Automated Property Directed Self Composition

Akshatha Shenoy[1], Sumanth Prabhu[1], Kumar Madhukar[2], Ron Shemer[3] and Mandayam Srivas[4]

[1] TCS Research, Pune, India {shenoy.akshatha, sumanth.prabhu}@tcs.com
[2] Indian Institute of Technology Delhi, New Delhi, India madhukar@cse.iitd.ac.in
[3] Mend.io, Tel Aviv, Israel ronsheme.mail.tau@gmail.com
[4] Chennai Mathematical Institute, Chennai, India mksrivas@hotmail.com

Abstract. We consider the problem of hypersafety verification, i.e. of verifying k-safety properties of a program. While this can, in principle, be addressed by self composition, which reduces the k-safety verification task into a standard $(1-)$safety verification exercise, verifying self-composed programs is not easy. The proofs often require that the functionality of every component program be captured fully, making invariant inference a challenge. Recently, a technique for property directed self composition (or, PDSC) was proposed to tackle this problem. PDSC tries to come up with a semantic self-composition function, together with the inductive invariant that is needed to verify the safety of the self-composed program. One of its crucial limitations, however, is that it relies on users to supply a set of predicates in which the composition and the invariant may be expressed. It is quite challenging even for a user to supply such a set of predicates – the set needs to be sufficiently expressive, so that the invariant can be expressed using those predicates (and their boolean combinations), but not overly expressive to increase the search-space unnecessarily. This paper proposes a technique to automate PDSC fully, by discovering new predicates whenever the given set is found to be insufficient. We present three different approaches for obtaining predicates – relying on syntax-guided synthesis, quantifier elimination, and interpolation – and discuss the strengths and limitations of these.

1 Introduction

A hypersafety or a k-safety property is a program safety property whose violation needs at least k program runs to be demonstrated. Determinism and non-interference are common examples of such properties. A straightforward way to transform a k-safety property into a usual (1-)safety property is *self-composition* [7], in which k memory-disjoint copies of the program are composed with each other. Since the copies are memory-disjoint, the composition may be thought of as an asynchronous parallel composition in which all interleavings have the same behavior. Thus, a hypersafety property that holds in some interleaving holds for all interleavings. A trace of such a composed program naturally corresponds to an interleaving of k traces of the original program, and that is how self-composition reduces a k-safety property to a safety property.

As a technique, self-composition is both sound and complete for k-safety [39]. It also allows us to use the rich literature that exists for verifying (1-)safety properties. However, verifying self-composed programs is not an easy exercise. For instance, if the programs are composed sequentially, proving properties may often require that the functionality of every component be captured fully, and the required invariants can be difficult to obtain even for very simple programs and properties. Since all interleavings (or compositions) behave similarly, it helps to shift the focus of the problem on finding one which is easy to prove correct [18,39].

A recent technique of property directed self composition [39] addresses this problem by appealing to this very insight – that the way the copies are composed determines how complicated it is to verify the composed program. Note that since the copies are memory-disjoint, all compositions are safe if any one of them is proved safe. Informally speaking, PDSC attempts to find an *easy-to-prove* composition and prove that it is safe. It comes up with a semantic self-composition function, together with the inductive invariant that is needed to verify the safety of the program composed according to that function. Since this problem is undecidable in general, it is made tractable by fixing a language of proofs, described by a given set of predicates and their boolean combinations, and navigating the space of all possible compositions to see if one of them can be proved safe by finding an inductive invariant in this language. The algorithm relies on the property that a transition system has an inductive invariant in a language of predicates (and boolean combinations) if and only if its abstraction using those predicates is safe. Thus, by using predicate abstraction, PDSC either obtains an inductive invariant or is able to prove that none exists in the given language.

Interestingly, 2-safety verification is closely related to the task of checking equivalence of two programs. Program equivalence is an important problem owing to its diverse applications, that include translation validation and compiler correctness [31,24,29], code refactoring [35], program synthesis [4], hypersafety verification [3,18,39], superoptimization [37,11], and programming and software engineering education [26] amongst many others. Naturally, self-composition offers a solution for this too, but verifying the composed programs can be quite challenging. Consider, for example, two programs that sum all the numbers in the range $[1, n]$ – even if both the programs are iterating over the digits from 1 to n and adding it to the sum, a sequential composition of these two programs requires non-linear inductive invariants to establish equivalence. An ideal composition in this case would be one where the program statements (or loop iterations) are composed statement by statement, i.e. in *lock-step*. The property itself, that the two sums are equal, becomes an inductive invariant of the lock-step composition. Thus, PDSC becomes a useful technique for addressing the problem of program equivalence as well.

An important caveat of PDSC, however, is that it relies on users to supply a set of predicates in which the composition and the invariant may be expressed. It is quite challenging even for a user to supply such a set of predicates – the set needs to be sufficiently expressive, so that the invariant can be expressed using those predicates (and their boolean combinations), but not overly expressive to increase the search-space unnecessarily. Therefore, it is crucial for the usefulness of PDSC that an automatic way of obtaining these predicates be devised and developed. While there are techniques that can mine predicates (to construct invariants) from program source [20,32,21,22], and PDSC itself proposes to do this in order to lessen the user-dependence, the necessary predicates may very often be absent from the source code (our motivating example, for instance, in Sect. 2). Another limitation of the PDSC algorithm is that it only tries to find an invariant that can establish the given k-safety property. If it fails in doing so, it does not look for a counterexample (a refutation witness).

This paper proposes an algorithm that works on top of PDSC, to *i)* automatically enrich the set of predicates, when it realizes that the current set is insufficient, till a proof is derived, and *ii)* look for a counterexample when it cannot obtain a proof with a given set of predicates, so that new predicates are added only if a refutation witness can also not be derived so far. Though explained later (in Sect. 3.4), the insufficiency of a given set of predicates emerges as an abstract counterexample trace. This trace must be spurious if the property holds, and if the property does not hold then it may correspond to an actual concrete counterexample. Therefore, the purpose of new predicates is to capture the reason for spuriousness. We have designed two different approaches to synthesize new predicates, one that uses a counterexample-guided method of predicate refinement, and another one that obtains them as interpolants from the infeasibility-proof of the counterexample trace. For the first approach, we encode the task as an abduction query, and solve it either using a Syntax-Guided Synthesis (SyGuS) solver (with CVC4-1.8 [6]) or an SMT Solver (Z3 [15]). For the second one, we compute interpolants using MathSAT5 [12]. An experimental comparison of these techniques have been presented in Sect. 6.

The core contributions of this paper are:

1. An improvement of the PDSC algorithm that not only makes it capable to look for proofs as well as refutations, but also removes its user-dependence and enables it to strengthen its proof language iteratively, on demand, in a counterexample-guided way.
2. An implementation on top of PDSC, with three different methods for deriving new predicates – using a SyGuS solver, or an SMT solver, or an interpolating prover. And an experimental comparison of these on several hypersafety verification and program equivalence benchmarks from the literature.

Outline of the paper The rest of the paper is organized as follows. We start with a motivating example in Sect. 2, and then move to the necessary background in Sect. 3, which includes a description of the PDSC algorithm that we build upon. Sect. 4 talks about the challenges and the key contributions that we have made. We describe our algorithm in Sect. 5, and the details of our implementation and experiments in Sect. 6. Sect. 7 discusses the related work, and Sect. 8 contains our concluding remarks.

2 Motivating Example

Consider the example shown in Fig. 1(a) and (b). This is a benchmark from [39]; for ease of understanding, we have presented it as two separate programs, and refer to the underlying 2-safety property simply as an equivalence check. The task is prove that these programs compute the same output value, given the same input. This is indeed true; both the programs take an integer x as input and compute $2 * x^2$, although differently. The first program (`v1`) goes through the while loop $2x$ times, adding x to y each time. The second program (`v2`) goes through the loop only x times, incrementing y by x each time, but doubles the value of y before it returns.

A self-composition approach that does a sequential composition in this case would require that both the programs be completely analyzed individually before the outputs can be compared. For example, one needs to synthesize invariants for loops in both programs separately, which in this case are non-linear expressions: $(0 \leq z \leq 2x) \wedge y = x * (2x - z)$ and $(0 \leq z \leq x) \wedge y = x * (x - z)$, for the versions `v1` and `v2` respectively.

```
dblSqr-v1(x){

  y = 0;
  z = 2 * x;

  while (z > 0) {
    z = z - 1;
    y = y + x;
  }

  return y;
}
```

(a)

```
dblSqr-v2(x){

  y = 0;
  z = x;

  while (z > 0){
    z = z - 1;
    y = y + x;
  }

  y = 2 * y;
  return y;
}
```

(b)

Fig. 1: doubleSquare example, from [39]

An alternative approach could be to analyze their runs in an interleaved fashion, up to selected “checkpoints” in each program. An advantage of using such an interleaved composition for analysis is that the required invariants are likely to be simpler because of the choice of the checkpoints as synchronization points for interleaving. The checkpoints can be chosen where the outputs are expected to be behaviorally equivalent, or if not, then at least there is a linear relation between them. For instance, for the programs shown in Fig. 1, the checkpoints can be added such that the components synchronized after every two iterations of the loop in `v1` and one iteration of the loop in `v2`. If this happens, then at each synchronization point the value of y in `v1` will be double that of y in `v2`. After the loop exit, before the programs return, since `v1` does not run any instruction, while `v2` multiplies its copy of y by 2, it becomes evident – by only tracking linear relation in the variables – that the programs are doing the same thing.

The technical challenge in this approach lies in finding good synchronization points, or equivalently, a suitable composition, that has an *easy-to-find* safe inductive invariant. An additional challenge lies in that the expressiveness of the proof language (i.e., the one in which invariants have to be searched) is dependent on the choice of the composition candidate. In the next section, we will understand how the PDSC technique addresses these concerns, and discuss the limitations and challenges that lie ahead.

3 Background

3.1 Programs, Safety Properties, and Invariants

Similar to [39], we model a program as a transition system that defines its behavior. A transition system is a tuple $T = (S, R, F)$, where S is a set of states, $R \subseteq S \times S$ is a transition relation that specifies an arbitrary step in an execution of the program, and $F \subseteq S$ is a set of terminal states such that every terminal state $s \in F$ has an outgoing transition to itself and no additional outgoing transitions (terminal states allow us to reason about *pre-post* specifications of programs).

An execution (or trace) of the program is given by a sequence of states $\pi = s_0, s_1, ...$ such that for every $i \geq 0, (s_i, s_{i+1}) \in R$. An execution is called *terminating* if its corresponding sequence has the suffix $s_i, s_i, ..$ for some $s_i \in F$, and the terminating execution is said to *end at* s_i.

We denote the set of variables by V, and the transition relation by a formula over $V \cup V'$ where post-states of transitions are over V'. We use sets of states and their symbolic representation via formulas interchangeably.

We consider safety properties defined via a $(pre, post)$ pair, where *pre* and *post* are formulas over V , representing sets of states. T satisfies $(pre, post)$ if every terminating execution of T that starts in a state that satisfies *pre*, ends at a state that satisfies *post*.

An inductive invariant, for a transition system T and a safety property given as $(pre, post)$, is a formula *Inv* such that the following conditions hold.

$$(1)\ pre \Rightarrow Inv,\ (2)\ Inv \wedge R \Rightarrow Inv',\ (3)\ Inv \Rightarrow (F \Rightarrow post)$$

Inv$'$ denotes the formula *Inv* with every variable replaced by its corresponding primed version.

It is noteworthy that any inductive invariant satisfies the first two conditions, while the last condition holds only for an invariant that is sufficiently strong to discharge the given safety property. We sometimes refer to such an inductive invariant, one that satisfies all the three conditions above, as a *safe* inductive invariant, and even as a safety *proof*.

A k-safety property refers to k interacting executions of T, and is also given by a $(pre, post)$ pair, except that pre and $post$ are defined over $V_1 \uplus \ldots \uplus V_k$ where V_i denotes the i^{th} copy of program variables. Naturally, pre and $post$ represent sets of k-tuples of program states, and a specific k-tuple of states $(s_1, \ldots, s_k)$ in the k-cartesian product of S can be represented as a conjunction of formulas over $V_1 \uplus \ldots \uplus V_k$. A terminal k-tuple of states is one in which all individual states are terminal, and a k execution is terminating if it ends at a terminal k-tuple of states. T is said to satisfy a k-safety property $(pre, post)$ if for every k terminating executions that start in states $s_1, \ldots, s_k$ such that $(s_1, \ldots, s_k) \models pre$, it holds that they end at states $t_1, \ldots, t_k$ such that $(t_1, \ldots, t_k) \models post$.

3.2 Abduction

Abductive inference [16] is a form of backward logical reasoning, to infer likely hypothesis from a given conclusion. Formally, given an invalid implication $\Gamma \Rightarrow \phi$, abductive inference finds a formula ψ such that $\Gamma \wedge \psi \Rightarrow \phi$ is valid, and $\Gamma \wedge \psi$ is satisfiable.

Note that ϕ is a trivial solution but it is not useful because it completely disregards our existing knowledge (of Γ). In this paper (as discussed later, Sect. 5.2), we rely on SyGuS and SMT solvers for doing abductive inference.

3.3 Interpolation

Consider an unsatisfiable set of clauses which have been partitioned into two sets, A and B. An interpolant [13] I for the pair (A, B) is a formula for which the following hold:

- $A \Rightarrow I$
- $I \wedge B$ is unsatisfiable
- I refers only to the common variables of A and B.

Such an interpolant, for first-order theories can be generated in linear time [33] from the resolution proof of unsatisfiability (of A and B).

3.4 Property Directed Self Composition

Property directed self composition, or PDSC, is a recent technique that combines the search of invariants with that of a composition. It does this by fixing a language of proofs, $\mathcal{L}_{\mathcal{P}}$, described by a given set of predicates, $\mathcal{P}$, and their boolean combinations, and navigating the space of all possible compositions to see if one of them has a proof in this language. Fixing the language not only makes the search tractable, it also allows PDSC to rely on the property that a transition system has an inductive invariant in $\mathcal{L}_{\mathcal{P}}$ if and only if its abstraction using $\mathcal{P}$ is safe. Therefore, by using predicate abstraction, it is possible with PDSC to either obtain an inductive invariant in $\mathcal{L}_{\mathcal{P}}$, or prove that none exists.

The way PDSC navigates through the composition space is by defining a composition function $f : S^k \to \mathbb{P}(\{1..k\})$, mapping each k-state to a non-empty set of copies that are to participate in the next step of the self composed program. This composition function is *semantic*, in that it does not necessarily depend on what are the syntactic constructs used in the next step of the component programs. This allows PDSC to explore beyond syntactic compositions, which may not always be possible or useful.

Given a composition function f, PDSC creates a composed transition relation $T^f = (S^k, R^f, F^k)$, where the set of states consists of all k-states, the terminal states are those in which all individual states are terminal, and R^f includes a transition from $(s_1, ..., s_k)$ to $(s'_1, ..., s'_k)$ *if and only if* $f(s_1, ..., s_k) = M$, and $(\forall i \in M.\ (s_i, s'_i) \in R) \wedge (\forall i \notin M.\ s_i = s'_i)$.

Intuitively, the composition function tells, for any state, what are the copies that are scheduled to move next, and the composed transition relation ensures that the components move as per their individual transition relation in the copies that are scheduled to move, and not move at all in any other copy.

Algorithm 1 Property Directed Self Composition

```
1: $\mathcal{F}_{block} \leftarrow \varnothing$
                              ▷ block compositions that cannot be proved safe
2: $f \leftarrow lockstep$
3: while $true$ do
4:     $\mathcal{T}^f = compose(f, T_1, \ldots, T_k)$
5:     $\mathcal{A}_{\mathcal{P}}^{\mathcal{T}^f} = abstract(\mathcal{T}^f, \mathcal{P})$
6:     $(res, inv, cex) = isBadReachable(\mathcal{A}_{\mathcal{P}}^{\mathcal{T}^f}, pre, post)$
                              ▷ where bad is negated post
7:     if $(res = safe)$ then
8:         return $(f, inv)$
9:     else
10:        $\mathcal{F}_{block} \leftarrow \mathcal{F}_{block} \cup \{f\}$      ▷ block $f$ to get rid of cex
11:        if (not all compositions are blocked) then
12:            $f \leftarrow pickUnblockedComposition(\mathcal{F}_{block})$
                              ▷ try a different, unblocked, composition
13:        else
14:            return (no proof in the language of $\mathcal{P}$)
```

Algorithm 1 presents an overview of how PDSC works. The composition is set to lockstep in the beginning, and the composed transition relation is obtained and abstracted with the given set of predicates. If the abstraction is found to be safe, at any stage, the algorithm returns a composition-invariant pair; otherwise, the composition is modified and the process is repeated. If none of the compositions succeed, the algorithm concludes that no invariant, which is a boolean combination of these given predicates, is inductive and safe. In other words, either the language is not rich enough to capture a safety proof for any of the compositions, or the program is not safe.

We have presented only a brief overview of the PDSC algorithm, with the aim of making this paper self-contained. We refer the interested readers to [39] for a detailed discussion.

3.5 Revisiting our Motivating Example

Let us recall the example shown in Fig. 1. We argued earlier that the loops in the two programs may be synchronized such that for every two iterations of the loop in the first one, we run only one iteration of that in the second one. This way of composing the loops of the two programs gives us a simpler loop invariant: $y_1 = 2{*}y_2$. The way PDSC arrives at this composition automatically is by fixing a proof language, and then by searching among the possible compositions allowed by the language.

Fig. 2 shows the composition and the proof obtained automatically by PDSC, for our motivating example. Intuitively, PDSC takes the input set of predicates (defining the proof language), and uses it to construct abstract states for every (consistent) combination of predicates and their negation. And then it explores transitions, labelled by the program copies whose next statement/block has to be executed, between these states to find a path to a final state where the property holds (e.g. the rightmost state in Fig. 2). Clearly, the search depends on the input set of predicates. For this example, PDSC expects four predicates from the user (without which it would not have been able to construct the three states shown in Fig. 2 and discharge the proof): $\texttt{z}_1\texttt{==2*z}_2$, $\texttt{y}_1\texttt{==2*y}_2$, $\texttt{z}_1\texttt{==2*z}_2\texttt{-1}$, and $\texttt{y}_1\texttt{==2*y}_2\texttt{+x}_2$. Note that the predicates $\texttt{y}_1\texttt{==y}_2$ and $\texttt{x}_1\texttt{==x}_2$ are available as the postcondition and precondition respectively, and thus need not be supplied externally.

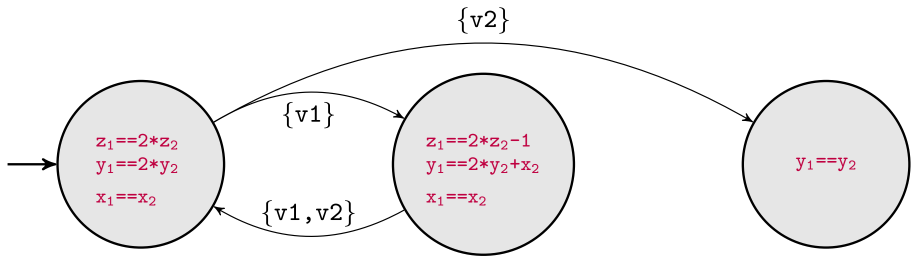

Fig. 2: Composition and proof obtained by PDSC, for the example in Fig. 1

4 Challenges and Contributions

We look at the important caveats of PDSC– *i*) it works for finding proofs but cannot detect real counterexamples, *ii*) it requires a set of predicates supplied externally, and *iii*) it cannot make progress if the supplied predicates are found to be insufficient to express a safe inductive invariant. The following key components of our algorithm helps us overcome these limitations.

1. *spuriousness checker*, to obtain a real counterexample trace if the programs do not satisfy the desired k-safety property (or, in our case, behave differently for the same input)
2. *predicate synthesizer*, to eliminate spurious counterexample traces and to enrich the language for finding safe invariants.

It is noteworthy that since PDSC works by checking safety of an abstraction of the composed transition relation, it may be possible to do the above by interfacing PDSC with a predicate-abstraction engine that can supply the predicates for refinement. However, interfacing with a black-box engine is not very useful because these predicates define the language of proofs, which in turn, decides the complexity of the composition-invariant search, and therefore it is important to have control on their quality and quantity to get a scalable solution.

We describe our algorithm formally in the next section, along with the details of the two components that we have added, and a proof that our algorithm works.

5 Algorithm

Algorithm 2 presents a pseudo-code of our approach, which enhances the original PDSC algorithm (shown in Algorithm 1) with the ability to synthesize predicates, to strengthen the language of proofs whenever necessary. In particular, the proposed enhancement is captured in lines 14-20, that *i*) returns the counterexample (*cex*) generated in line 7 if it is indeed a feasible trace by using a *spuriousness* check (lines 14 and 15), *ii*) adds a predicate to refine the counterexample if spurious (lines 18 and 18), and *iii*) resets the composition space and restarts the search from the lockstep composition (lines 19 and 20).

The next two subsections describe the two procedures – *isSpurious* and *synthesizePredicates* – mentioned in the algorithm.

5.1 Spuriousness Check

The counterexample obtained in line 7 of Algorithm 2 is essentially a sequence of abstract states that end in a bad state, i.e., a state that violates the *post*. Each abstract state is defined by a valuation of all the predicates in $\mathcal{P}$. Let us denote this sequence of abstract states as: $A_0, A_1, \ldots, A_{bad}$. These states, in the counterexample trace, are connected by a transition relation which is defined by the transition relations of the component programs, and the *current* composition function. We start by taking a concrete initial state c_0, a model of A_0, and then applying the transitions of the trace on c_0 step by step. After taking the i^{th} step, it is checked that the concrete target state arrived at, let us call it c_i, is actually a model of the corresponding abstract state A_i. If this indeed holds all the way

Algorithm 2 Pdsc with Predicate Synthesis

```
1: $\mathcal{F}_{block} \leftarrow \varnothing$
                                        ▷ block compositions that cannot be proved safe
2: $f \leftarrow lockstep$
3: while $true$ do
4:     $\mathcal{T}^f = compose(f, T_1, \ldots, T_k)$
5:     $\mathcal{A}_{\mathcal{P}}^{\mathcal{T}^f} = abstract(\mathcal{T}^f, \mathcal{P})$
6:     $(res, inv, cex) = isBadReachable(\mathcal{A}_{\mathcal{P}}^{\mathcal{T}^f}, pre, post)$
                                        ▷ where $bad$ is negated $post$
7:     if $(res = safe)$ then
8:         return $(f, inv)$
9:     else
10:        $\mathcal{F}_{block} \leftarrow \mathcal{F}_{block} \cup \{f\}$        ▷ block $f$ to get rid of $cex$
11:        if (not all compositions are blocked) then
12:            $f \leftarrow pickUnblockedComposition(\mathcal{F}_{block})$
                                        ▷ try a different, unblocked, composition
13:        else
14:            if $(isSpurious(cex)$ is $false)$ then
15:                return $(unsafe, cex)$
16:            else                                        ▷ cex is spurious
17:                $\mathcal{P}' \leftarrow synthesizePredicates(cex)$
                                        ▷ new predicates that eliminate $cex$
18:                $\mathcal{P} = \mathcal{P} \cup \mathcal{P}'$        ▷ strengthen the language of proofs
19:                $\mathcal{F}_{block} \leftarrow \varnothing$        ▷ unblock the blocked compositions
20:                $f \leftarrow lockstep$        ▷ restart, with lockstep composition
```

up to A_{bad}, then we have an actual counterexample trace. Otherwise, there must be a transition $\langle A_i, \mathcal{T}^f, A_{i+1}\rangle$ in the abstract trace that could not be taken by the concrete state c_i (i.e., $c_{i+1} \wedge A_{i+1}$ was unsat), where c_{i+1} is the concrete state reached after taking $\mathcal{T}^f$ from c_i, and $\mathcal{T}^f$ is the composed transition relation as per the current composition function f.

Intuitively, this means that it is not possible to go from a part of A_i (that part exists, because we know that c_i belongs to it) to A_{i+1} along the composed transition relation. Therefore, in order to refine this spuriousness, it is necessary to add a predicate that identifies the part. The goal of the *synthesizePredicates* procedure is to find such a predicate.

5.2 Synthesizing Predicates from Counterexamples

We illustrate how a spurious transition of the form $\langle A_{src}, \mathcal{T}^f, A_{tgt}\rangle$, where $\mathcal{T}^f$ is the composed transition relation and A_{src} and A_{tgt} are the source and target states, is blocked by doing a counterexample-guided abstraction refinement. The problem of blocking a spurious transition is essentially a problem of logical abduction [17], which works towards finding an explanatory hypothesis for a desired outcome. The desired outcome here is that A_{tgt} should not be reachable along $\mathcal{T}^f$ from A_{src}, but currently it is. In other words,

$$A_{src}(V) \wedge \mathcal{T}^f(V, V') \not\Rightarrow \neg A_{tgt}(V')$$

Therefore, we need to find a hypothesis, $p(X)$, possibly over a subset of variables, i.e. $X \subseteq V$, such that

$$p(X \subseteq V) \wedge A_{src}(V) \wedge \mathcal{T}^f(V, V') \Rightarrow \neg A_{tgt}(V')$$

But, at the same time, it is important to discard *trivial* solutions – one that uses the consequent itself as p, and another that makes the antecedent *false*. Therefore, the abducer looks for a minimal (logically weakest) solution under the condition that

$$p(X \subseteq V) \wedge A_{src}(V) \wedge \mathcal{T}^f(V, V') \not\Rightarrow \bot$$

We use the following two ways to solve for p.

Using a SyGuS solver We encode these constraints directly into the SyGuS input language [34], and use CVC4 [6] (version 1.8) to obtain a solution. SyGuS allows an enumerative search strategy that leads to smaller predicates.

Quantifier Elimination using Z3 A solution to the abductive inference problem is given by

$$\forall\ ((V \cup V') \setminus X).\ A_{src}(V) \wedge \mathcal{T}^f(V, V') \Rightarrow \neg A_{tgt}(V')$$

or, equivalently, by the negation of

$$\exists\ ((V \cup V') \setminus X).\ A_{src}(V) \wedge \mathcal{T}^f(V, V') \wedge A_{tgt}(V')$$

We obtain a solution by quantifier elimination using Z3 [14]. Since we are looking at the problem of predicate refinement, it is not necessary to negate the solution (negating a predicate does not affect the expressiveness of our proof language in any way).

Given an input program and a safety property specified as $(pre, post)$, and an initial proof language – defined by predicates that are needed to specify *pre* and *post* and their boolean combinations – Algorithm 2 either terminates with a proof, or a counterexample, or goes on enriching the language of proofs, in each iteration making it provably more expressive than the earlier one. The algorithm is not guaranteed to terminate, since the problem of finding the composition-invariant pair is undecidable in general [39]. It can, in principle, terminate for finite state systems, although with exponential complexity, since the set of possible composition-invariant pairs is itself finite for finite state systems. Note the number of predicates is also finite in a finite-state systems.

Theorem 1. *The proposed refinement ensures progress, i.e., the predicate added in every step, which is the solution of the abduction query, gets rid of the spurious counterexample, and strictly strengthens the proof language.*

Proof. It is easy to see that the refinement indeed removes the spurious counterexample, because the queries given to SyGuS solver and to Z3 exactly encode this constraint that the target should not be reachable from the source. We also know that a solution certainly exists, because the concrete state corresponding to the A_{src} is itself a non-trivial solution.

To argue that the newly added predicate p_n (say) strictly enriches the proof language given by $\mathcal{P} = \{p_1, \ldots, p_{n-1}\}$, let us assume, on the contrary, that it does not. In that case, p_n can be written as a boolean combination of the predicates in $\mathcal{P}$. Since A_{src} is also a boolean combination of predicates in $\mathcal{P}$, they (A_{src} and p_n) must either agree on all predicates in $\mathcal{P}$ or disagree on at least one of them. The latter is not possible since this disagreement would mean that abduction problem was solved trivially by falsifying the antecedent, however we know that it is not true because trivial solutions are avoided. On the other hand, if they agree on all the predicates $\{p_1, \ldots, p_{n-1}\}$ then $A_{src} \wedge p_n$ is simply A_{src} and the spuriousness could not have been removed. Hence, a contradiction.

The soundness of our algorithm follows directly from the soundness of PDSC.

5.3 Obtaining Predicates from Infeasibility Proofs

As described in Sect. 5.1, an abstract counterexample trace is a sequence of transitions that begin at the initial abstract state and end at a bad state. Note that the transitions in such a trace are labelled by program statements from the two (or more) program copies/components. To check whether an abstract trace is feasible, we can collect the program statements from the transitions of the entire trace, and give it to a solver to check for satisfiability. This gives us an alternate, more general way to check if the abstract counterexample was spurious, independent of any concrete initial state.

If the solver returns *sat*, we get an actual counterexample trace as a model, which demonstrates that the desired property fails to hold. However, if the solver returns *unsat*, then the sequence interpolants [28] obtained from the infeasibility proof (of the concrete trace) may be used as additional predicates to strengthen the proof language. Although at this point it would be sound to add all the interpolants as predicates to strengthen the proof language, it must be noted that the search of a proof gets more and more difficult as the proof language gets richer (because the number of abstract states is exponential in the number of predicates, and PDSC searches through the states to find a composition-invariant pair). Therefore, there is a downside to making the proof language needlessly expressive – the search of a composition-invariant pair will become considerably harder in every iteration. A practical solution, naturally, is to add only a few interpolants or even sub-expressions from what the prover gives us. In particular, our implementation:

– is parametrized to add at most p predicates each time (in our experiments, we use $p = 2$)
– prioritizes expressions that relate variables that have not been related so far in the existing set of predicates
– prioritizes adding shorter and logically stronger expressions.

Note that this approach is still sound, though we cannot guarantee now that the newly added predicates necessarily remove the spurious counterexample. Adding all the interpolants obtained from the infeasibility proof would certainly eliminate the counterexample, but it will make the algorithm quite inefficient. Our experiments support that the compromise of adding only a few interpolants, or sub-expressions derived from them, is indeed very useful in practice.

6 Implementation and Experiments

6.1 Implementation

We have implemented our approach in a tool[5], PdscSynth, which is built on the Pdsc tool[6]. Like Pdsc, our input is a transition system encoded by Constrained Horn Clauses (CHC) in SMT2 format, a correctness (k-safety) property, and a set of predicates that specify the *pre* and *post* conditions. While Pdsc expects an additional set of predicates (that may be mined automatically from the program syntax, or supplied manually), PdscSynth gets them automatically, lazily on demand, by doing:

1. Syntax-Guided Synthesis, using CVC4 [6], version 1.8
2. Quantifier Elimination, using Z3 [14], version 4.8.9, *and*
3. Craig Interpolation, using MathSAT5 [12], version 5.6.6.

In CVC4, we use restrict ourselves to Linear Integer Arithmetic (which is what our benchmarks are also restricted to), and use the default grammar that CVC provides for LIA. The variables and the constants for the grammar come from the program. Quantifier elimination is performed using the recursive QSAT technique [8], available in Z3 tool.

6.2 Benchmarks

An interesting use-case of 2-safety verification is automated evaluation of programming assignments which may be done by checking equivalence between a submitted program and a reference solution. With this in mind, we have used 9 programming assignments samples in our experiments, derived from [26]. In addition, our benchmarks consists of 7 examples derived from [39], and 3 crafted examples. Each benchmark consists of two component programs (that may be copies, or syntactically/semantically different programs), and the correctness

[5] Artifacts available at: `https://github.com/Akshatha-Shenoy/PdscSynth`
[6] `https://bitbucket.org/sharonsh/pdsc/src/master/`

property is stated as a set of pre- and post-conditions. Intuitively, we check the equivalence property for all the benchmarks, except in case of `squareSum` where the components compute the sum of squares of integers in a given interval, and the property is that a bigger interval leads to a bigger sum.

We call a benchmark *safe* if the composed programs in the benchmark satisfy the given correctness property, and *unsafe* otherwise. Since the sample programming assignment solutions include correct as well as incorrect solutions, we have 6 unsafe benchmarks and 3 safe ones. The benchmarks derived from the Pdsc paper are all safe, and we got them by deleting *all* the manually supplied predicates (excluding those predicates that are necessary to specify the pre- and post-condition). For the `doubleSquare` benchmark, we also created instances of varying difficulty by retaining some of the manually supplied predicates. The crafted benchmarks were obtained from two different programs: one that sums all numbers from 1 to n (a safe and an unsafe version), and another one that increments two equal numbers by different values and then decrements them by the same value to get equal numbers in the end again (safe version).

6.3 Results

We ran PdscSynth on all the 19 benchmarks described above. Table 1 shows the results of our experiments. The columns SyGuS, QE, and Interpolation contain, except in case of timeouts, two comma-separated entries – the number of predicates synthesized on the left, and the time taken to produce a proof (or a counterexample) on the right. The letters 'm' and 's' denote minutes and seconds, respectively. The experiments were run on an Intel i5 machine running at 1.70 GHz, with 16 GB of RAM. The 'timeout' indicates that the technique could not decide the benchmark within 10 minutes.

It is noteworthy that interpolation was able to produce the desired predicates in *every* case. The time taken and the predicates added by the interpolation technique confirm that the technique was effective (in its selection of predicates to add, so that the proof language becomes richer but not needlessly expressive). In several unsafe benchmarks (`fig4_1`, `fig4_2`, `subsume_1`, `subsume_2` and `puzzle_1`), interpolation needed fewer predicates in comparison to QE and SyGuS. This is because with SyGuS and QE, we check the spuriousness of one concretization of the abstract trace at a time, unlike in case of interpolation where all possible concretizations are checked at once. Thus, interpolation adds a predicate only when none of the concretizations of an abstract trace is feasible. Whereas, QE and SyGuS may add a predicate needlessly even though the current abstract trace has a feasible concretization (which hasn't been looked at yet).

The quantifier elimination with Z3 was also able to get the necessary and sufficient predicates in a larger number of cases (in particular, where SyGuS could not scale). For the examples where SyGuS also worked, it almost always got smaller predicates than QE, though not necessarily fewer in numbers (for `sum_pc`, SyGuS had to synthesize four more predicates as compared to quantifier elimination, whereas for `inc_dec` SyGuS managed with two predicates lesser).

The quantifier elimination with Z3 performed quite well in comparison to interpolation as well, solving all the benchmarks except `halfSquare` and most of `doubleSquare` variants. However, the proofs generated with QE were often quite big, as the technique obtained predicates that were much bigger in size compared to SyGuS and Interpolation.

Table 1: No. of predicates synthesized, and the time taken by SyGuS (Syntax-Guided Synthesis, using CVC4-1.8), QE (Quantifier Elimination, using Z3 4.8.9), and Interpolation (using MathSAT5 5.6.6), and a comparison with LLRÊVE on our benchmarks

S. No.	Benchmark	Source	Safe/Unsafe	SyGuS (#pred, time)	QE (#preds, time)	Interpolation (#preds, time)	LLRÊVE
1.	sum_to_n	crafted	safe	timeout	8, 1m32s	3, 17.36s	0.056s
2.	sum_to_n_err	crafted	unsafe	0, 0.34s	0, 0.77s	0, 1.43s	0.069s
3.	inc_dec	crafted	safe	2, 9.15s	4, 20.95s	3, 13.95s	unknown
4.	squareSum	cav19	safe	0, 0.66s	0, 1.14s	0, 1.67s	–
5.	sum_pc	cav19	safe	5, 1m32s	1, 13.58s	4, 1m28s	unknown
6.	fig4_1	icse16	unsafe	1, 3.65s	2, 7.16s	0, 2.26s	0.038s
7.	fig4_2	icse16	unsafe	1, 3.86s	2, 7.24s	0, 2.27s	0.067s
8.	fig4_ref_ref	icse16	safe	0, 0.11s	0, 0.58s	0, 0.96s	0.044s
9.	subsume_1	icse16	unsafe	timeout	3, 7.88s	0, 2.20s	0.041s
10.	subsume_2	icse16	unsafe	timeout	2, 5.22s	0, 2.24s	0.061s
11.	subsume_ref_ref	icse16	safe	timeout	1, 3.9s	8, 24.48s	0.051s
12.	puzzle_1	derived from icse16	unsafe	timeout	4, 26.8s	2, 9.04s	0.040s
13.	puzzle_2	derived from icse16	unsafe	timeout	8, 2m31s	8, 4m7s	0.029s
14.	puzzle_ref_ref	derived from icse16	safe	timeout	2, 10.33s	1, 5.73s	0.061s
15.	halfSquare	cav19	safe	timeout	timeout	3, 6m9s	unknown
16.	doubleSquare_1	derived from cav19	safe	timeout	timeout	11, 3m42s	timeout
17.	doubleSquare_2	derived from cav19	safe	timeout	timeout	10, 3m14s	timeout
18.	doubleSquare_3	derived from cav19	safe	timeout	timeout	7, 1m50s	timeout
19.	doubleSquare_4	derived from cav19	safe	4, 1m19s	9, 3m26s	1, 17.12s	timeout

6.4 Performance on our Motivating Example

Let us recall our motivating example once again. As described in Sect. 3.5, Pdsc was able to construct a proof with the help of four user-supplied predicates: $z_1==2*z_2$, $y_1==2*y_2$, $z_1==2*z_2-1$, and $y_1==2*y_2+x_2$. We created four variants of this benchmark: `doubleSquare_1` (where none of these four were supplied), `doubleSquare_2` (where only $z_1==2*z_2$ was supplied), `doubleSquare_3` (where $z_1==2*z_2$ and $y_1==2*y_2$ were supplied), and `doubleSquare_4` (where $y_1==2*y_2$ was removed, the other three were supplied). As shown in the results table, PdscSynth was able to solve all the four benchmarks using Interpolation, whereas none of the other techniques could work even for the simpler variants (except `doubleSquare_4` which could be solved but needed more predicates and a lot more time with SyGuS and QE).

6.5 Comparison with LLRÊVE

We also compared PdscSynth with LLRÊVE[7] , an automated regression verification tool, as it can automatically check programs for equivalence [23]. Table 1

[7] `https://formal.kastel.kit.edu/projects/improve/reve/`

shows the results of this comparison; we used both Z3 v4.8.9 and Eldarica v2.0.8 as the backend solver, and have reported the better of the two results. While LLRÊVE could solve all the unsafe benchmarks fairly quickly, the only safe benchmarks that it could solve were the ones for which the components were exactly the same. There were four such benchmarks in our experiments: `sum_to_n`, and the `*_ref_ref` benchmarks where reference implementations for programming assignments were compared to themselves. For all other safe benchmarks, LLRÊVE could not decide that they were indeed safe, even though we let it run beyond the timeout for about 30 minutes. Also, note that we could not run LLRÊVE on the `squareSum` benchmark because the safety property there is not an equivalence check.

6.6 Reducing Predicate Size for Quantifier Elimination

In order to discover smaller predicates as solutions, we implemented a strategy for Z3 to eliminate as many variables as possible, and return a solution in the smallest set of variables possible, under the broad assumption that predicates in fewer variables would also be smaller. This is not always true; in fact, we realized that it is better to eliminate all but two variables to begin with, and come to eliminating all but one variable only in the end. In general, while this strategy helps in reducing the size of predicates, such strategies can impact the performance adversely. Striking a good balance between scalability and usefulness of the predicates, therefore, is crucial, and makes for an important direction of future work.

7 Related Work

The novelty of our work lies in giving a completely automatic approach for doing property directed self composition [39] to address the problem of k-safety verification. While the user-dependence has been described here as a problem only for PDSC, related techniques like [9] are also dependent on predicates that may be used to align the component programs. In particular, [9] uses an alignment predicate to construct a program alignment automaton that semantically aligns the programs between which equivalence is to be checked, quite like how PDSC composes the component programs. The predicates play an important role in these techniques, and therefore it is crucial to have techniques that can generate useful predicates completely automatically.

Since self-composition poses the same challenges for proving equivalence of programs as it does for 2-safety verification, an automated property directed self composition technique can be helpful in a number of applications of program equivalence. This includes evaluation of programming assignment w.r.t. a given correct implementation [3], semantic alignment [10], translation validation [40,25], design and verification of compiler optimizations [41,30], and program synthesis and superoptimization [5,38], among several others. We reiterate that it is the combined strength of PDSC and the automation that makes this approach usable and effective in practice.

Our method relies on different techniques for synthesizing predicates: SyGuS [2], Abductive inference [16], and Interpolation [27]. These techniques are certainly related in the way they can address a common problem, which in our case is the strengthening of the proof language. They have also been used together, sometimes in conjunction with other techniques, to address related problems like inferring inductive invariants [19,22] and maximal specifications [1,36]. The commonality of the techniques, which makes them suitable for these problems, is their ability to generalize (from examples or counterexamples). Whereas, they differ in how they perform the generalization, and thus have different strengths as confirmed by our experiments.

8 Conclusion and Future Work

This paper proposes an algorithm that builds on top of a property directed self composition technique for hypersafety verification, and overcomes some of its important caveats. PDSC expects users to supply a *proof language* in which it searches for an easy-to-prove composition. Our algorithm gets rid of the user-dependence that PDSC has, and makes it capable to do refutations as well. We have implemented, and experimented with three different techniques relying on SyGuS, Quantifier Elimination, and Interpolation, that can construct and enrich the proof language as and when required for a given program and a property. Our experiments demonstrate that the proposed techniques are effective as well as efficient.

Looking ahead, we see several interesting directions of future work. For one, since the space of compositions is navigated repeatedly, it may be useful to identify good and bad regions of the composition space each time, and use it in subsequent iterations to scale better. However, this is challenging as the composition space changes every time the proof language is strengthened. For certain applications, e.g. while proving equivalence of programs, it may be desirable to obtain proofs that are shorter and thus easier to understand. Therefore, the task of finding smaller, and few but useful predicates is an important one, and makes for an interesting future work. It would also be worthwhile to enhance our technique to handle programs with arrays and other data-structures so that we can look at a wider set of benchmarks.

References

1. A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 789–801, New York, NY, USA, 2016. Association for Computing Machinery.
2. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In

Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pages 1–8, 2013.

3. J. K. Anil, S. Prabhu, K. Madhukar, and R. Venkatesh. Using hypersafety verification for proving correctness of programming assignments. In G. Rothermel and D. Bae, editors, *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, pages 81–84. ACM, 2020.
4. S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 394–403, New York, NY, USA, 2006. Association for Computing Machinery.
5. S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. *SIGARCH Comput. Archit. News*, 34(5):394–403, Oct. 2006.
6. C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
7. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, page 100, USA, 2004. IEEE Computer Society.
8. N. Bjørner and M. Janota. Playing with quantified satisfaction. *LPAR (short papers)*, 35:15–27, 2015.
9. B. Churchill, O. Padon, R. Sharma, and A. Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery.
10. B. Churchill, O. Padon, R. Sharma, and A. Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery.
11. B. Churchill, R. Sharma, J. Bastien, and A. Aiken. Sound loop superoptimization for google native client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 313–326, New York, NY, USA, 2017. Association for Computing Machinery.
12. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. *The MathSAT5 SMT Solver*, pages 93–107. Springer Berlin Heidelberg, 2013.
13. W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, 1957.
14. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.
15. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
16. I. Dillig. Abductive inference and its applications in program analysis, verification, and synthesis. In R. Kaivola and T. Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, page 4. IEEE, 2015.

17. I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 181–192, New York, NY, USA, 2012. Association for Computing Machinery.
18. A. Farzan and A. Vandikas. Automated hypersafety verification. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification*, pages 200–218, Cham, 2019. Springer International Publishing.
19. G. Fedyukovich and R. Bodík. Accelerating Syntax-Guided Invariant Synthesis. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 251–269. Springer, 2018.
20. G. Fedyukovich, S. J. Kaufman, and R. Bodík. Sampling invariants from frequency distributions. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 100–107, 2017.
21. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Solving constrained horn clauses using syntax and data. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9, 2018.
22. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Quantified invariants via syntax-guided synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 259–277, 2019.
23. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, ASE '14, pages 349–360. ACM, Sept. 2014.
24. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53 – 71, 2005. Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004).
25. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6):327–337, June 2009.
26. S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann. Measuring code behavioral similarity for programming and software engineering education. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 501–510, New York, NY, USA, 2016. ACM.
27. K. L. McMillan. Interpolation and sat-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
28. K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, pages 123–136, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
29. G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 83–94, New York, NY, USA, 2000. Association for Computing Machinery.
30. G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000.
31. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

32. S. Prabhu, K. Madhukar, and R. Venkatesh. Efficiently learning safety proofs from appearance as well as behaviours. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, pages 326–343, 2018.
33. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.
34. M. Raghothaman and A. Udupa. Language to Specify Syntax-Guided Synthesis Problems. *CoRR*, abs/1405.5590, 2014.
35. D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 669–685, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
36. S. P. S, G. Fedyukovich, K. Madhukar, and D. D'Souza. Specification synthesis with constrained horn clauses. In S. N. Freund and E. Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1203–1217. ACM, 2021.
37. E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 305–316, New York, NY, USA, 2013. Association for Computing Machinery.
38. E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGARCH Comput. Archit. News*, 41(1):305–316, Mar. 2013.
39. R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel. Property directed self composition. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification*, pages 161–179, Cham, 2019. Springer International Publishing.
40. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM.
41. J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for llvm. *SIGPLAN Not.*, 46(6):295–305, June 2011.