. **12** (2008)
22. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI. (2008) 209–224
23. SIR: Software-artifact infrastructure repository. (http://sir.unl.edu)
24. Weka: Machine learning software. (http://sourceforge.net/projects/weka)
25. Soot: a java optimization framework. (http://www.sable.mcgill.ca/soot/)
26. Bjørner, N.: personal communication (2009)