Property Directed Self Composition

Ron Shemer[1(✉)], Arie Gurfinkel[2], Sharon Shoham[1], and Yakir Vizel[3]

[1] Tel Aviv University, Tel Aviv, Israel
ronsheme@mail.tau.ac.il
[2] University of Waterloo, Waterloo, Canada
[3] The Technion, Haifa, Israel

Abstract. We address the problem of verifying *k-safety properties*: properties that refer to k interacting executions of a program. A prominent way to verify k-safety properties is by *self composition*. In this approach, the problem of checking k-safety over the original program is reduced to checking an "ordinary" safety property over a program that executes k copies of the original program in some order. The way in which the copies are composed determines how complicated it is to verify the composed program. We view this composition as provided by a *semantic self composition function* that maps each state of the composed program to the copies that make a move. Since the "quality" of a self composition function is measured by the ability to verify the safety of the composed program, we formulate the problem of inferring a self composition function together with the inductive invariant needed to verify safety of the composed program, where both are restricted to a given language. We develop a *property-directed* inference algorithm that, given a set of predicates, infers composition-invariant pairs expressed by Boolean combinations of the given predicates, or determines that no such pair exists. We implemented our algorithm and demonstrate that it is able to find self compositions that are beyond reach of existing tools.

1 Introduction

Many relational properties, such as noninterference [12], determinism [21], service level agreements [9], and more, can be reduced to the problem of k-safety. Namely, reasoning about k different traces of a program simultaneously. A common approach to verifying k-safety properties is by means of *self composition*, where the program is composed with k copies of itself [4,32]. A state of the composed program consists of the states of each copy, and a trace naturally corresponds to k traces of the original program. Therefore, k-safety properties of the original program become ordinary safety properties of the composition, hence reducing k-safety verification to ordinary safety. This enables reasoning about k-safety properties using any of the existing techniques for safety verification such as Hoare logic [20] or model checking [7].

While self composition is sound and complete for k-safety, its applicability is questionable for two main reasons: (i) considering several copies of the program greatly increases the state space; and (ii) the way in which the different copies are composed when reducing the problem to safety verification affects the complexity of the resulting self composed program, and as such affects the complexity of verifying it. Improving the applicability of self composition has been the topic of many

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 161–179, 2019.
https://doi.org/10.1007/978-3-030-25540-4_9

works [2,14,18,26,30,33]. However, most efforts are focused on compositions that are pre-defined, or only depend on syntactic similarities.

In this paper, we take a different approach; we build upon the observation that by choosing the "right" composition, the verification can be greatly simplified by leveraging "simple" correlations between the executions. To that end, we propose an algorithm, called PDSC, for inferring a *property directed* self composition. Our approach uses a *dynamic* composition, where the composition of the different copies can change during verification, directed at simplifying the verification of the composed program.

Compositions considered in previous work differ in the order in which the copies of the program execute: either synchronously, asynchronously, or in some mix of the two [3,14,34]. To allow general compositions, we define a *composition function* that maps every state of the composed program to the set of copies that are scheduled in the next step. This determines the order of execution for the different copies, and thus induces the self composed program. Unlike most previous works where the composition is pre-defined based on syntactic rules only, our composition is *semantic* as it is defined over the state of the composed program.

To capture the difficulty of verifying the composed program, we consider verification by means of inferring an inductive invariant, parameterized by a language for expressing the inductive invariant. Intuitively, the more expressive the language needs to be, the more difficult the verification task is. We then define the problem of inferring a composition function *together* with an inductive invariant for verifying the safety of the composed program, where both are restricted to a given language. Note that for a fixed language $\mathcal{L}$, an inductive invariant may exist for some composition function but not for another[1]. Thus, the restriction to $\mathcal{L}$ defines a target for the inference algorithm, which is now directed at finding a composition that admits an inductive invariant in $\mathcal{L}$.

Example 1. To demonstrate our approach, consider the program in Fig. 1. The program inserts a new value into an array. We assume that the array A and its length len are "low"-security variables, while the inserted value h is "high"-security. The first loop finds the location in which h will be inserted. Note that the number of iterations depends on the value of h. Due to that, the second loop executes to ensure that the output i (which corresponds to the number of iterations) does not leak sensitive data. As an example, we emphasize that without the second loop, i could leak the location of h in A. To express the property that i does not leak sensitive data, we use the 2-safety property that in any two executions, if the inputs A and len are the same, so is the output i.

To verify the 2-safety property, consider two copies of the program. Let the language $\mathcal{L}$ for verifying the self composition be defined by the predicates depicted in Fig. 1. The most natural self composition to consider is a lock-step composition, where the copies execute synchronously. However, for such a composition the composed program may reach a state where, for example, $i_1 = i_2 + 1$. This occurs when the first copy exists the first loop, while the second copy is still executing it. Since the language cannot express this correlation between the two copies, no inductive invariant suffices to verify that $i_1 = i_2$ when the program terminates.

[1] See the extended version [29] for an example that requires a non-linear inductive invariant with a composition that is based on the control structure but has a linear invariant with another.

```
int arrayInsert(int[] A, int len, int h) {
     int i=0;
  1: while (i < len && A[i] < h)
       i++;
  2: len = shift_array(A, i, 1);
     A[i] = h;
  3: while (i < len)
       i++;
  4: return i;
  }

predicates: i1 = i2, i1 < len1, i2 < len2,
     A1[i1] < h1, A2[i2] < h2, len1 = len2,
     len1 = len2 + 1, len2 = len1 + 1
```

```
composition:
if(pc1 < 3 && (pc2 > 0 || !cond1)
   && (pc2 == 3||(pc2 == 0 && cond2)))
      step(1);
else if (pc2 < 3 && (pc1 > 0 || !cond2)
         && (pc1 == 3 || (pc1 == 0 && cond1)))
      step(2);
else step(1,2);

      cond1 := i1 < len1 && A1[i1] < h1
      cond2 := i2 < len2 && A2[i2] < h2
```

Fig. 1. Constant-time insert to an array.

In contrast, when verifying the 2-safety property, PDSC directs its search towards a composition function for which an inductive invariant in $\mathcal{L}$ does exist. As such, it infers the composition function depicted in Fig. 1, as well as an inductive invariant in $\mathcal{L}$. The invariant for this composition implies that $i_1 = i_2$ at every state.

As demonstrated by the example, PDSC focuses on logical languages based on predicate abstraction [17], where inductive invariants can be inferred by model checking. In order to infer a composition function that admits an inductive invariant in $\mathcal{L}$, PDSC starts from a default composition function, and modifies its definition based on the reasoning performed by the model checker during verification. As the composition function is part of the verified model (recall that it is defined over the program state), different compositions are part of the state space explored by the model checker. As a result, a key ingredient of PDSC is identifying "bad" compositions that prevent it from finding an inductive invariant in $\mathcal{L}$. It is important to note that a naive algorithm that tries all possible composition functions has a time complexity $O(2^{2^{|\mathcal{P}|}})$, where $\mathcal{P}$ is the set of predicates considered. However, integrating the search for a composition function into the model checking algorithm allows us to reduce the time complexity of the algorithm to $2^{O(|\mathcal{P}|)}$, where we show that the problem is in fact PSPACE-hard.[2]

We implemented PDSC using SEAHORN [19], Z3 [25] and SPACER [22] and evaluated it on examples that demonstrate the need for nontrivial semantic compositions. Our results clearly show that PDSC can solve complex examples by inferring the required composition, while other tools cannot verify these examples. We emphasize that for these particular examples, lock-step composition is not sufficient. We also evaluated PDSC on the examples from [26,30] that are proven with the trivial lock-step composition. On these examples, PDSC is comparable to state of the art tools.

Related Work. This paper addresses the problem of verifying k-safety properties (also called hyperproperties [8]) by means of self composition. Other approaches tackle the problem without self-composition, and often focus on more specific properties, most noticeably the 2-safety noninterference property (e.g. [1,33]). Below we focus on works that use self-composition.

[2] Proofs of the claims made in this paper can be found in the extended version [29].

Previous work such as [2–4,14,15,32] considered self composition (also called product programs) where the composition function is constant and set a-priori, using syntax-based hints. While useful in general, such self compositions may sometimes result in programs that are too complex to verify. This is in contrast to our approach, where the composition function is evolving during verification, and is adapted to the capabilities of the model checker.

The work most closely related to ours is [30] which introduces Cartesian Hoare Logic (CHL) for verification of k-safety properties, and designs a verification framework for this logic. This work is further improved in [26]. These works search for a proof in CHL, and in doing so, implicitly modify the composition. Our work infers the composition explicitly and can use off-the-shelf model checking tools. More importantly, when loops are involved both [30] and [26] use lock-step composition and align loops syntactically. Our algorithm, in contrast, does not rely on syntactic similarities, and can handle loops that cannot be aligned trivially.

There have been several results in the context of harnessing Constraint Horn Clauses (CHC) solvers for verification of relational properties [11,24]. Given several copies of a CHC system, a product CHC system that synchronizes the different copies is created by a syntactical analysis of the rules in the CHC system. These works restrict the synchronization points to CHC predicates (i.e., program locations), and consider only one synchronization (obtained via transformations of the system of CHCs). On the other hand, our algorithm iteratively searches for a good synchronization (composition), and considers synchronizations that depend on program state.

Equivalence Checking and Regression Verification. Equivalence checking is another closely related research field, where a composition of several programs is considered. As an example, equivalence checking is applied to verify the correctness of compiler optimizations [10,18,28,34]. In [28] the composition is determined by a brute-force search for possible synchronization points. While this brute-force search resembles our approach for finding the correct composition, it is not guided by the verification process. The works in [10,18] identify possible synchronization points syntactically, and try to match them during the construction of a simulation relation between programs.

Regression verification also requires the ability to show equivalence between different versions of a program [15,16,31]. The problem of synchronizing unbalanced loops appears in [31] in the form of unbalanced recursive function calls. To allow synchronization in such cases, the user can specify different unrolling parameters for the different copies. In contrast, our approach relies only on user supplied predicates that are needed to establish correctness, while synchronization is handled automatically.

2 Preliminaries

In this paper we reason about programs by means of the transition systems defining their semantics. 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 the steps in an execution of the program, and $F \subseteq S$ is a set of *terminal states* $F \subseteq S$ such that every terminal state $s \in F$ has an outgoing transition to itself and no additional transitions (terminal states allow us to

reason about pre/post specifications of programs). An *execution* or *trace* $\pi = s_0, s_1, \ldots$ is a (finite or infinite) sequence of states such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. The execution is *terminating* if there exists $0 \leq i \leq |\pi|$ such that $s_i \in F$. In this case, the suffix of the execution is of the form $s_i, s_i, \ldots$ and we say that π ends at s_i.

As usual, we represent transition systems using logical formulas over a set of variables, corresponding to the program variables. We denote the set of variables by $\mathcal{V}$. The set of terminal states is represented by a formula over $\mathcal{V}$ and the transition relation is represented by a formula over $\mathcal{V} \uplus \mathcal{V}'$, where $\mathcal{V}$ represents the pre-state of a transition and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ represents its post-state. In the sequel, we use sets of states and their symbolic representation via formulas interchangeably.

Safety and Inductive Invariants. We consider safety properties defined via pre/post conditions.[3] A *safety property* is a pair $(pre, post)$ where $pre, post$ are formulas over $\mathcal{V}$, representing subsets of S, denoting the pre- and post-condition, respectively. T *satisfies* $(pre, post)$, denoted $T \models (pre, post)$, if every terminating execution π of T that starts in a state s_0 such that $s_0 \models pre$ ends in a state s such that $s \models post$. In other words, for every state s that is reachable in T from a state in *pre* we have that $s \models F \rightarrow post$.

A prominent way to verify safety properties is by finding an inductive invariant. An *inductive invariant* for a transition system T and a safety property $(pre, post)$ is a formula Inv such that(1) $pre \Rightarrow Inv$ (initiation), (2) $Inv \wedge R \Rightarrow Inv'$ (consecution), and (3) $Inv \Rightarrow (F \rightarrow post)$ (safety), where $\varphi \Rightarrow \psi$ denotes the validity of $\varphi \rightarrow \psi$, and φ' denotes $\varphi(\mathcal{V}')$, i.e., the formula obtained after substituting every $v \in \mathcal{V}$ by the corresponding $v' \in \mathcal{V}$. If there exists such an inductive invariant, then $T \models (pre, post)$.

k-safety. A *k-safety property* refers to k interacting executions of T. Similarly to an ordinary property, it is defined by $(pre, post)$, except that *pre* and *post* are defined over $\mathcal{V}^1 \uplus \ldots \uplus \mathcal{V}^k$ where $\mathcal{V}^i = \{v^i \mid v \in \mathcal{V}\}$ denotes the ith copy of the program variables. As such, *pre* and *post* represent sets of k-tuples of program states (*k-states* for short): for a k-tuple $(s_1, \ldots, s_k)$ of states and a formula φ over $\mathcal{V}^1 \uplus \ldots \uplus \mathcal{V}^k$, we say that $(s_1, \ldots, s_k) \models \varphi$ if φ is satisfied when for each i, the assignment of $\mathcal{V}^i$ is determined by s_i. We say that T *satisfies* $(pre, post)$, denoted $T \models^k (pre, post)$, if for every k terminating executions $\pi^1, \ldots, \pi^k$ of T that start in states $s_1, \ldots, s_k$, respectively, such that $(s_1, \ldots, s_k) \models pre$, it holds that they end in states $t_1, \ldots, t_k$, respectively, such that $(t_1, \ldots, t_k) \models post$.

For example, the *non interference* property may be specified by the following 2-safety property: $pre = \bigwedge_{v \in \mathrm{LowIn}} v^1 = v^2$, $post = \bigwedge_{v \in \mathrm{LowOut}} v^1 = v^2$ where LowIn and LowOut denote subsets of the program inputs, resp. outputs, that are considered "low security" and the rest are classified as "high security". This property asserts that every 2 terminating executions that start in states that agree on the "low security" inputs end in states that agree on the low security outputs, i.e., the outcome does not depend on any "high security" input and, hence, does not leak secure information.

Checking k-safety properties reduces to checking ordinary safety properties by creating a *self composed program* that consists of k copies of the transition system, each

[3] Our results can be extended to arbitrary safety (and k-safety) properties by introducing "observable" states to which the property may refer.

with its own copy of the variables, that run in parallel in some way. Thus, the self composed program is defined over variables $\mathcal{V}^{\|k} = \mathcal{V}^1 \uplus \ldots \uplus \mathcal{V}^k$, where $\mathcal{V}^i = \{v^i \mid v \in \mathcal{V}\}$ denotes the variables associated with the ith copy. For example, a common composition is a *lock-step* composition in which the copies execute simultaneously. The resulting composed transition system $T^{\|k} = (S^{\|k}, R^{\|k}, F^{\|k})$ is defined such that $S^{\|k} = S \times \ldots \times S$, $F^{\|k} = \bigwedge_{i=1}^{k} F(\mathcal{V}^i)$ and $R^{\|k} = \bigwedge_{i=1}^{k} R(\mathcal{V}^j, \mathcal{V}^{j\prime})$. Note that $R^{\|k}$ is defined over $\mathcal{V}^{\|k} \uplus \mathcal{V}^{\|k\prime}$ (as usual). Then, the k-safety property $(pre, post)$ is satisfied by T if and only if an ordinary safety property $(pre, post)$ is satisfied by $T^{\|k}$. More general notions of *self composition* are investigated in Sect. 3.

3 Inferring Self Compositions for Restricted Languages of Inductive Invariants

Any self-composition is sufficient for reducing k-safety to safety, e.g., lock-step, sequential, synchronous, asynchronous, etc. However, the choice of the self-composition used determines the difficulty of the resulting safety problem. Different self composed programs would require different inductive invariants, some of which cannot be expressed in a given logical language.

In this section, we formulate the problem of inferring a self composition function such that the obtained self composed program may be verified with a given language of inductive invariants. We are, therefore, interested in inferring both the self composition function and the inductive invariant for verifying the resulting self composed program. We start by formulating the kind of self compositions that we consider.

In the sequel, we fix a transition system $T = (S, R, F)$ with a set of variables $\mathcal{V}$.

3.1 Semantic Self Composition

Roughly speaking, a k self composition of T consists of k copies of T that execute together in some order, where steps may interleave or be performed simultaneously. The order is determined by a self composition function, which may also be viewed as a scheduler that is responsible for scheduling a subset of the copies in each step. We consider *semantic* compositions in which the order may depend on the *states* of the different copies, as well as the correlations between them (as opposed to *syntactic* compositions that only depend on the control locations of the copies, but may not depend on the values of other variables):

Definition 1 (Semantic Self Composition Function). *A* semantic k self composition function *(k-composition function for short) is a function $f : S^k \to \mathbb{P}(\{1..k\})$, mapping each k-state to a* nonempty *set of copies that are to participate in the next step of the self composed program*[4].

[4] We consider *memoryless* composition functions. Compositions that depend on the history of the (joint) execution are supported via ghost state added to the program to track the history.

We represent a k-composition function f by a set of logical conditions, with a condition C_M for every nonempty subset $M \subseteq \{1..k\}$ of the copies. For each such $M \subseteq \{1..k\}$, the condition C_M is defined over $\mathcal{V}^{\|k} = \mathcal{V}^1 \uplus \ldots \uplus \mathcal{V}^k$, and hence it represents a set of k-states, with the meaning that all the k-states that satisfy C_M are mapped to M by f:

$$f(s_1, \ldots, s_k) = M \text{ if and only if } (s_1, \ldots, s_k) \models C_M.$$

To ensure that the function is well defined, we require that $(\bigvee_M C_M) \equiv \mathit{true}$, which ensures that every k-state satisfies at least one of the conditions. We also require that for every $M_1 \neq M_2$, $C_{M_1} \wedge C_{M_2} \equiv \mathit{false}$, hence every k-state satisfies at most one condition. Together these requirements ensure that the conditions induce a partition of the set of all k-states. In the sequel, we identify a k-composition function f with its symbolic representation via conditions $\{C_M\}_M$ and use them interchangeably.

Definition 2 (Composed Program). *Given a k-composition function f, represented via conditions C_M for every nonempty set $M \subseteq \{1..k\}$, we define the k* self composition *of T to be the transition system $T^f = (S^{\|k}, R^f, F^{\|k})$ over variables $\mathcal{V}^{\|k} = \mathcal{V}^1 \uplus \ldots \uplus \mathcal{V}^k$ defined as follows: $F^{\|k} = \bigwedge_{i=1}^k F^i$, where $F^i = F(\mathcal{V}^i)$, and*

$$R^f = \bigvee_{\emptyset \neq M \subseteq \{1..k\}} (C_M \wedge \varphi_M) \quad \textit{where} \quad \varphi_M = \bigwedge_{j \in M} R(\mathcal{V}^j, \mathcal{V}^{j\prime}) \wedge \bigwedge_{j \notin M} \mathcal{V}^j = \mathcal{V}^{j\prime}$$

Thus, in T^f, the set of states consists of k-states ($S^{\|k} = S \times \ldots \times S$), the terminal states are k-states in which all the individual states are terminal, and the transition relation includes a transition from $(s_1, \ldots, s_k)$ to $(s'_1, \ldots, s'_k)$ if and only if $f(s_1, \ldots, s_k) = M$ and $(\forall i \in M.\ (s_i, s'_i) \in R) \wedge (\forall i \notin M.\ s_i = s'_i)$. That is, every transition of T^f corresponds to a simultaneous transition of a subset M of the k copies of T, where the subset is determined by the self composition function f. If $f(s_1, \ldots, s_k) = M$, then for every $i \in M$ we say that i is *scheduled* in $(s_1, \ldots, s_k)$.

Example 2. A k self composition that runs the k copies of T sequentially, one after the other, corresponds to a k-composition function f defined by $f(s_1, \ldots, s_k) = \{i\}$ where $i \in \{1..k\}$ is the minimal index of a non-terminal state in $\{s_1, \ldots, s_k\}$. If all states in $\{s_1, \ldots, s_k\}$ are terminal then $i = k$ (or any other index). This is encoded as follows: for every $1 \leq i < k$, $C_{\{i\}} = \neg F^i \wedge \bigwedge_{j<i} F^j$, $C_{\{k\}} = \bigwedge_{j<k} F^j$ and $C_M = \mathit{false}$ for every other $M \subseteq \{1..k\}$.

Example 3. The lock-step composition that runs the k copies of T synchronously corresponds to a k-self composition function f defined by $f(s_1, \ldots, s_k) = \{1, \ldots, k\}$, and encoded by $C_{\{1,\ldots,k\}} = \mathit{true}$ and $C_M = \mathit{false}$ for every other $M \subseteq \{1..k\}$.

In order to ensure soundness of a reduction of k-safety to safety via self composition, one has to require that the self composition function does not "starve" any copy of the transition system that is about to terminate if it continues to execute. We refer to this requirement as *fairness*.

Definition 3 (Fairness). *A k-self composition function f is* fair *if for every k terminating executions $\pi^1, \ldots, \pi^k$ of T there exists an execution $\pi^{\|}$ of T^f such that for every copy $i \in \{1..k\}$, the projection of $\pi^{\|}$ to i is π^i.*

Note that by the definition of the terminal states of T^f, $\pi^{\|}$ as above is guaranteed to be terminating. We say that the ith copy *terminates* in $\pi^{\|}$ if $\pi^{\|}$ contains a k-state $(s_1, \ldots, s_k)$ such that $s_i \in F$. Fairness may be enforced in a straightforward way by requiring that whenever $f(s_1, \ldots, s_k) = M$, the set M includes no index i for which $s_i \in F$, unless all have terminated. Since we assume that terminal states may only transition to themselves, a weaker requirement that suffices to ensure fairness is that M includes at least one index i for which $s_i \notin F$, unless there is no such index.

The following claim is now straightforward:

Lemma 1. *Let T be a transition system, $(pre, post)$ a k-safety property, and f a fair k-composition function for T and $(pre, post)$. Then*

$$T \models^k (pre, post) \text{ iff } \; T^f \models (pre, post).$$

Proof (sketch). Every terminating execution of T^f corresponds to k terminating executions of T. Fairness of f ensures that the converse also holds.

To demonstrate the necessity of the fairness requirement, consider a (non-fair) self composition function f that maps every state to $\{1\}$. Then, regardless of what the actual transition system T does, the resulting self composition T^f satisfies every pre-post specification vacuously, as it never reaches a terminal state.

Remark 1. While we require the conditions $\{C_M\}_M$ defining a self composition function f to induce a partition of $S^{\|k}$ in order to ensure that f is well defined as a (total) function, the requirement may be relaxed in two ways. First, we may allow C_{M_1} and C_{M_2} to overlap. This will add more transitions and may make the task of verifying the composed program more difficult, but it maintains the soundness of the reduction. Second, it suffices that the conditions cover the set of *reachable states* of the composed program rather than the entire state space. These relaxations do not damage soundness. Technically, this means that f represented by the conditions is a relation rather than a function. We still refer to it as a function and write $f(s_1, \ldots, s_k) = M$ to indicate that $(s_1, \ldots, s_k) \models C_M$, not excluding the possibility that $(s_1, \ldots, s_k) \models M'$ for $M' \neq M$ as well. We note that as long as the language used to describe compositions is closed under Boolean operations, we can always extract from the conditions $\{C_M\}_M$ a function f'. This is done as follows: First, to prevent the overlap between conditions, determine an arbitrary total order $<$ on the sets $M \subseteq \{1..k\}$ and set $C'_M := C_M \wedge \bigwedge_{N<M} \neg C_N$. Second, to ensure that the conditions cover the entire state space, set $C'_{\{1..k\}} := C'_{\{1..k\}} \vee \neg(\bigvee_M C_M)$. It is easy to verify that f' defined by $\{C'_M\}_M$ is a total self composition function and that if f is fair, then so is f'.

3.2 The Problem of Inferring Self Composition with Inductive Invariant

Lemma 1 states the soundness of the reduction of k-safety to ordinary safety. Together with the ability to verify safety by means of an inductive invariant, this leads to a verification procedure. However, while soundness of the reduction holds for *any* self composition, an inductive invariant in a given language may exist for the composed program

resulting from some compositions but not from others. We therefore consider the self composition function and the inductive invariant together, as a pair, leading to the following definition.

Definition 4. *Let T be a transition system and $(pre, post)$ a k safety property. For a formula Inv over $\mathcal{V}^{\|k}$ and a self composition function f represented by conditions $\{C_M\}_M$, we say that (f, Inv) is a* composition-invariant *pair for T and $(pre, post)$ if the following conditions hold:*

- $pre \implies Inv$ *(initiation of Inv),*
- *for every* $\emptyset \neq M \subseteq \{1..k\}$, $Inv \wedge C_M \wedge \varphi_M \implies Inv'$ *(consecution of Inv for* R^f*),*
- $Inv \implies \left((\bigwedge_{j=1}^{k} F^j) \rightarrow post\right)$ *(safety of Inv),*
- $Inv \implies \bigvee_M C_M$ *(f covers the reachable states),*
- *for every* $\emptyset \neq M \subseteq \{1..k\}$, $C_M \wedge (\bigvee_{j=1}^{k} \neg F^j) \implies \bigvee_{j \in M} \neg F^j$ *(f is fair).*

As commented in Remark 1, we relax the requirement that $(\bigvee_M C_M) \equiv true$ to $Inv \implies \bigvee_M C_M$, thus ensuring that the conditions cover all the reachable states. Since the reachable states of T^f are determined by $\{C_M\}_M$ (which define f), this reveals the interplay between the self composition function and the inductive invariant. Furthermore, we do not require that $C_{M_1} \wedge C_{M_2} \equiv false$ for $M_1 \neq M_2$, hence a k-state may satisfy multiple conditions. As explained earlier, these relaxations do not damage soundness. Furthermore, if we construct from f a self composition function f' as described in Remark 1, Inv would be an inductive invariant for $T^{f'}$ as well.

Lemma 2. *If there exists a composition-invariant pair (f, Inv) for T and $(pre, post)$, then $T \models^k (pre, post)$.*

If we do not restrict the language in which f and Inv are specified, then the converse also holds. However, in the sequel we are interested in the ability to verify k-safety with a given language, e.g., one for which the conditions of Definition 4 belong to a decidable fragment of logic and hence can be discharged automatically.

Definition 5 (Inference in $\mathcal{L}$). *Let $\mathcal{L}$ be a logical language. The problem of inferring a composition-invariant pair in $\mathcal{L}$ is defined as follows. The input is a transition system T and a k-safety property $(pre, post)$. The output is a composition-invariant pair (f, Inv) for T and $(pre, post)$ (as defined in Definition 4), where $Inv \in \mathcal{L}$ and f is represented by conditions $\{C_M\}_M$ such that $C_M \in \mathcal{L}$ for every $\emptyset \neq M \subseteq \{1..k\}$. If no such pair exists, the output is "no solution".*

When no solution exists, it does not necessarily mean that $T \not\models^k (pre, post)$. Instead, it may be that the language $\mathcal{L}$ is simply not expressive enough. Unfortunately, for expressive languages (e.g., quantified formulas or even quantifier free linear integer arithmetic), the problem of inferring an inductive invariant alone is already undecidable, making the problem of inferring a composition-invariant pair undecidable as well:

Lemma 3. *Let $\mathcal{L}$ be closed under Boolean operations and under substitution of a variable with a value, and include equalities of the form $v = a$, where v is a variable and a is a value (of the same sort). If the problem of inferring an inductive invariant in $\mathcal{L}$ is undecidable, then so is the problem of inferring a composition-invariant pair in $\mathcal{L}$.*

For example, linear integer arithmetic satisfies the conditions of the lemma. This motivates us to restrict the languages of inductive invariants. Specifically, we consider languages defined by a finite set of predicates. We consider *relational* predicates, defined over $\mathcal{V}^{\|k} = \mathcal{V}^1 \uplus \ldots \uplus \mathcal{V}^k$. For a finite set of predicates $\mathcal{P}$, we define $\mathcal{L}_\mathcal{P}$ to be the set of all formulas obtained by Boolean combinations of the predicates in $\mathcal{P}$.

Definition 6 (Inference using predicate abstraction). *The problem of inferring a predicate-based composition-invariant pair is defined as follows. The input is a transition system T, a k-safety property $(pre, post)$, and a finite set of predicates $\mathcal{P}$. The output is the solution to the problem of inferring a composition-invariant pair for T and $(pre, post)$ in $\mathcal{L}_\mathcal{P}$.*

Remark 2. It is possible to decouple the language used for expressing the self composition function from the language used to express the inductive invariant. Clearly, different sets of predicates (and hence languages) can be assigned to the self composition function and to the inductive invariant. However, since inductiveness is defined with respect to the transitions of the composed system, which are in turn defined by the self composition function, if the language defining f is not included in the language defining Inv, the conditions C_M themselves would be over-approximated when checking the requirements of Definition 4 and therefore would incur a precision loss. For this reason, we use the same language for both.

Since the problem of invariant inference in $\mathcal{L}_\mathcal{P}$ is PSPACE-hard [23], a reduction from the problem of inferring inductive invariants to the problem of inferring composition-invariant pairs (similar to the one used in the proof of Lemma 3) shows that composition-invariant inference in $\mathcal{L}_\mathcal{P}$ is also PSPACE-hard:

Theorem 1. *Inferring a predicate-based composition-invariant pair is PSPACE-hard.*

4 Algorithm for Inferring Composition-Invariant Pairs

In this section, we present Property Directed Self-Composition, PDSC for short—our algorithm for tackling the composition-invariant inference problem for languages of predicates (Definition 6). Namely, given a transition system T, a k-safety property $(pre, post)$ and a finite set of predicates $\mathcal{P}$, we address the problem of finding a pair (f, Inv), where f is a self composition function and Inv is an inductive invariant for the composed transition system T^f obtained from f, and both of them are in $\mathcal{L}_\mathcal{P}$, i.e., defined by Boolean combinations of the predicates in $\mathcal{P}$.

We rely on the property that a transition system (in our case T^f) has an inductive invariant in $\mathcal{L}_\mathcal{P}$ if and only if its abstraction obtained using $\mathcal{P}$ is safe. This is because, the set of reachable abstract states is the strongest set expressible in $\mathcal{L}_\mathcal{P}$ that satisfies initiation and consecution. Given T^f, this allows us to use predicate abstraction to either obtain an inductive invariant in $\mathcal{L}_\mathcal{P}$ for T^f (if the abstraction of T^f is safe) or determine that no such inductive invariant exists (if an abstract counterexample trace is obtained). The latter indicates that a different self composition function needs to be considered. A naive realization of this idea gives rise to an iterative algorithm that starts from an

$f \leftarrow$ lockstep , $E \leftarrow \emptyset$, *Unreach* $\leftarrow$ *false*
while *(true)* **do**
  $(res, Inv, cex) \leftarrow$ `Abs_Reach`$(\mathcal{P}, T^f, pre, post, Unreach)$
  if $res =$ *safe* **then return** $(f, Inv(\mathcal{P}))$
  $(\hat{s}, M) \leftarrow$ `Last_Step`(cex)
  $E \leftarrow E \cup \{(\hat{s}, M)\}$
  while *(*`All_Excluded_Or_Starving`$(\hat{s}, E)$*)* **do**
    Unreach $\leftarrow$ *Unreach* $\vee \hat{s}$
    if *Unreach* $\wedge \varphi_{pre}(\mathcal{B}) \not\equiv$ *false* **then return** "no solution in $\mathcal{L}_{\mathcal{P}}$"
    $cex \leftarrow$ `Remove_Last_Step`(cex)
    $(\hat{s}, M) \leftarrow$ `Last_Step`(cex)
    $E \leftarrow E \cup \{(\hat{s}, M)\}$
  $f \leftarrow$ `Modify_SC`$(f, \hat{s}, E)$

Algorithm 1. PDSC: Property-Directed Self-Composition.

arbitrary initial composition function and in each iteration computes a new composition function. At the worst case such an algorithm enumerates all self composition functions defined in $\mathcal{L}_{\mathcal{P}}$, i.e., has time complexity $O(2^{2^{|\mathcal{P}|}})$. Importantly, we observe that, when no inductive invariant exists for some composition function, we can use the abstract counterexample trace returned in this case to (i) generalize and eliminate multiple composition functions, and (ii) identify that some abstract states must be unreachable if there is to be a composition-invariant pair, i.e., we "block" states in the spirit of *property directed reachability* [5,13]. This leads to the algorithm depicted in Algorithm 1 whose worst case time complexity is $2^{O(|\mathcal{P}|)}$. Next, we explain the algorithm in detail.

Finding an Inductive Invariant for a Given Composition Function Using Predicate Abstraction. We use predicate abstraction [17,27] to check if a given candidate composition function has a corresponding inductive invariant. This is done as follows. The abstraction of T^f using $\mathcal{P}$, denoted $A_{\mathcal{P}}(T^f)$, is a transition system $(\hat{S}, \hat{R})$ defined over variables $\mathcal{B}$, where $\mathcal{B} = \{b_p \mid p \in \mathcal{P}\}$ (we omit the terminal states). $\hat{S} = \{0,1\}^{\mathcal{B}}$, i.e., each abstract state corresponds to a valuation of the Boolean variables representing $\mathcal{P}$. An abstract state $\hat{s} \in \hat{S}$ represents the following set of states of T^f:

$$\gamma(\hat{s}) = \{s^{\|} \in S^{\|k} \mid \forall p \in \mathcal{P}.\ s^{\|} \models p \Leftrightarrow \hat{s}(b_p) = 1\}$$

We extend γ to sets of states and to formulas representing sets of states in the usual way. The abstract transition relation is defined as usual:

$$\hat{R} = \{(\hat{s}_1, \hat{s}_2) \mid \exists {s^{\|}}_1 \in \gamma(\hat{s}_1)\ \exists {s^{\|}}_2 \in \gamma(\hat{s}_2).\ ({s^{\|}}_1, {s^{\|}}_2) \in R^f\}$$

Note that the set of abstract states in $A_{\mathcal{P}}(T^f)$ does *not* depend on f.

Notation. We sometimes refer to an abstract state $\hat{s} \in \hat{S}$ as the formula $\bigwedge_{\hat{s}(b_p)=1} b_p \wedge \bigwedge_{\hat{s}(b_p)=0} \neg b_p$. For a formula $\psi \in \mathcal{L}_{\mathcal{P}}$, we denote by $\psi(\mathcal{B})$ the result of substituting each $p \in \mathcal{P}$ in ψ by the corresponding Boolean variable b_p. For the opposite direction, given

a formula ψ over $\mathcal{B}$, we denote by $\psi(\mathcal{P})$ the formula in $\mathcal{L}_\mathcal{P}$ resulting from substituting each $b_p \in \mathcal{B}$ in ψ by p. Therefore, $\psi(\mathcal{P})$ is a symbolic representation of $\gamma(\psi)$.

Every set defined by a formula $\psi \in \mathcal{L}_\mathcal{P}$ is precisely represented by $\psi(\mathcal{B})$ in the sense that $\gamma(\psi(\mathcal{B}))$ is equal to the set of states defined by ψ, i.e., $\psi(\mathcal{B})$ is a precise abstraction of ψ. For simplicity, we assume that the termination conditions as well as the pre/post specification can be expressed precisely using the abstraction, in the following sense:

Definition 7. *$\mathcal{P}$ is* adequate *for T and $(pre, post)$ if there exist $\varphi_{pre}, \varphi_{post}, \varphi_{F^i} \in \mathcal{L}_\mathcal{P}$ such that $\varphi_{pre} \equiv pre$, $\varphi_{post} \equiv post$ and $\varphi_{F^i} \equiv F^i$ (for every copy $i \in \{1..k\}$).*

The following lemma provides the foundation for our algorithm:

Lemma 4. *Let T be a transition system, $(pre, post)$ a k safety property, and $\mathcal{P}$ a finite set of predicates adequate for T and $(pre, post)$. For a self composition function f defined via conditions $\{C_M\}_M$ in $\mathcal{L}_\mathcal{P}$, there exists an inductive invariant Inv in $\mathcal{L}_\mathcal{P}$ such that (f, Inv) is a composition-invariant pair for T and $(pre, post)$ if and only if the following three conditions hold:*

S1 *All reachable states of $A_\mathcal{P}(T^f)$ from $\varphi_{pre}(\mathcal{B})$ satisfy $(\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})) \rightarrow \varphi_{post}(\mathcal{B})$,*
S2 *All reachable states of $A_\mathcal{P}(T^f)$ from $\varphi_{pre}(\mathcal{B})$ satisfy $\bigvee_M C_M(\mathcal{B})$, and*
S3 *For every $\emptyset \neq M \subseteq \{1..k\}$, $C_M(\mathcal{B}) \wedge (\bigvee_{j=1}^{k} \neg\varphi_{F^j}(\mathcal{B})) \implies \bigvee_{j \in M} \neg\varphi_{F^j}(\mathcal{B})$.*

Furthermore, if the conditions hold, then the symbolic representation of the set of abstract states of $A_\mathcal{P}(T^f)$ reachable from $\varphi_{pre}(\mathcal{B})$ is a formula Inv over $\mathcal{B}$ such that $(f, Inv(\mathcal{P}))$ is a composition-invariant pair for T and $(pre, post)$.

Algorithm 1 starts from the lock-step self composition function (Line 1), which is fair[5], and constructs the next candidate f such that condition **S3** in Lemma 4 always holds (see discussion of `Modify_SC`). Thus, condition **S3** need not be checked explicitly.

Algorithm 1 checks whether conditions **S1** and **S2** hold for a given candidate composition function f by calling `Abs_Reach` (Line.3) – both checks are performed via a (non-)reachability check in $A_\mathcal{P}(T^f)$, checking whether a state violating $(\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})) \rightarrow \varphi_{post}(\mathcal{B})$ or $\bigvee_M C_M(\mathcal{B})$ is reachable from $\varphi_{pre}(\mathcal{B})$. Algorithm 1 maintains the abstract states that are not in $\bigvee_M C_M(\mathcal{B})$ by the formula *Unreach* defined over $\mathcal{B}$, which is initialized to *false* (as the lock-step composition function is defined for every state) and is updated in each iteration of Algorithm 1 to include the abstract states violating $\bigvee_M C_M(\mathcal{B})$. If no abstract state violating **S1** or **S2** is reachable, i.e., the conditions hold, then `Abs_Reach` returns the (potentially overapproximated) set of reachable abstract states, represented by a formula Inv over $\mathcal{B}$. In this case, by Lemma 4, $(f, Inv(\mathcal{P}))$ is a composition-invariant pair (line 4). Otherwise, an abstract counterexample trace is obtained. (We can of course apply bounded model checking to check if the counterexample is real; we omit this check as our focus is on the case where the system is safe.)

Remark 3. In practice, we do not construct $A_\mathcal{P}(T^f)$ explicitly. Instead, we use the *implicit predicate abstraction* approach [6].

[5] Any fair self composition can be chosen as the initial one; we chose lock-step since it is a good starting point in many applications.

Eliminating Self Composition Candidates Based on Abstract Counterexamples. An abstract counterexample to conditions **S1** or **S2** indicates that the candidate composition function f has no corresponding Inv. Violation of **S1** can only be resolved by changing f such that the abstract trace is no longer feasible. Violation of **S2** may, in principle, also be resolved by extending the definition of f such that it is defined for all the abstract states in the counterexample trace.

However, to prevent the need to explore both options, our algorithm maintains the following invariant for every candidate self composition function f that it constructs:

Claim. Every abstract state that is *not* in $\bigvee_M C_M(\mathcal{B})$ is not reachable w.r.t. the abstract composed program of *any* composition function that is part of a composition-invariant pair for T and $(pre, post)$.

This property clearly holds for the lock-step composition function, which the algorithm starts with, since for this composition, $\bigvee_M C_M(\mathcal{B}) \equiv true$. As we explain in Corollary 2, it continues to hold throughout the algorithm.

As a result of this property, whenever a candidate composition function f does not satisfy condition **S1** or **S2**, it is never the case that $\bigvee_M C_M(\mathcal{B})$ needs to be extended to allow the abstract states in cex to be reachable. Instead, the abstract counterexample obtained in violation of the conditions needs to be eliminated by modifying f.

Let $cex = \hat{s}_1, \ldots, \hat{s}_{m+1}$ be an abstract counterexample of $A_{\mathcal{P}}(T^f)$ such that $\hat{s}_1 \models \varphi_{pre}(\mathcal{B})$ and $\hat{s}_{m+1} \models (\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})) \wedge \neg\varphi_{post}(\mathcal{B})$ (violating **S1**) or $\hat{s}_{m+1} \models Unreach$ (violating **S2**). Any self composition f' that agrees with f on the states in $\gamma(\hat{s}_i)$ for every $\hat{s}_i$ that appears in cex has the same transitions in R^f and, hence, the same transitions in $\hat{R}$. It, therefore, exhibits the same abstract counterexample in $A_{\mathcal{P}}(T^{f'})$. Hence, it violates **S1** or **S2** and is not part of any composition-invariant pair.

Notation. Recall that f is defined via conditions $C_M \in \mathcal{L}_{\mathcal{P}}$. This ensures that for every abstract state $\hat{s}$, f is defined in the same way for all the states in $\gamma(\hat{s})$. We denote the value of f on the states in $\gamma(\hat{s})$ by $f(\hat{s})$ (in particular, $f(\hat{s})$ may be undefined). We get that $f(\hat{s}) = M$ if and only if $\hat{s} \models C_M(\mathcal{B})$.

Using this notation, to eliminate the abstract counterexample cex, one needs to eliminate at least one of the transitions in cex by changing the definition of $f(\hat{s}_i)$ for *some* $1 \leq i \leq m$. For a new candidate function f' this may be encoded by the disjunctive constraint $\bigvee_{i=1}^{m} f'(\hat{s}_i) \neq f(\hat{s}_i)$. However, we observe that a stronger requirement may be derived from cex based on the following lemma:

Lemma 5. *Let f be a self composition function and $cex = \hat{s}_1, \ldots, \hat{s}_{m+1}$ a counterexample trace in $A_{\mathcal{P}}(T^f)$ such that $\hat{s}_1 \models \varphi_{pre}(\mathcal{B})$ but $\hat{s}_{m+1} \models (\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})) \wedge \neg\varphi_{post}(\mathcal{B})$ or $\hat{s}_{m+1} \models Unreach$. Then for any self composition function f' such that $f'(\hat{s}_m) = f(\hat{s}_m)$, if $\hat{s}_m$ is reachable in $A_{\mathcal{P}}(T^{f'})$ from $\varphi_{pre}(\mathcal{B})$, then a counterexample trace to* **S1** *or* **S2** *exists.*

Corollary 1. *If there exists a composition-invariant pair (f', Inv'), then there is also one where $f'(\hat{s}_m) \neq f(\hat{s}_m)$.*

Therefore, we require that in the next self composition candidates the abstract state $\hat{s}_m$ must not be mapped to its current value in f, i.e., $f'(\hat{s}_m) \neq M$, where $f(\hat{s}_m) = M$[6].

Algorithm 1 accumulates these constraints in the set E (Line 6). Formally, the constraint $(\hat{s}, M) \in E$ asserts that C'_M must imply $\neg(\bigwedge_{\hat{s}(b_p)=1} p \wedge \bigwedge_{\hat{s}(b_p)=0} \neg p)$, and hence $f'(\hat{s}) \neq M$.

Identifying Abstract States that Must Be Unreachable. A new candidate self composition is constructed such that it satisfies all the constraints in E (thus ensuring that no abstract counterexample will re-appear). In the construction, we make sure to satisfy **S3** (fairness). Therefore, for every abstract state $\hat{s}$, we choose a value $f'(\hat{s})$ that satisfies the constraints in E and is *non-starving*: a value M is starving for $\hat{s}$ if $\hat{s} \models \bigvee_{j=1}^{k} \neg\varphi_{F^j}(\mathcal{B})$ but $\hat{s} \not\models \bigvee_{j \in M} \neg\varphi_{F^j}(\mathcal{B})$, i.e., some of the copies have not terminated in $\hat{s}$ but none of the non-terminating copies is scheduled. (Due to adequacy, a value M is starving for $\hat{s}$ if and only if it is starving for every $s^{\|} \in \gamma(\hat{s})$.)

If for some abstract state $\hat{s}$, all the non-starving values have already been excluded (i.e., $(\hat{s}, M) \in E$ for every non-starving M), we conclude that there is *no* f' such that $\hat{s}$ is reachable in $A_{\mathcal{P}}(T^{f'})$ and f' is part of a composition-invariant pair:

Lemma 6. *Let $\hat{s} \in \hat{S}$ be an abstract state such that for every $\emptyset \neq M \subseteq \{1..k\}$ either M is starving for $\hat{s}$ or $(\hat{s}, M) \in E$. Then, for every f' that satisfies* **S3***, if $A_{\mathcal{P}}(T^{f'})$ satisfies* **S1** *and* **S2***, then $\hat{s}$ is unreachable in $A_{\mathcal{P}}(T^{f'})$.*

Corollary 2. *If there exists a composition-invariant pair (f', Inv'), then $\hat{s}$ is unreachable in $A_{\mathcal{P}}(T^{f'})$.*

This is because no matter how the self composition function f' would be defined, $\hat{s}$ is guaranteed to have an outgoing abstract counterexample trace in $A_{\mathcal{P}}(T^{f'})$.

We, therefore, turn $f'(\hat{s})$ to be undefined. As a result, condition **S2** of Algorithm 4 requires that $\hat{s}$ will be unreachable in $A_{\mathcal{P}}(T^{f'})$. In Algorithm 1, this is enforced by adding $\hat{s}$ to *Unreach* (Line 8).

Every abstract state $\hat{s}$ that is added to *Unreach* is a strengthening of the safety property by an additional constraint that needs to be obeyed in any composition-invariant pair, where obtaining a composition-invariant pair is the target of the algorithm. This makes our algorithm *property directed*.

If an abstract state that satisfies $\varphi_{pre}(\mathcal{B})$ is added to *Unreach*, then Algorithm 1 determines that no solution exists (Line 9). Otherwise, it generates a new constraint for E based on the abstract state preceding $\hat{s}$ in the abstract counterexample (Line 12).

Constructing the Next Candidate Self Composition Function. Given the set of constraints in E and the formula *Unreach*, `Modify_SC` (Line 13) generates the next candidate composition function by (i) taking a constraint $(\hat{s}, M)$ such that $\hat{s} \not\models$ *Unreach* (typically the one that was added last), (ii) selecting a non-starving value M_{new} for $\hat{s}$ (such

[6] If the conditions $\{C_M\}_M$ defining f may overlap, we consider the condition C_M by which the transition from $\hat{s}_m$ to $\hat{s}_{m+1}$ was defined.

a value must exist, otherwise $\hat{s}$ would have been added to *Unreach*), and (iii) updating the conditions defining f' as follows:

$$C'_M = C_M \wedge \neg\hat{s}(\mathcal{P}) \qquad C'_{M_{\text{new}}} = (C_{M_{\text{new}}} \vee \hat{s}(\mathcal{P}))$$

The conditions of other values remain as before. This definition is facilitated by the fact that the same set of predicates is used both for defining f' and for defining the abstract states $\hat{s} \in \hat{S}$ (by which Inv is obtained). Note that in practice we do not explicitly turn f' to be undefined for $\gamma(\mathit{Unreach})$. However, these definitions are ignored. The definition ensures that f' is non-starving (satisfying condition **S3**) and that no two conditions $C'_{M_1} \neq C'_{M_2}$ overlap. While the latter is not required, it also does not restrict the generality of the approach (since the language we consider is closed under Boolean operations).

Theorem 2. *Let T be a transition system, $(pre, post)$ a k-safety property and $\mathcal{P}$ a set of predicates over $\mathcal{V}^{\|k}$. If Algorithm 1 returns "no solution" then there is no composition-invariant pair for T and $(pre, post)$ in $\mathcal{L}_\mathcal{P}$. Otherwise, $(f, Inv(\mathcal{P}))$ returned by Algorithm 1 is a composition-invariant pair in $\mathcal{L}_\mathcal{P}$, and thus $T \models^k (pre, post)$.*

Complexity. Each iteration of Algorithm 1 adds at least one constraint to E, excluding a potential value for f over some abstract state $\hat{s}$. An excluded values is never re-used. Hence, the number of iterations is at most the number of abstract states, $2^{|\mathcal{P}|}$, multiplied by the number of potential values for each abstract state, $n = 2^k$. Altogether, the number of iterations is at most $O(2^{|\mathcal{P}|} \cdot 2^k)$. Each iteration makes one call to `Abs_Reach` which checks reachability via predicate abstraction, hence, assuming that satisfiability checks in the original logic are at most exponential, its complexity is $2^{O(|\mathcal{P}|)}$. Therefore, the overall complexity of the algorithm is $2^{O(|\mathcal{P}|)+k}$. Typically, k is a small constant, hence the complexity is dominated by $2^{O(|\mathcal{P}|)}$.

5 Evaluation and Conclusion

Implementation. We implemented PDSC (Algorithm 1) in Python on top of Z3 [25]. Its input is a transition system encoded by Constrained Horn Clauses (CHC) in SMT2 format, a k-safety property and a set of predicates. The abstraction is implicitly encoded using the approach of [6], and is parameterized by a composition function that is modified in each iteration. For reachability checks (`Abs_Reach`) we use SPACER [22], which supports LRA and arrays. For the set of predicates used by PDSC, we implemented an automatic procedure that mines these predicates from the CHC. Additional predicates may be added manually.

Experiments. To evaluate PDSC, we compare it to SYNONYM [26], the current state of the art in k-safety verification.

To show the effectiveness of PDSC, we consider examples that require a *nontrivial* composition (these examples are detailed in [29]). We emphasize that the motivation for these example is originated in real-life scenarios. For example, Fig. 1 follows a pattern of constant-time execution. The results of these experiments are summarized in Table 1.

Table 1. Examples that require semantic compositions

Program	PDSC		SYNONYM
	Time(s)	Iteations	
DoubleSquareNI	7	33	fail
HalfSquareNI	3.4	28	fail
ArrayIntMod	58.2	168	fail
SquaresSum	2.8	4	fail
ArrayInsert	19.5	102	fail

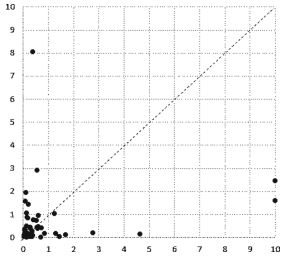

Fig. 2. Runtime comparison (in sec.): PDSC (x-axis) and SYNONYM (y-axis).

PDSC is able to find the right composition function and prove all of the examples, while SYNONYM cannot verify any of them. We emphasize that for these examples, lock-step composition is not sufficient. However, PDSC infers a composition that depends on the programs' state (variable values), rather than just program locations.

Next we consider Java programs from [26,30], which we manually converted to C, and then converted to CHC using SEAHORN [19]. For all but 3 examples, only 2 types of predicates, which we mined automatically, were sufficient for verification: (i) relational predicates derived from the pre- and post-conditions, and (ii) for simple loops that have an index variable (e.g., for iterating over an array), an equality predicate between the copies of the indices. These predicates were sufficient since we used a large-step encoding of the transition relation, hence the abstraction via predicates takes effect only at cut-points. For the remaining 3 examples, we manually added 2–4 predicates. With the exception of 1 example where a timeout of 10 seconds was reached, all examples were solved with a lock-step composition function. Yet, we include them to show that on examples with simple compositions PDSC performs similarly to SYNONYM. This can be seen in Fig. 2.

Conclusion and Future Work. This work formulates the problem of inferring a self composition function together with an inductive invariant for the composed program, thus capturing the interplay between the self composition and the difficulty of verifying the resulting composed program. To address this problem we present PDSC– an algorithm for inferring a semantic self composition, directed at verifying the composed program with a given language of predicates. We show that PDSC manages to find non-trivial self compositions that are beyond reach of existing tools. In future work, we are interested in further improving PDSC by extending it with additional (possibly lazy) predicate discovery abilities. This has the potential to both improve performance and verify properties over wider range of programs. Additionally, we consider exploring further generalization techniques during the inference procedure.

Acknowledgements. This publication is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [759102-SVIS]). The research was partially supported by Len Blavatnik and the Blavatnik Family foundation, the Blavatnik Interdisciplinary

Cyber Research Center, Tel Aviv University, the Israel Science Foundation (ISF) under grant No. 1810/18 and the United States-Israel Binational Science Foundation (BSF) grant No. 2016260.

References

1. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18–23, 2017. pp. 362–375 (2017). https://doi.org/10.1145/3062341.3062378
2. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Proceedings of the FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20–24, 2011, pp. 200–214 (2011). https://doi.org/10.1007/978-3-642-21437-0_17
3. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: asymmetric product programs for relational program verification. In: Artemov, S., Nerode, A. (eds.) LFCS 2013. LNCS, vol. 7734, pp. 29–43. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35722-0_3
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28–30 June 2004, Pacific Grove, CA, USA. pp. 100–114 (2004). https://doi.org/10.1109/CSFW.2004.17
5. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
6. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 46–61. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_4
7. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018)
8. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23–25 June 2008, pp. 51–65 (2008). https://doi.org/10.1109/CSF.2008.7
9. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
10. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 127–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_7
11. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Relational verification through horn clause transformation. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 147–169. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_8
12. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
13. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, Austin, TX, USA, October 30 - November 02, 2011, pp. 125–134 (2011). http://dl.acm.org/citation.cfm?id=2157675
14. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 502–529. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_18

15. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden - September 15–19, 2014, pp. 349–360 (2014). https://doi.org/10.1145/2642937.2642987
16. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26–31, 2009. pp. 466–471 (2009). https://doi.org/10.1145/1629911.1630034
17. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
18. Gupta, S., Saxena, A., Mahajan, A., Bansal, S.: Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 365–382. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_22
19. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
21. Karimpour, J., Isazadeh, A., Noroozi, A.A.: Verifying observational determinism. In: Federrath, H., Gollmann, D. (eds.) SEC 2015. IAICT, vol. 455, pp. 82–93. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18467-8_6
22. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
23. Lahiri, S.K., Qadeer, S.: Complexity and algorithms for monomial and clausal predicate abstraction. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 214–229. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_18
24. Mordvinov, D., Fedyukovich, G.: Synchronizing constrained Horn clauses. In: LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7–12, 2017, pp. 338–355 (2017). http://www.easychair.org/publications/paper/340359
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 164–182. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_9
27. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_38
28. Sharma, R., Schkufza, E., Churchill, B.R., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013. pp. 391–406 (2013). https://doi.org/10.1145/2509136.2509509
29. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. CoRR abs/1905.07705 (2019). http://arxiv.org/abs/1905.07705

30. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016, pp. 57–69 (2016). https://doi.org/10.1145/2908080.2908092
31. Strichman, O., Veitsman, M.: Regression verification for unbalanced recursive functions. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 645–658. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_39
32. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24
33. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., Malik, S.: Lazy self-composition for security verification. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 136–156. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_11
34. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5