Tel Aviv University
Raymond and Beverly Sackler Faculty of Exact Sciences
The Blavatnik School of Computer Science

PROPERTY DIRECTED SELF COMPOSITION

by

Ron Shemer

under the supervision of
Dr. Sharon Shoham

Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science

2019

Abstract

Property Directed Self Composition

Ron Shemer
Master of Science
School of Computer Science
Tel Aviv University

We address the problem of verifying *k-safety properties*: properties that refer to k 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.

Acknowledgements

Contents

List of Tables

Chapter 1

Introduction

Many relational properties, such as noninterference [13], determinism [22], 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, 31]. 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 [21] 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 works [2, 29, 15, 19, 26, 32]. However, most efforts are focused on compositions that are pre-defined, or only depend on syntactic similarities.

In this thesis, 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 [33, 3, 15].

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, see Section 3.2.1 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. 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 $\mathcal{L}$.

Motivating Example. To demonstrate our approach, consider the program in Figure 1.1. 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 Figure 1.1. 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.

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

```
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: $i_1 = i_2$, $i_1 < len_1$, $i_2 < len_2$,
     $A_1[i_1] < h_1$, $A_2[i_2] < h_2$, $len_1 = len_2$,
     $len_1 = len_2 + 1$, $len_2 = len_1 + 1$
```

```
composition:
if($pc_1 < 3$ && ($pc_2 > 0$ || !$cond_1$)
   && ($pc_2 == 3$||($pc_2 == 0$ && $cond_2$)))
       step(1);
else if ($pc_2 < 3$ && ($pc_1 > 0$ || !$cond_2$)
         && ($pc_1 == 3$ || ($pc_1 == 0$ && $cond_1$)))
           step(2);
else step(1,2);

       $cond_1$ := $i_1 < len_1$ && $A_1[i_1] < h_1$
       $cond_2$ := $i_2 < len_2$ && $A_2[i_2] < h_2$
```

Figure 1.1: Constant-time insert to an array.

composition function depicted in Figure 1.1, 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 [18], 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.

We implemented PDSC using SEAHORN [20], Z3 [12] and SPACER [23] 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 [29, 26] that are proven with the trivial lock-step composition. On these examples, PDSC is comparable to state of the art tools.

1.1 Main Results

The contributions of this thesis may be summarized as follows.

- We formulate the problem of inferring a semantic composition function jointly with the inductive invariant needed to verify the composed program, where both are restricted to a given language.
- We present PDSC– a property-directed algorithm that solves the inference problem for languages based on predicate abstraction by integrating the search for a composition function into the model checking of the composed program. The complexity of PDSC is $2^{O(|\mathcal{P}|)}$, where this reduced complexity (compared to the naive algorithm) is obtained by using generalized composition elimination.
- We implement PDSC and show that it solves 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 show that PDSC is comparable to state of the art tools when evaluated over examples with trivial composition.

Chapter 2

Preliminaries

In this chapter, we provide background on programs and their modeling as transition systems, on safety properties and on k-safety properties.

2.1 Transition Systems

In this work 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*, also called a *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.

In this work we consider formulas in First Order Logic (FOL) with theories, specifically Linear Integer Arithmetic (LIA) and the theory of arrays, but the methods and definitions we present in the following sections can be extended to support other fragments of FOL. In the implementation section the examples are naturally proved within these theories. The decidability of the fragment that is used affects the soundness and completeness of the inference algorithm presented in the next chapter.

2.2 Safety and inductive invariants

We consider safety properties (and later k-safety properties) defined via pre/post conditions. Our results can be extended to more general safety properties by introducing "observable" states to which the property may refer. A *safety property* is a pair $(\mathit{pre}, \mathit{post})$ where $\mathit{pre}, \mathit{post}$ are formulas over $\mathcal{V}$, representing subsets of S, denoting the pre- and post-condition, respectively. T *satisfies* $(\mathit{pre}, \mathit{post})$, denoted $T \models (\mathit{pre}, \mathit{post})$, if every terminating execution π of T that starts in a state s_0 such that $s_0 \models \mathit{pre}$ ends in a state s such that $s \models \mathit{post}$. In other words, for every state s that is reachable in T from a state in pre we have that $s \models F \to \mathit{post}$.

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 $(\mathit{pre}, \mathit{post})$ is a formula Inv such that (1) $\mathit{pre} \Rightarrow \mathit{Inv}$ (initiation), (2) $\mathit{Inv} \wedge R \Rightarrow \mathit{Inv}'$ (consecution), and (3) $\mathit{Inv} \Rightarrow (F \to \mathit{post})$ (safety), where $\varphi \Rightarrow \psi$ denotes the validity of $\varphi \to \psi$, 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 $T \models (\mathit{pre}, \mathit{post})$. This is because conditions (1) and (2) ensure that Inv over-approximates the set of states that are reachable from pre, hence condition (3) ensures that every such state satisfies $F \to \mathit{post}$.

2.3 k-safety

A *k-safety property* refers to k interacting executions of T. Similarly to an ordinary safety property, it is defined by $(\mathit{pre}, \mathit{post})$, 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* $(\mathit{pre}, \mathit{post})$, denoted $T \models^k (\mathit{pre}, \mathit{post})$, 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 $(s_1, \ldots, s_k) \models \mathit{pre}$, it holds that they end in states $t_1, \ldots, t_k$, respectively, such that $(t_1, \ldots, t_k) \models \mathit{post}$.

Example 1 (Non Interference)**.** *The* non interference *property may be specified by the following 2-safety property:*

$$\mathit{pre} = \bigwedge_{v \in \mathrm{LowIn}} v^1 = v^2 \qquad \mathit{post} = \bigwedge_{v \in \mathrm{LowOut}} v^1 = v^2$$

where LowIn *and* LowOut *denote subsets of the program inputs, respectively 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 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 $(\mathit{pre}, \mathit{post})$ is satisfied by T if and only if an ordinary safety property $(\mathit{pre}, \mathit{post})$ is satisfied by $T^{\|k}$. More general notions of *self composition* are investigated in Chapter 3.

Chapter 3

Inferring Self Compositions With 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 chapter, 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 2 (Semantic Self Composition Function)**.** *A* semantic k self composition function *(k-composition function for short) is a function $f : S^k \to \mathcal{P}(\{1..k\})$, mapping each k-state* *to a* nonempty *set of copies that are to participate in the next step of the self composed program.*

Note that 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 $(\bigvee_M C_M) \equiv \text{True}$, which ensures that every k-state satisfies at least one of the conditions. We also require that for every $M_1 \neq M_2$, $C_{M_1} \wedge C_{M_2} \equiv \text{False}$, 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 3 (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*

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

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, i.e.,

$$(s_1, \ldots, s_k) \in F^{\|k} \text{ if and only if } s_i \in F^i \text{ for every } 1 \leq i \leq k,$$

and the transition relation is defined as follows:

$$\begin{aligned}((s_1,\dots,s_k),(s'_1,\dots,s'_k)) \in R^f \text{ if and only if } & f(s_1,\dots,s_k) = M, \text{ and} \\ & (s_i, s'_i) \in R \text{ for every } i \in M, \text{and} \\ & s_i = s'_i \text{ for every } i \notin M\end{aligned}$$

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 $f(s_1,\dots,s_k) = M$, then for every $i \in M$ we say that i is *scheduled* in $(s_1,\dots,s_k)$.

Example 4. *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 $f(s_1,\dots,s_k) = \{i\}$ where $i \in \{1..k\}$ is the minimal index of a non-terminal state in $\{s_1,\dots,s_k\}$. If all states in $\{s_1,\dots,s_k\}$ are terminal then $i = k$ (or any other index). This is encoded as follows: for every $1 \le i < k$, $C_{\{i\}} = \neg F^i \wedge \bigwedge_{j<i} F^j$, $C_{\{k\}} = \bigwedge_{j<k} F^j$ and $C_M = False$ for every other $M \subseteq \{1..k\}$.*

Example 5. *The lock-step composition that runs the k copies of T synchronously corresponds to a k-self composition function f defined by $f(s_1,\dots,s_k) = \{1,\dots,k\}$, and encoded by $C_{\{1,\dots,k\}} = True$ and $C_M = False$ 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 6 (Fairness)**.** *A k-self composition function f is* fair *if for every k terminating executions $\pi^1,\dots,\pi^k$ of T there exists an execution $\pi^{\parallel}$ of T^f such that for every copy $i \in \{1..k\}$, the projection of $\pi^{\parallel}$ to i is π^i.*

Note that by the definition of the terminal states of T^f, $\pi^{\parallel}$ as above is guaranteed to be terminating. We say that the ith copy *terminates* in $\pi^{\parallel}$ if $\pi^{\parallel}$ contains a k-state $(s_1,\dots,s_k)$ such that $s_i \in F$. Fairness may be enforced in a straightforward way by requiring that whenever $f(s_1,\dots,s_k) = M$, 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 7. *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*

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

Proof. $(\Rightarrow)$: Assume $T \models^k (pre, post)$ and let $\pi^{||} = (s_0^1, \dots, s_0^k), \dots, (s_n^1, \dots, s_n^k)$ be a terminating execution of T^f. We show that if $(s_0^1, \dots, s_0^k) \models pre$ then $(s_n^1, \dots, s_n^k) \models post$. For $\pi^{||}$ we consider the k terminating executions of T, denoted $\pi^1, \dots, \pi^k$, obtained by the projection of $\pi^{||}$ on each copy index, i.e., π^i is obtained from the sequence $s_0^i, \dots, s_n^i$ by merging identical consecutive states. From Definition 3 we get that $\pi^1, \dots, \pi^k$ are well defined and are legal terminating executions of T. By definition of k-safety, since $T \models^k (pre, post)$ we get that if $(s_0^1, \dots, s_0^k) \models pre$ then $(s_n^1, \dots, s_n^k) \models post$.

$(\Leftarrow)$: Let $\pi^1, \dots, \pi^k$ be some k terminating executions of T where $\pi^i = s_0^i, \dots, s_{n_i}^i$. Since f is fair, from Definition 6 we get that there exists a terminating execution $\pi^{||} = ({s_0}^1, \dots, s_0^k), \dots, (s_n^1, \dots, s_n^k)$ of T^f such that the projections of $\pi^{||}$ are $\pi^1, \dots, \pi^k$. In particular, this means that $(s_n^1, \dots, s_n^k) = (s_{n_1}^1, \dots, s_{n_k}^k)$, i.e., the last state in $\pi^{||}$ consists of the last states in $\pi^1, \dots, \pi^k$. From $T^f \models (pre, post)$ we conclude that if $(s_0^1, \dots, s_0^k) \models pre$ then $(s_n^1, \dots, s_n^k) \models post$, hence $(s_{n_1}^1, \dots, s_{n_k}^k) \models post$. This holds for any k terminating executions of T and thus $T \models^k (pre, post)$. □

Lemma 7 states the soundness and completeness of the reduction to safety for *any* fair self composition. Intuitively, the variables of the k copies of T are completely disjoint, making the states of the individual copies completely independent. Therefore, the final state reached by the execution of each copy does not depend on the actual interleaving (or scheduling) of the copies. Hence, as long as the self composition function is fair, if some interleaving (determined by the self composition function) violates the postcondition, all of them will. Thus, soundness is ensured for every fair self composition function. Completeness is guaranteed even without the fairness requirement.

To demonstrate the necessity of the fairness requirement for the soundness of the reduction, 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* $(s_1, \ldots, s_k) \models C_{M'}$ *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:*

- *To prevent the overlap between conditions, determine an arbitrary total order* $<$ *on the sets* $M \subseteq \{1..k\}$ *and set* $C'_M := C_M \wedge \bigwedge_{N<M} \neg C_N$*.*
- *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 7 states the soundness and completeness 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 and completeness 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, hindering the completeness of the overall verification procedure.

In this section, we present an example of a k-safety problem such that when a natural self composition that is based on the control structure only is applied, no inductive invariant in the language of Quantifier-Free Linear Integer arithmetic (QFLIA) can establish the desired property. Motivated by this example, we then introduce the problem of inferring a self composition function that admits an inductive invariant in a given language.

3.2.1 Demonstrating the Interplay Between Self Composition and Inductive Invariants

We illustrate the effect of the self composition function on the difficulty of verifying the obtained composed program, as well as the need for a semantic self composition function on the simple example depicted in Figure 3.1. The program receives as input an integer x and a secret bit h, and outputs $y = 2x^2$. The desired specification is that the output does not pre($x_1 == x_2$)

```
doubleSquare(bool h, int x){
  int z, y=0;
  if(h) { z = 2*x; }
  else { z = x; }
  while (z>0) {
        z--;
        y = y+x;
  }
  if(!h) { y = 2*y; }
  return y;
}
```

post($y_1 == y_2$)

```
doubleSquare_sc(bool h1, bool h2, int x){
  int z1, z2, y1 = 0; y2 = 0;
  if(h1) {z1 = 2*x;} else {z1 = x;}
  if(h2) {z2 = 2*x;} else {z2 = x;}
  while (z1>0 || z2>0) {
    if (z1>0) {z1--; y1 = y1+x;}
    if (z2>0) {z2--; y2 = y2+x;}
  }
  if(!h1) {y1 = 2*y1;}
  if(!h2) {y2 = 2*y2;}
  return y1, y2;
}
```

Figure 3.1: (Left) a program that computes $2x^2$; the computation depends on a secret bit h while x is the low input, and (Right) its self composition based on [15].

depend on h, which is indeed the case. Formally, this is a 2-safety property, requiring that in any two terminating executions that start with the same values for x, the final value of y is the same.

Self composition addresses the problem of verifying the 2-safety problem by creating two independent copies of the program: one copy with all variables indexed by 1, and another copy with all variables indexed by 2. This allows reducing the problem of verifying the 2-safety property to the problem of verifying a traditional safety problem (in fact, partial correctness). Namely, when considering the two copies of the program as one program, the desired property is that if the *precondition* $x_1 = x_2$ holds initially, then the *postcondition* $y_1 = y_2$ also holds when (if) both copies terminate. As explained in Section 3.1, the actual interleaving, or the self composition function, does not affect the soundness of the reduction to traditional safety (as long as it is fair).

However, when we turn to verifying the safety of the composed program by finding an inductive invariant in a given language, the specific self composition function used plays a significant role. For example, consider a composition function that "synchronizes" the two copies in each control structure (e.g. [15]). Such a composited program runs the two copies of the loop in parallel until one copy exits the loop, and then continues to run the other copy. We show that for this composition function, there exists *no inductive invariant* in quantifier free linear integer arithmetic (QFLIA) that is sufficient for establishing safety of the composed program.

Proof. If we examine the set *Reach* of the reachable states of the composed program at the exit point we see that it includes (for every natural number n):

$$
\begin{aligned}
(x, y_1, z_1, y_2, z_2) \mapsto \quad & (n, 0, 2n, 0, n) \\
& (n, n, 2n-1, n, n-1) \\
& \ldots \\
& (n, kn, 2n-k, kn, n-k) \\
& \ldots \\
& (n, n^2, n, n^2, 0) \\
& (n, n^2+n, n-1, n^2, 0) \\
& \ldots \\
& (n, n^2+kn, n-k, n^2, 0) \\
& \ldots \\
& (n, 2n^2, 0, n^2, 0)
\end{aligned}
$$

(We omit the second copy of x since both copies are equal in all the reachable states – a fact that is also expressible in QFLIA – and similarly, we omit h_1 and h_2.)

Clearly, an inductive invariant must be satisfied by all of these states, since all of them are reachable. However, we show that any QFLIA formula that is satisfied by all of these states is also satisfied by a state that reaches a bad state (i.e., a state where $y_1 \neq y_2$), thus if it is safe, it necessarily violates the consecution requirement, which means it is not an inductive invariant.

Let $\varphi = \varphi_1 \vee \ldots \varphi_r$ be a QFLIA formula, written in DNF form, where each φ_i is a cube (conjunction of literals). Define $\mathit{Reach}_1, \ldots, \mathit{Reach}_r \subseteq \mathit{Reach}$ such that $\mathit{Reach}_i = \{s \in \mathit{Reach} \mid s \models \varphi_i\}$ includes all states in *Reach* that satisfy φ_i. We show that there exists i such that φ_i is also satisfied by a state that reaches a bad state.

Reach includes infinitely many "points" of the form $n, n^2, n, n^2, 0$ where n is an even number. Therefore, since there are finitely many Reach_i's that together cover *Reach*, there exists i such that Reach_i also includes infinitely many such points. Take two such points $(n, n^2, n, n^2, 0)$ and $(m, m^2, m, m^2, 0)$ in Reach_i where $n \neq m$. Then $(1/2(n+m), 1/2(n^2+m^2), 1/2(n+m), 1/2(n^2+m^2), 0)$ is a state (all values are integers) in the convex hull of Reach_i. In particular, it must satisfy φ_i (φ_i is a cube in LIA that is satisfied by all states in Reach_i, hence it is also satisfied by all states in its convex hull).

However, when executing the while loop starting from the state $x \mapsto 1/2(n+m), y_1 \mapsto$

$1/2(n^2+m^2), z_1 \mapsto 1/2(n+m), y_2 \mapsto 1/2(n^2+m^2), z_2 \mapsto 0$, the outcome is the state $x \mapsto 1/2(n+m), y_1 \mapsto 1/2(n^2+m^2)+1/4(n+m)^2, z_1 \mapsto 0, y_2 \mapsto 1/2(n^2+m^2), z_2 \mapsto 0$, where $y_1 \neq y_2$, hence safety is violated.

This means that φ is not an inductive invariant strong enough to establish safety of the composed program, in contradiction. $\square$

In contrast, with the composition function inferred by PDSC (see Figure 5.2 in Section 5.2.1), the composed program has an inductive invariant in QFLIA.

3.2.2 Composition-Invariant Pairs

We showed that there are k-safety problems for which using a specific self composition might prevent the existence of an inductive invariant (in the given language). For this reason, in our work we consider the self composition function and the inductive invariant together, as a pair, leading to the following definition.

Definition 8. *Let T be a transition system and $(pre, post)$ a k safety property. For a formula Inv over $\mathcal{V}^{\parallel k}$ 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*),*
- $Inv \implies \big((\bigwedge_{j=1}^{k} F^j) \rightarrow post\big)$ *(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 $(\bigvee_M C_M) \equiv \text{True}$ 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 $C_{M_1} \wedge C_{M_2} \equiv \text{False}$ 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 9. *If there exists a composition-invariant pair (f, Inv) for T and $(pre, post)$, then $T \models^k (pre, post)$.*

Proof. Let T be a transition system and $(pre, post)$ a k-safety property, and assume that (f, Inv) is a composition-invariant pair for them, as in Definition 8. We use f to define a fair composition function f' as in Remark 1. We show that Inv is an inductive invariant for $T^{f'}$. The initiation and safety of Inv for $T^{f'}$ do not depend on the composition function and so they hold for $T^{f'}$ due to the initiation and safety conditions of Definition 8 which hold for (f, Inv). We now turn to showing the consecution of Inv for $R^{f'}$. Consider some $M \subseteq \{1..k\}$ and some states $\hat{s}, \hat{s}'$ of the composed program, we will show that $(\hat{s}, \hat{s}') \models \mathit{Inv} \wedge C'_M \wedge \varphi_M \to \mathit{Inv}'$. If $(\hat{s}, \hat{s}') \models \mathit{Inv} \wedge C'_M \wedge \varphi_M$ then $\hat{s} \models C'_M$ and by definition of f', either $\hat{s} \models C_M$ or $\hat{s} \models \neg(\bigvee_M C_M)$. In the first case we get from the consecution of Inv for R^f that $\hat{s}' \models \mathit{Inv}'$. Otherwise, $\hat{s} \models \neg(\bigvee_M C_M)$ and we assumed also that $\hat{s} \models \mathit{Inv}$ (because $(\hat{s}, \hat{s}') \models \mathit{Inv} \wedge C'_M \wedge \varphi_M$). This contradicts the assumption that f covers the reachable states (see Definition 8) and therefore we conclude that it cannot be the case that $\hat{s} \models \neg(\bigvee_M C_M)$. This proves that Inv is an inductive invariant for $T^{f'}$. By construction of f' it is a fair composition function (Remark 1). We can use Lemma 7 and conclude that $T \models^k (\mathit{pre}, \mathit{post})$. □

3.2.3 The Problem of Inferring Composition-Invariant Pairs in a Given Language

If we do not restrict the language in which f and Inv are specified, then the converse of Lemma 9 also holds, i.e., $T \models^k (\mathit{pre}, \mathit{post})$ implies the existence of a composition-invariant pair for T and $(\mathit{pre}, \mathit{post})$. 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 8 belong to a decidable fragment of logic and hence can be discharged automatically. We therefore define the problem of inferring a composition-invariant pair in a given language.

Definition 10 (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 $(\mathit{pre}, \mathit{post})$. The output is a composition-invariant pair (f, Inv) for T and $(\mathit{pre}, \mathit{post})$ (as defined in Definition 8), where $\mathit{Inv} \in \mathcal{L}$ 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 $T \not\models^k (\mathit{pre}, \mathit{post})$. 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 11. *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 $\mathcal{L}$.*

Proof. We show a reduction from the ordinary invariant inference problem in $\mathcal{L}$ to the problem of inferring a composition-invariant pair in $\mathcal{L}$. Given a transition system T and an ordinary safety property $(\mathit{pre}, \mathit{post})$ the reduction constructs a transition system $T^* = (S^*, R^*, F^*)$ over $\mathcal{V}^* = \mathcal{V} \uplus \{b\}$, where b is a new Boolean variable such that when $b = \text{True}$ the original transitions are taken and when $b = \text{False}$ the systems remains in the same state, which is also added to the set of terminal states. Formally, for every $v \in \mathcal{V}$, let a_v be an arbitrary fixed value in the domain of v. For example, if v is Boolean, $a_v = \text{False}$. The reduction constructs

$$R^* = (b \wedge R \wedge b') \vee (\neg b \wedge (\bigwedge_{v \in \mathcal{V}} v' = a_v) \wedge \neg b') \qquad F^* = F \vee (\neg b \wedge \bigwedge_{v \in \mathcal{V}} v' = a_v),$$

and the following 2-safety property:

$$\mathit{pre}^* = \left(b^1 \wedge \mathit{pre}(\mathcal{V}^1) \wedge \neg b^2 \wedge \bigwedge_{v \in \mathcal{V}} v^2 = a_v \right) \quad \mathit{post}^* = \left(b^1 \wedge \mathit{post}(\mathcal{V}^1) \wedge \neg b^2 \wedge \bigwedge_{v \in \mathcal{V}} v^2 = a_v \right).$$

That is, the first copy is "initialized" with $b = \text{True}$ and with the original pre-condition and is required to terminate in a state that satisfies the original post-condition, while the second copy is initialized with $b = \text{False}$, and with the value a_v for each original variable, and is required to terminate in the same state. Clearly, if T has an inductive invariant Inv for $(\mathit{pre}, \mathit{post})$, then $(f, b^1 \wedge \mathit{Inv}(\mathcal{V}^1) \wedge \neg b^2 \wedge \bigwedge_{v \in \mathcal{V}} v^2 = a_v)$ is a composition-invariant pair for T^* and $(\mathit{pre}^*, \mathit{post}^*)$, where f is defined by $C_{\{1,2\}} = \text{True}$ and $C_M = \text{False}$ for any other M, which is clearly in $\mathcal{L}$. For the converse direction, if T^* has a composition-invariant pair (f, Inv^*) for $(\mathit{pre}^*, \mathit{post}^*)$ then Inv obtained by substituting each positive occurrence of b^2 in Inv^* by False, each negative occurrence of b^2 by True and each occurrence of v^2 by a_v is an inductive invariant for T and $(\mathit{pre}, \mathit{post})$. □

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 12 (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 8 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 [24], 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 11) shows that composition-invariant inference in $\mathcal{L}_{\mathcal{P}}$ is also PSPACE-hard:

Theorem 13. *Inferring a predicate-based composition-invariant pair is PSPACE-hard.*

Chapter 4

Algorithm for Inferring Composition-Invariant Pairs

In this chapter, we present Property Directed Self-Composition, PDSC for short — our algorithm for tackling the composition-invariant inference problem for languages of predicates (Definition 12). 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 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, 14]. This leads to the algorithm depicted in Algorithm 1 whose worst case time complexity is $2^{O(|\mathcal{P}|)}$.

```
$f \leftarrow$ lockstep
$E \leftarrow \emptyset$
$Unreach \leftarrow$ False
while (true) do
    // abstract reachability check for a candidate composition function
    $(res, Inv, cex) \leftarrow$ Abs_Reach($\mathcal{P}, T^f, pre, post, Unreach$)
    if $res = safe$ then return $(f, Inv(\mathcal{P}))$
    // accumulate constraints from "bad" composition functions
    $(\hat{s}, M) \leftarrow$ 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
            // abstract counterexample exists for any composition function
            return "no solution in $\mathcal{L}_{\mathcal{P}}$"
        end
        // traverse the trace backwards in an attempt to "block" more states
        $cex \leftarrow$ Remove_Last_Step($cex$)
        $(\hat{s}, M) \leftarrow$ Last_Step($cex$)
        $E \leftarrow E \cup \{(\hat{s}, M)\}$
    end
    // choose a new composition function satisfying the constraints
    $f \leftarrow$ Modify_SC($f, \hat{s}, E$)
end
```

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

Next, we explain the algorithm in detail, establish its correctness and complexity, and demonstrate its execution via an example.

4.1 Finding an inductive invariant for a given composition function using predicate abstraction

We use predicate abstraction [18, 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 the set $\mathcal{B} = \{b_p \mid p \in \mathcal{P}\}$ of Boolean variables, where a Boolean variable b_p is introduced for each predicate $p \in \mathcal{P}$. (Technically, our definition of a transition system includes a set of terminal states – these are important for examining safety properties defined via pre/post specifications. However, we do not consider such properties of the abstract transition system, and therefore we omit the terminal states from the abstract transition system.) The set of abstract states is $\hat{S} = \{0,1\}^{\mathcal{B}}$, 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:

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

and may be represented by the following formula over $\mathcal{B} \uplus \mathcal{B}'$:

$$\hat{R} \ = \ \exists \mathcal{V}^{\|k}\ \exists \mathcal{V}^{\|k'}.\ \bigwedge_{p \in \mathcal{P}} (b_p \leftrightarrow p) \wedge \big(\bigvee_{M} C_M \wedge \varphi_M\big) \wedge \bigwedge_{p \in \mathcal{P}} (b'_p \leftrightarrow p')$$

where φ_M defines the transition relation formula for a step taken by all the copies in M, see Definition 3. That is, every abstract transition is associated with a set M of copies that make (an abstract) move. Note that the set of abstract states in $A_{\mathcal{P}}(T^f)$ does *not* depend on f.

Notation 1. *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 14. $\mathcal{P}$ *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 15. *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* $(\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})) \to \varphi_{post}(\mathcal{B})$*,*

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* $(pre, post)$*.*

Proof. The proof relies on the following statement, denoted by $(*)$: for a formula φ in $\mathcal{L}_{\mathcal{P}}$ and an abstract state $\hat{s}$, for every $s^{\|} \in \gamma(\hat{s})$ it holds that $s^{\|} \models \varphi \Leftrightarrow \hat{s} \models \varphi(\mathcal{B})$ (which follows by induction on the structure of a formula in $\mathcal{L}_{\mathcal{P}}$, relying on the definition of $\gamma(\hat{s})$). In particular, this implies that for a formula ψ over $\mathcal{B}$, it holds that $s^{\|} \models \psi(\mathcal{P}) \Leftrightarrow \hat{s} \models \psi$ whenever $s^{\|} \in \gamma(\hat{s})$.

($\Rightarrow$) Let T, $(pre, post)$ and $\mathcal{P}$ be as described, and let (f, Inv) be a composition-invariant pair for T and $(pre, post)$ in $\mathcal{L}_{\mathcal{P}}$. We first show that every (abstract) state that is reachable from $\varphi_{pre}(\mathcal{B})$ in $A_{\mathcal{P}}(T^f)$ satisfies $Inv(\mathcal{B})$. Let $\hat{s}$ be such a reachable state. Then there exists an abstract trace $\hat{s}_1, \ldots, \hat{s}_m$ such that $\hat{s}_1 \models \varphi_{pre}(\mathcal{B})$, $\hat{s}_m = \hat{s}$ and $(\hat{s}_i, \hat{s}_{i+1}) \in \hat{R}$ for every $1 \leq i < m$. Consider a concrete state ${s^{\|}}_1$ of T^f such that ${s^{\|}}_1 \in \gamma(\hat{s}_1)$, then $\hat{s}_1 \models \varphi_{pre}(\mathcal{B})$ and from $(*)$ we get ${s^{\|}}_1 \models \varphi_{pre}$, hence ${s^{\|}}_1 \models pre$ (recall that $pre \equiv \varphi_{pre}$). From the definition of a composition-invariant pair (Definition 8) we get that ${s^{\|}}_1 \models Inv$ (initiation). Since Inv is in $\mathcal{L}_{\mathcal{P}}$ we get from $(*)$ that also $\hat{s}_1 \models Inv(\mathcal{B})$. For $\hat{s}_2$, the next state in the abstract trace, it also holds that $\hat{s}_2 \models Inv(\mathcal{B})$: since $(\hat{s}_1, \hat{s}_2) \in \hat{R}$, we know that there exist some ${s^{\|}}_a \in \gamma(\hat{s}_1)$ and ${s^{\|}}_b \in \gamma(\hat{s}_2)$ such that $({s^{\|}}_a, {s^{\|}}_b) \in R^f$, using $(*)$ we get that ${s^{\|}}_a \models Inv$, the consecution of Inv implies ${s^{\|}}_b \models Inv$ and from $(*)$ we get $\hat{s}_2 \models Inv(\mathcal{B})$. By induction over the length of the abstract trace we get that $\hat{s} \models Inv(\mathcal{B})$. We now turn to show that conditions **S1**–**S3** hold. First, the safety of Inv for T^f together with adequacy of $\mathcal{P}$ and $(*)$ imply that $Inv(\mathcal{B}) \implies \big((\bigwedge_{j=1}^{k} \varphi_F^j(\mathcal{B})) \to \varphi_{post}(\mathcal{B})\big)$, and since all the reachable states of $A_{\mathcal{P}}(T^f)$ satisfy $Inv(\mathcal{B})$, **S1** follows. Similarly, the covering requirement of f together with the property that C_M is in $\mathcal{L}_{\mathcal{P}}$ for every M and together with $(*)$ imply **S2**. Finally, **S3** is implied directly from the fairness of f (Definition 8).

($\Leftarrow$) Assume that for T, $(pre, post)$, $\mathcal{P}$ and some composition function f as described, conditions **S1**–**S3** hold. Condition **S1** ensures that $A_{\mathcal{P}}(T^f)$ satisfies the safety property $(\varphi_{pre}(\mathcal{B}), \varphi_{post}(\mathcal{B}))$, when we augment $A_{\mathcal{P}}(T^f)$ with a set of terminal states given by the formula $\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})$. Hence, there exists an inductive invariant Inv over $\mathcal{B}$ for $A_{\mathcal{P}}(T^f)$ and $(\varphi_{pre}(\mathcal{B}), \varphi_{post}(\mathcal{B}))$. Furthermore, condition **S2** ensures that there exists such Inv for which $Inv \implies \bigvee_M C_M(\mathcal{B})$ (for example, such Inv may be obtained by conjoining the inductive invariant ensured by **S1** with another inductive invariant that establishes **S2**). To conclude the proof we show that $(f, Inv(\mathcal{P}))$ is a composition-invariant pair for T and $(pre, post)$, as defined in Definition 8. First, initiation and safety of *Inv* with respect to $A_{\mathcal{P}}(T^f)$ and $(\varphi_{pre}(\mathcal{B}), \varphi_{post}(\mathcal{B}))$ imply initiation and safety (respectively) of $Inv(\mathcal{P})$ with respect to T and $(pre, post)$ due to $(*)$, and the fact that $pre \equiv \varphi_{pre}$ and $post \equiv \varphi_{post}$ (adequacy of $\mathcal{P}$). As for consecution of $Inv(\mathcal{P})$: for a pair of states $s^{\|}{}_1, s^{\|}{}_2$ in T^f such that $(s^{\|}{}_1, s^{\|}{}_2) \in R^f$, if $s^{\|}{}_1 \in \gamma(\hat{s}_1)$ and $s^{\|}{}_2 \in \gamma(\hat{s}_2)$, then $(\hat{s}_1, \hat{s}_2) \in \hat{R}$. Therefore, if $s^{\|}{}_1 \models Inv(\mathcal{P})$ then $\hat{s}_1 \models Inv$ (according to $(*)$), and from consecution of *Inv* in $A_{\mathcal{P}}(T^f)$ also $\hat{s}_2 \models Inv$, and from $(*)$ we get $s^{\|}{}_2 \models Inv(\mathcal{P})$ and conclude the consecution of $Inv(\mathcal{P})$ in T^f. Similarly, for covering of f: recall that $Inv \implies \bigvee_M C_M(\mathcal{B})$, hence by $(*)$, $Inv(\mathcal{P}) \implies \bigvee_M C_M$, i.e., f covers the states satisfying $Inv(\mathcal{P})$. Finally, the fairness of f follows from **S3**. □

Algorithm 1 starts from the lock-step self composition function (Line 1), which is fair (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). Then, the algorithm constructs (if needed) the next candidate f such that condition **S3** in Lemma 15 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 5) – 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 15, $(f, Inv(\mathcal{P}))$ is a composition-invariant pair (Line 6). 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] (see Section 5.1.2).*

4.2 Eliminating self composition candidates based on abstract counterexamples

Every iteration of Algorithm 1 checks whether using a certain candidate composition function leads to successful verification of the property. If it does not, then an abstract counterexample trace is obtained. In this section, we explain how such a trace is used to prune the space of candidate composition functions before constructing another candidate.

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 16. *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 function, $\bigvee_M C_M(\mathcal{B}) \equiv \text{True}$. As we explain in Corollary 20, 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 $\hat{s}_{m+1} \models \mathit{Unreach}$ (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 2. *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 17. *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.*

Proof. Suppose that $\hat{s}_m$ is reachable in $A_{\mathcal{P}}(T^{f'})$ from $\varphi_{pre}(\mathcal{B})$. Then there exists a trace $\hat{s}'_1, \ldots, \hat{s}'_m$ in $A_{\mathcal{P}}(T^{f'})$ such that $\hat{s}'_1 \models \varphi_{pre}(\mathcal{B})$ and $\hat{s}'_m = \hat{s}_m$. Since $f'(\hat{s}_m) = f(\hat{s}_m)$, the outgoing transitions of $\hat{s}_m$ are the same in both $A_{\mathcal{P}}(T^f)$ and $A_{\mathcal{P}}(T^{f'})$. In particular, the transition $(\hat{s}_m, \hat{s}_{m+1})$ from $A_{\mathcal{P}}(T^f)$ also exists in $A_{\mathcal{P}}(T^{f'})$. Therefore, $cex' = \hat{s}'_1, \ldots, \hat{s}'_m, \hat{s}_{m+1}$ is a trace to $\hat{s}_{m+1}$ in $A_{\mathcal{P}}(T^{f'})$. If $\hat{s}_{m+1} \models (\bigwedge_{i=1}^{k} \varphi_{F^i}(\mathcal{B})) \wedge \neg\varphi_{post}(\mathcal{B})$, then cex' is a counterexample to **S1** in $A_{\mathcal{P}}(T^{f'})$ as well. Consider the case where $\hat{s}_{m+1} \models$ *Unreach*. By the construction of *Unreach*, this indicates that $\hat{s}_{m+1}$ has an outgoing abstract trace that leads to violation of **S1** or **S2** with every non-starving self composition function, and in particular in $A_{\mathcal{P}}(T^{f'})$. □

Corollary 18. *If there exists a composition-invariant pair (f', Inv'), then there is also one where $f'(\hat{s}_m) \neq f(\hat{s}_m)$.*

Proof. If $f'(\hat{s}_m) = f(\hat{s}_m)$, then by Lemma 17, $\hat{s}_m$ is necessarily unreachable in $A_{\mathcal{P}}(T^{f'})$ from $\varphi_{pre}(\mathcal{B})$. Therefore, if we change $f'(\hat{s}_m)$, all the requirements of Lemma 15 will still hold. If no alternative value that admits the fairness requirement exists, then $f'(\hat{s}_m)$ can remain undefined. □

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 $f(\hat{s}_m) = M$. 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.

Algorithm 1 accumulates these constraints in the set E (Line 8). 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 $f'(\hat{s}) \neq M$.

4.3 Identifying abstract states that must be unreachable

The constraints accumulated in E are used to construct a new candidate composition function, as well as to deduce that certain abstract states must be unreachable in a composed transition system if the verification is to succeed. Such abstract states are added to *Unreach* and are therefore "blocked". This step makes the algorithm *property directed* in spirit. Next we explain how this is done.

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 19. *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 $A_{\mathcal{P}}(T^{f'})$.*

Proof. If f' satisfies **S3** and $A_{\mathcal{P}}(T^{f'})$ satisfies **S1** and **S2**, then according to Lemma 15 f' is a part of some composition-invariant pair (f', Inv) for T. Furthermore, as shown in the proof of Lemma 15, every (abstract) state that is reachable from $\varphi_{pre}(\mathcal{B})$ in $A_{\mathcal{P}}(T^{f'})$ satisfies $\mathit{Inv}(\mathcal{B})$. Assume to the contrary that $\hat{s}$ is reachable in $A_{\mathcal{P}}(T^{f'})$. Then $\hat{s} \models \mathit{Inv}(\mathcal{B})$. According to Definition 8, f' must be defined for $\hat{s}$, thus $f'(\hat{s}) = M'$ for some $\emptyset \neq M' \subseteq \{1 \dots k\}$. Since f' is fair (satisfies **S3**) it must be the case that $(\hat{s}, M') \in E$. According to the algorithm, at some iteration there was a composition function f'' with $f''(\hat{s}) = M'$ that caused adding $(\hat{s}, M')$ to E, i.e., there was a counterexample to **S1** or **S2** in $A_{\mathcal{P}}(T^{f''})$ in the form of a trace to $\hat{s}$. Then Lemma 17 implies that there is also a counterexample to **S1** or **S2** in $A_{\mathcal{P}}(T^{f'})$ because $f'(\hat{s}) = f''(\hat{s}) = M'$. This contradicts the assumption that $A_{\mathcal{P}}(T^{f'})$ satisfies **S1** and **S2**. □

Corollary 20. *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 Lemma 15 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 10).

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 12). Otherwise, it generates a new constraint for E based on the abstract state preceding $\hat{s}$ in the abstract counterexample (Line 16).

4.4 Constructing the next candidate self composition function

At the end of each iteration, the algorithm constructs a candidate composition function that satisfies the constraints accumulated in E. In this section, we explain the procedure for constructing the new candidate.

Given the set of constraints in E and the formula *Unreach*, `Modify_SC` (Line 18) 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 a value must exist, otherwise $\hat{s}$ would have been added to *Unreach*), and (iii) updating the conditions defining f' as follows:

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

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).

4.5 Correctness and Complexity

In this section we summarize the correctness of Pdsc as well as its complexity, and discuss a possible optimization.

Theorem 21. *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 $T \models^k (pre, post)$.*

Proof. Algorithm 1 returns " no solution" when $Unreach \wedge \varphi_{pre}(\mathcal{B})$ is satisfiable. This means that there is an abstract state $\hat{s}$ that satisfies $\varphi_{pre}(\mathcal{B})$ but also satisfies *Unreach*. By the construction of *Unreach*, this means that $\hat{s}$ must be unreachable from $\varphi_{pre}(\mathcal{B})$ in any $A_{\mathcal{P}}(T^{f'})$ such that (f', Inv') a composition-invariant pair in $\mathcal{L}_{\mathcal{P}}$ (see Corollary 20). Hence, no such (f', Inv') exists. Conversely, Algorithm 1 returns $(f, Inv(\mathcal{P}))$ when all the conditions listed in Lemma 15 are met, thus $(f, Inv(\mathcal{P}))$ is a composition-invariant pair. □

4.5.1 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 $2^{O(|\mathcal{P}|)}$.

4.5.2 Optimization

To further enhance the search for a suitable self composition function, it is possible to generalize the constraints that are added to E. Rather than adding $(\hat{s}_m, M)$, where $\hat{s}_m$ is the abstract state before last in an abstract counterexample trace, we can first generalize $\hat{s}$ by finding its minimal sub-cube a such that all the states in $\gamma(a)$ transition to $\hat{s}_{m+1}$ when the copies in M make a step. (Alternatively, different generalization schemes based on a weakest precondition computation may be used). This way, each constraint may block a value M for multiple abstract states at once.

4.6 Example

In this section we use the program SquaresSum depicted in Figure 4.1, which computes the sum of squares of a given integer range, to demonstrate the fundamental actions that PDSC performs. For this program, we consider the monotonicity property – a 2-safety property with pre-condition $[a_1, b_1] \supset [a_2, b_2]$ and post-condition $c_1 > c_2$. When lock-step composition is applied, no corresponding inductive invariant exists in the language of predicates described in Figure 4.1. Intuitively, the reason is that for lock-step composition, the condition $c_1 > c_2$ is violated at the end of an unbounded number of loop iterations, i.e., for every $n \in \mathbb{N}$ there

```
pre($a_1 < a_2$ && $b_1 > b_2$)

squaresSum(int a, int b){
  assume(0 < a < b);
  int c=0;
  while (a<b) {c+=a*a; a++;}
  return c;
}

post($c_1 > c_2$)
```

```
predicates:
  $c_1 > c_2$, $c_1 = c_2$, $a_1 < a_2$
  $a_1 = a_2$, $b_1 > b_2$, $a_1 < b_1$
  $a_2 < b_2$, $b_1 > 1$, $b_2 > 1$


composition function=
  if ($a_1 < a_2$)
        step(1);
  else
        step(1,2);
```

Figure 4.1: A program that computes $\sum_{a \leq n < b} n^2$.

exists an input such that $c_1 > c_2$ does not hold in more than n loop iterations. However, Pdsc manages to verify the monotonicity property by inferring a composition function that schedules the copies such that $c_1 > c_2$ holds from the first iteration of copy 2 and onwards. The corresponding composition function appears in Figure 4.1.

To explain the run of Pdsc on this program, we start by encoding the transition system of SquaresSum via formulas over the set of variables $\mathcal{V} = \{a, b, c\}$ (and their primed counterparts $\mathcal{V}' = \{a', b', c'\}$). The transition relation and terminal states are defined by:

$$\begin{aligned} R(\mathcal{V}, \mathcal{V}') &= (a' = a + 1) \wedge (a < b) \wedge (c' = c + a^2) \wedge (b' = b) \\ F(\mathcal{V}) &= (a \geq b) \end{aligned}$$

Note that in this encoding, each transition corresponds to an execution of the loop body.

The monotonicity property is encoded by:

$$\begin{aligned} pre(\mathcal{V}^1, \mathcal{V}^2) &= (a_1 < a_2) \wedge (b_1 > b_2) \wedge \\ &\qquad (c_1 = 0) \wedge (a_1 < b_1) \wedge (a_1 > 0) \wedge \\ &\qquad (c_2 = 0) \wedge (a_2 < b_2) \wedge (a_2 > 0) \\ post(\mathcal{V}^1, \mathcal{V}^2) &= (c_1 > c_2) \end{aligned}$$

Note that the pre-condition encodes not only the condition $[a_1, b_1] \supset [a_2, b_2]$ but also the assumption $0 < a < b$ in both copies as well as the initialization of c.

To run Pdsc, the following set of predicates $\mathcal{P}$ is supplied:

$$\mathcal{P} = \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2,\ a_1 = a_2,\ b_1 > b_2,\ a_1 < b_1,\ a_2 < b_2,\ c_1 > c_2,\ c_1 = c_2\}$$

PDSC starts with the lock-step composition function, specified by:

$$\begin{aligned} C_{\{1\}} &= \text{False} \\ C_{\{2\}} &= \text{False} \\ C_{\{1,2\}} &= \text{True} \end{aligned}$$

The composition function induces the self-composed program represented by the following formulas:

$$\begin{aligned} R^{||2}(\mathcal{V}^1, \mathcal{V}^2, \mathcal{V}^{1'}, \mathcal{V}^{2'}) = &\big(C_{\{1,2\}}(\mathcal{V}^1, \mathcal{V}^2) \wedge R(\mathcal{V}^1, \mathcal{V}^{1'}) \wedge R(\mathcal{V}^2, \mathcal{V}^{2'})\big) \vee \\ &\big(C_{\{1\}}(\mathcal{V}^1, \mathcal{V}^2) \wedge R(\mathcal{V}^1, \mathcal{V}^{1'}) \wedge (\mathcal{V}^2 = \mathcal{V}^{2'})\big) \vee \\ &\big(C_{\{2\}}(\mathcal{V}^1, \mathcal{V}^2) \wedge R(\mathcal{V}^2, \mathcal{V}^{2'}) \wedge (\mathcal{V}^1 = \mathcal{V}^{1'})\big) \\ F^{||2}(\mathcal{V}^1, \mathcal{V}^2) = &F(\mathcal{V}^1) \wedge F(\mathcal{V}^2) \end{aligned}$$

PDSC runs `Abs_Reach` on the composed transition system with the predicates in $\mathcal{P}$ (the implementation of `Abs_Reach` is described in detail in Section 5.1.2). The result is the following abstract counterexample trace, where each abstract state is represented by the predicates that it satisfies and falsifies:

$$\begin{aligned} \hat{s}_0 &= \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2, \neg(a_1 = a_2),\ b_1 > b_2,\ a_1 < b_1,\ a_2 < b_2, \neg(c_1 > c_2),\ c_1 = c_2\} \\ \hat{s}_1 &= \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2, \neg(a_1 = a_2),\ b_1 > b_2,\ a_1 < b_1, \neg(a_2 < b_2), \neg(c_1 > c_2), \neg(c_1 = c_2)\} \\ \hat{s}_2 &= \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2, \neg(a_1 = a_2),\ b_1 > b_2, \neg(a_1 < b_1), \neg(a_2 < b_2), \neg(c_1 > c_2), \neg(c_1 = c_2)\} \end{aligned}$$

This is a counterexample trace to condition **S1** since $\hat{s}_2 \models \varphi_{F^1}(\mathcal{B}) \wedge \varphi_{F^2}(\mathcal{B}) \wedge \neg\varphi_{post}(\mathcal{B})$. The trace is used to eliminate the lock-step composition function. The elimination is performed by extending the set E with a constraint that requires that the abstract state $\hat{s}_1$ is not mapped to the value $\{1,2\}$ in any subsequent candidate composition function.

The following composition function f' is chosen as the next candidate:

$$\begin{aligned} {C_{\{1\}}}' &= \hat{s}_1(\mathcal{P}) \\ {C_{\{2\}}}' &= \text{False} \\ {C_{\{1,2\}}}' &= \neg\hat{s}_1(\mathcal{P}) \end{aligned}$$

f' is generated by `Modify_SC` as explained in Section 4.4. It is a fair composition function that satisfies the constraints in E, since $E = \{(\hat{s}_1, \{1,2\})\}$. Note, though, that it is still not a part of a composition-invariant pair for this problem. The next call to `Abs_Reach`, with the new candidate composition function, produces another abstract counterexample trace that violates condition **S1**:

$$\hat{s}'_0 = \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2, \neg(a_1 = a_2),\ b_1 > b_2,\ a_1 < b_1,\ a_2 < b_2, \neg(c_1 > c_2),\ c_1 = c_2\}$$
$$\hat{s}'_1 = \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2, \neg(a_1 = a_2),\ b_1 > b_2,\ a_1 < b_1, \neg(a_2 < b_2), \neg(c_1 > c_2), \neg(c_1 = c_2)\}$$
$$\hat{s}'_2 = \{a_1 > 0,\ a_2 > 0,\ a_1 < a_2, \neg(a_1 = a_2),\ b_1 > b_2, \neg(a_1 < b_1), \neg(a_2 < b_2), \neg(c_1 > c_2),\ c_1 = c_2\}$$

This trace leads to another constraint on the next composition functions explored. Note that $\hat{s}'_1 = \hat{s}_1$, therefore, the new constraint is captured by the pair $(\hat{s}_1, \{1\})$ that is added to E. The only option left for a composition function f'' that satisfies the constraints in E is to define $f''(\hat{s}_1) = \{2\}$, but such a function is not a fair composition function since $\hat{s}_1 \models \varphi_{F^2}(\mathcal{B})$ (recall that $F^2 = a_2 \geq b_2$). In this case Pdsc identifies $\hat{s}_1$ as an abstract state that must be unreachable, and extends *Unreach* with it. Additionally, as detailed in Line 16 in the algorithm, while increasing *Unreach*, Pdsc attempts to gain additional constraints by "backward traversal" over the counterexample trace. In our example, the result is extending E with the constraint $(\hat{s}_0, \{1, 2\})$.

After one additional iteration, in which another composition function is eliminated, the composition function depicted in Figure 4.1 is chosen. For this composition function `Abs_Reach` succeeds and the verification task is complete.

Chapter 5

Evaluation

In this chapter we present our implementation of Pdsc and its evaluation.

5.1 Implementation

We implemented Pdsc (Algorithm 1) in Python on top of Z3 [12]. The input is a C program encoded (by SeaHorn [20]) as a transition system using Constrained Horn Clauses (CHC) in SMT2 format, a k-safety property and a set of predicates. The implementation encodes the abstraction implicitly using the approach of [6], where the encoding is parameterized by a composition function that is modified in each iteration. For reachability checks (`Abs_Reach`) we use Spacer [23], which is implemented in Z3 and supports LIA and arrays. If an abstract counterexample is obtained with the lock-step composition function, our implementation of Pdsc runs Bounded Model Checking and may hence sometimes provide a concrete counterexample trace for unsafe programs. For the set of predicates used by Pdsc, we implemented an automatic procedure that mines these predicates from the CHC (see Section 5.2.2). Additional predicates may be added manually. We elaborate next.

5.1.1 Constrained Horn Clauses

We encode (k-)safety verification problems as Constrained Horn Clauses. Given a language $\mathcal{L}$ and a background theory $\mathcal{T}$ that interprets formulas over $\mathcal{L}$, a Constrained Horn Clause (CHC) is a first order formula of the form:

$$\forall \mathcal{U}.\, \underbrace{\phi \wedge p_1(X_1) \ldots \wedge p_n(X_n)}_{\text{body}} \rightarrow \underbrace{h(X)}_{\text{head}}$$

where:

- $\mathcal{U}$ is a set of variables.
- $X_1, \ldots, X_n, X$ are terms in $\mathcal{L}$ over $\mathcal{U}$.
- $p_1, \ldots, p_n$ are predicate symbols that do not appear in $\mathcal{L}$.
- ϕ is a constraint formula in $\mathcal{L}$ over $\mathcal{U}$.
- h is either a predicate symbol that does not appear in $\mathcal{L}$ or a formula in $\mathcal{L}$ over $\mathcal{U}$.

The left hand side of the implication in a CHC is called the *body* of the CHC, while the right hand side is called the *head*. The predicate symbols $p_1, \ldots, p_n$ (and possibly h) in a CHC are uninterpreted ("unknown").

Examples of CHCs are

$$\forall x, y, z.\ q(y) \wedge r(z) \wedge \varphi(x, y, z) \rightarrow p(x, y) \tag{5.1}$$

$$\forall x, y, z.\ q(y) \wedge r(z) \wedge \varphi(x, y, z) \rightarrow \psi(z, x) \tag{5.2}$$

where p, q, r are uninterpreted predicate symbols applied to variables x, y, z, and φ, ψ are formulas in the language $\mathcal{L}$. The body of Equation (5.1) is $q(y) \wedge r(z) \wedge \varphi(x, y, z)$ and its head is $p(x, y)$, while the body of Equation (5.2) is $q(y) \wedge r(z) \wedge \varphi(x, y, z)$ and its head is $\psi(z, x)$.

Given a set of CHCs, a CHC solver (SPACER in our implementation) attempts to decide whether the set of CHCs is satisfiable. More specifically, it checks whether there exists an interpretation (a model) for the uninterpreted predicates that satisfies the CHCs and is expressible via formulas in $\mathcal{L}$.

As an example, we show how safety of a self-composed program is encoded by a system of CHCs. We note that this is *not* the system of CHCs that is used by PDSC. The CHCs that encode safety of a self-composed program are defined over (two copies of) the set of variables of the k-self-composed program – $\mathcal{V}^{\|k} \uplus \mathcal{V}^{\|k'}$. For the encoding, we define an uninterpreted predicate *Inv* that represents an inductive invariant (an overapproximation of the reachable states) of the self-composed program. In order to encode the self-composed program we use the composition function represented via formulas C_M for every $\emptyset \neq M \subseteq \{1, \ldots, k\}$, as defined in Section 3.1. The complete encoding is presented in the following definition.

Definition 22. *(CHC system of a self-composed program) Given a transition system $T = (R, F)$, a k-safety property $(pre, post)$ as defined in Section 2.3 and a composition function f defined via conditions C_M for every $\emptyset \neq M \subseteq \{1, \ldots, k\}$, the encoding of the safety problem of the self-composed program T^f consists of the following CHCs:*

1. $\forall \mathcal{V}^{\|k}.\ pre \rightarrow Inv(\mathcal{V}^{\|k})$.
2. *for each* $\emptyset \neq M \subseteq \{1 \ldots k\}$:
   $\forall \mathcal{V}^{\|k}\ \forall \mathcal{V}^{\|k'}.\ Inv(\mathcal{V}^{\|k}) \wedge C_M \wedge \varphi_M \rightarrow Inv(\mathcal{V}^{\|k'})$.

3. $\forall \mathcal{V}^{\|k}.\ Inv(\mathcal{V}^{\|k}) \wedge F^{\|k} \wedge \neg post \rightarrow False.$

where Inv is an unknown predicate symbol and φ_M *is as defined in Definition 3.*

The presented CHCs encode safety of the transition system T^f in the following sense. If the system is satisfiable by some interpretation of *Inv* then T^f is safe (and the interpretation of *Inv* is an inductive invariant for it). If the system is not satisfiable, then a derivation of False exists, which corresponds to a counterexample trace in T^f.

In our examples, the language used to define the transition system, the safety property and the composition function is QFLIA and/or the theory of arrays. In this case, solving the corresponding system of CHCs is undecidable. Hence, as explained in Chapter 4, PDSC does not attempt to solve this system. Instead, PDSC uses predicate abstraction in order to ensure progress while iterating over different composition functions (see Chapter 4).

5.1.2 Implementing `Abs_Reach` via Implicit Predicate Abstraction

In each iteration, PDSC uses `Abs_Reach` to perform a reachability check over the abstract (composed) system $A_{\mathcal{P}}(T^f)$ of some self composition function f via predicate abstraction. To this end, given a set of predicates, `Abs_Reach` encodes the reachability check over $A_{\mathcal{P}}(T^f)$ as a CHC system. The encoding is inspired by the approach of *implicit predicate abstraction* presented in [6]. Importantly, it avoids the explicit construction of the abstract transition system, and is therefore convenient for efficiently encoding (and re-using the encoding of) an abstract self-composed program.

The encoding of implicit abstraction with respect to a set of predicates $\mathcal{P}$ uses an *abstraction relation* $H_{\mathcal{P}}$, which pairs together states of the self-composed program and their corresponding abstract states:

$$H_{\mathcal{P}} = \{(s^{\|}, \hat{s}) \mid s^{\|} \in \gamma(\hat{s})\}$$

The abstraction relation is expressed via the following formula over the variables $\mathcal{V}^{\|k}$ of the self-composed program and the Boolean variables $\mathcal{B} = \{b_p \mid p \in \mathcal{P}\}$:

$$H_{\mathcal{P}} = \bigwedge_{p \in \mathcal{P}} b_p \leftrightarrow p$$

Recall that the transition relation of the abstract self-composed program $A_{\mathcal{P}}(T^f)$ is defined by the following formula over $\mathcal{B} \uplus \mathcal{B}'$:

$$\hat{R} \ = \ \exists \mathcal{V}^{\|k}\ \exists \mathcal{V}^{\|k'}.\ \bigwedge_{p \in \mathcal{P}} (b_p \leftrightarrow p) \wedge \big(\bigvee_{M} C_M \wedge \varphi_M\big) \wedge \bigwedge_{p \in \mathcal{P}} (b'_p \leftrightarrow p')$$

Hence, when using the abstraction relation $H_{\mathcal{P}}$, we get that

$$\begin{aligned}\hat{R} &= \exists \mathcal{V}^{\|k}\, \exists \mathcal{V}^{\|k'}.\ H_{\mathcal{P}} \land \big(\bigvee_M C_M \land \varphi_M\big) \land H'_{\mathcal{P}} \\ &\equiv \bigvee_M \exists \mathcal{V}^{\|k}\, \exists \mathcal{V}^{\|k'}.\ H_{\mathcal{P}} \land C_M \land \varphi_M \land H'_{\mathcal{P}}\end{aligned}$$

where $H'_{\mathcal{P}}$ denotes the result of substituting each variable in $\mathcal{V}^{\|k}$ or $\mathcal{B}$ with its primed counterpart. Furthermore, since $C_M \in \mathcal{L}_{\mathcal{P}}$, we get that

$$\hat{R} \equiv \bigvee_M \exists \mathcal{V}^{\|k}\, \exists \mathcal{V}^{\|k'}.\ H_{\mathcal{P}} \land C_M(\mathcal{B}) \land \varphi_M \land H'_{\mathcal{P}}$$

Proof sketch. For each M, the formula C_M is a Boolean combination of predicates in $\mathcal{P}$. Therefore, by induction over the Boolean structure of C_M, due to the correspondence between each $p \in \mathcal{P}$ and the corresponding $b_p \in \mathcal{B}$ that is enforced by $H_{\mathcal{P}}$, we get that $H_{\mathcal{P}} \land C_M$ is satisfied by a model (assignments to $\mathcal{B}$ and $\mathcal{V}^{\|k}$) if and only if the same model satisfies $H_{\mathcal{P}} \land C_M(\mathcal{B})$. Hence, $H_{\mathcal{P}} \land C_M \equiv H_{\mathcal{P}} \land C_M(\mathcal{B})$. The rest of the formulas are identical. □

Next we define the CHC system that encodes condition **S1** from Lemma 15 via implicit predicate abstraction. This condition corresponds to the safety problem of an abstract self-composed program. In this encoding, *Inv* is an unknown predicate defined over the Boolean variables $\mathcal{B}$ instead of the concrete variables of the composed program.

Definition 23. *(CHC system of an abstract self-composed program) Given a transition system* $T = (R, F)$*, a k-safety property* $(pre, post)$ *as defined in Section 2.3, a composition function f defined via conditions* C_M *for every* $\emptyset \neq M \subseteq \{1, \dots, k\}$*, and a set of predicates* $\mathcal{P}$ *that is adequate for T and* $(pre, post)$ *with a corresponding set of Boolean variables* $\mathcal{B}$*, the encoding of the safety problem of the abstract self-composed program* $A_{\mathcal{P}}(T^f)$ *consists of the following CHCs:*

1. $\forall \mathcal{B}.\, \varphi_{pre}(\mathcal{B}) \rightarrow Inv(\mathcal{B})$.
2. *for each* $\emptyset \neq M \subseteq \{1 \dots k\}$*:*
   $\forall \mathcal{B}\, \forall \mathcal{B}'\, \forall \mathcal{V}^{\|k}\, \forall \mathcal{V}^{\|k'}.\, Inv(\mathcal{B}) \land H_{\mathcal{P}} \land C_M(\mathcal{B}) \land \varphi_M \land H'_{\mathcal{P}} \rightarrow Inv(\mathcal{B}')$
3. $\forall \mathcal{B}.\, Inv(\mathcal{B}) \land \varphi_{F^{\|k}}(\mathcal{B}) \land \neg\varphi_{post}(\mathcal{B}) \rightarrow False.$

where Inv is an unknown predicate symbol and φ_M *is as defined in Definition 3.*

The resulting CHC system represents safety of the abstract transition system $A_{\mathcal{P}}(T^f)$ in the sense that if the CHC system is satisfiable, the interpretation of *Inv* is an inductive invariant for $A_{\mathcal{P}}(T^f)$. If the CHC system is not satisfiable, a derivation of False induces a trace which is not necessarily a continuous sequence of transitions according to R^f, but rather a sequence of possibly disconnected transitions, where every gap between two transitions is forced to lay in some abstract state (see Figure 5.1). This makes it a trace of $\hat{R}$ (the transition relation of $A_{\mathcal{P}}(T^f)$), i.e., an abstract counterexample trace.

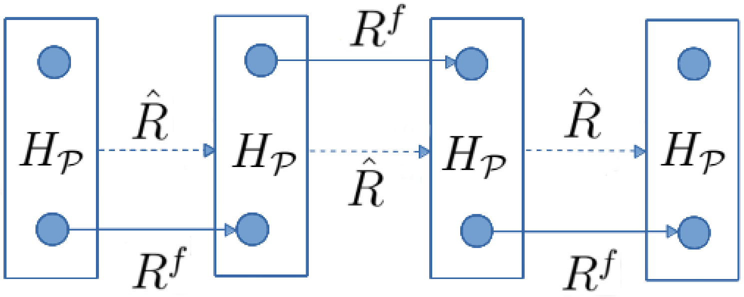

Figure 5.1: Abstract trace in an abstract program.

Finally, we present the CHC system that is constructed by PDSC for the abstract reachability check `Abs_Reach`. The system is identical to the CHC system from Definition 23, except that the safety condition (Item 3) is strengthened and now considers *Unreach*, the set of abstract states that were identified as unreachable, in addition to $\neg\varphi_{post}(\mathcal{B})$. The resulting CHC system consists of the following CHCs:

1. $\forall \mathcal{B}.\, \varphi_{pre}(\mathcal{B}) \rightarrow \mathit{Inv}(\mathcal{B})$.
2. for each $\emptyset \neq M \subseteq \{1 \dots k\}$:
   $\forall \mathcal{B}\, \forall \mathcal{B}'\, \forall \mathcal{V}^{\| k}\, \forall \mathcal{V}^{\| k'}.\, \mathit{Inv}(\mathcal{B}) \wedge H_{\mathcal{P}} \wedge C_M(\mathcal{B}) \wedge \varphi_M \wedge H'_{\mathcal{P}} \rightarrow \mathit{Inv}(\mathcal{B}')$.
3. $\forall \mathcal{B}.\, \mathit{Inv}(\mathcal{B}) \wedge \big((\varphi_{F^{\| k}}(\mathcal{B}) \wedge \neg\varphi_{post}(\mathcal{B})) \vee \mathit{Unreach}\big) \rightarrow \text{False}$.

The following lemma, which summarizes the correctness of the implementation of `Abs_Reach` via the CHC encoding described above, is now straightforward:

Lemma 24. *Let $A_{\mathcal{P}}(T^f)$ be an abstract self-composed program and $(pre, post)$ a k-safety property. Conditions* **S1, S2** *hold for $A_{\mathcal{P}}(T^f)$ and $(pre, post)$ if and only if the CHC system that implements* `Abs_Reach` *for $A_{\mathcal{P}}(T^f)$ and $(pre, post)$ is satisfiable. Furthermore, if the CHC system is satisfiable, then the satisfying interpretation of Inv is an over-approximation of the abstract states of $A_{\mathcal{P}}(T^f)$ reachable from $\varphi_{pre}(\mathcal{B})$; if the CHC is unsatisfiable then the corresponding derivation of False induces an abstract counterexample trace.*

Re-using the CHC encoding. In each iteration of the algorithm a single check of `Abs_Reach` is performed. Note that the formulas that may change between iterations are *Unreach* and $C_M(\mathcal{B})$. When a state is identified as unreachable it is added to the CHC in Item 3, as part of *Unreach*. Applying a new candidate composition function to these CHCs only requires updating the values of the formulas $C_M(\mathcal{B})$. Therefore, except for $C_M(\mathcal{B})$ and

Unreach, all formulas that construct the CHCs for `Abs_Reach` remain unchanged and are reused in every iteration of PDSC.

5.2 Experiments

To evaluate PDSC we compare it to an existing tool, SYNONYM [26], the current state of the art in k-safety property verification. Our evaluation consists of two parts: examples that require nontrivial composition functions (Section 5.2.1), and examples from previous works (Section 5.2.2).

5.2.1 Nontrivial composition functions

To show the effectiveness of PDSC, we consider examples that require a *nontrivial* composition. We emphasize that the motivation for these examples is originated in real-life scenarios. For example, Figure 1.1 follows a pattern of constant-time execution. The results of these experiments are summarized in Table 5.1. For PDSC, each row in the table displays its number of iterations (where each iteration corresponds to a call to SPACER for checking `Abs_Reach` with a candidate self composition function), the running time and the number of predicates used. PDSC is able to find a suitable composition function and verify 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 function that depends on the programs' state, rather than just program locations.

Program	PDSC			SYNONYM
	Iterations	Time (s)	Predicate count	
ArrayInsert	102	19.5	16	fail
SquaresSum	4	2.8	9	fail
DoubleSquare	33	7	20	fail
HalfSquare	28	3.4	13	fail
ArrayIntMod	168	58.2	20	fail

Table 5.1: Examples that require semantic composition functions

In the following we present the programs and their k-safety properties. For each program we present the predicate language used for verification and include the composition function that is inferred by PDSC as a part of a composition-invariant pair over the provided predicate language.

ArrayInsert

The program with a detailed explanation of its proof using a composition-invariant pair are presented in Chapter 1.

SquaresSum

The program is discussed in Section 4.6, where it is used to demonstrate a run of PDSC.

DoubleSquare

```
pre($x_1 == x_2$)

doubleSquare(bool h, int x){
  int z, y=0;
  if(h) { z = 2*x; }
  else { z = x; }
  while (z>0) {
        z--;
        y = y+x;
  }
  if(!h) { y = 2*y; }
  return y;
}

post($y_1 == y_2$)
```

```
predicates:
  $h_1$, $h_2$, $x_1 > 0$, $y_1 \geq 0$, $y_2 \geq 0$, $z_2 \geq 0$,
  $z_2 \geq 0$, $x_1 = x_2$, $y_1 = y_2$, $y_1 = 2y_2$, $y_2 = 2y_1$,
  $z_1 = z_2$, $z_1 = 2z_2$, $z_2 = 2z_1$, $z_1 = 2z_2 - 1$,
  $z_2 = 2z_1 - 1$, $y_1 = 2y_2 + x_2$, $y_2 = 2y_1 + x_1$


composition function =
if((($z0 > 0$ & $z_2 > 0$ | ($z_1 \leq 0$ & $z_2 \leq 0$))
  && ($h_1$ & $z_1 == 2 * z_2$)
  && !($h_1 == h_2$ || ($z_1 == 0$ & $z_2 == 0$)))
    || (!($z_1 > 0$ & $z_2 > 0$ | ($z_1 \leq 0$ & $z_2 \leq 0$))
      & $z_2 \leq 0$ & $z_1 > 0$))
    step (1);
else if ((($z_1 > 0$ & $z_2 > 0$ | ($z_1 \leq 0$ & $z_2 \leq 0$))
        && !($h_1 == h_2$ | ($z_1 == 0$ & $z_2 == 0$))
        && !($h_1$ & $z_1 == 2 * z_2$) & ($z_2 == 2 * z_1$))
          || !($z_1 > 0$ & $z_2 > 0$ | ($z_1 \leq 0$ & $z_2 \leq 0$)))
        step (2);
else
    step(1,2);
```

Figure 5.2: The program that computes $2x^2$ from Figure 3.1 with the composition function found by PDSC.

Figure 5.2 re-displays the example from Figure 3.1 for which no proof in QFLIA exists when the modular product program presented in [15] is considered (see Section 3.2.1). This is a non-interference problem (a 2-safety problem) where x is the low input and h is the high input. Taint analysis methods fail to prove non-interference for this program. However, using the language of predicates presented (also in Figure 5.2), PDSC infers a composition-invariant pair that proves non-interference for the program.

HalfSquare

```
pre($low_1 == low_2$)

halfSquare(int h, int low){
  assume(low > h > 0);
  int i = 0, y = 0, v = 0
  while (h > i) {
        i++; y += y;
  }
  v = 1;
  while (low > i) {
        i++; y += y;
  }
  return y;
}

post($y_1 == y_2$)
```

```
predicates:
  $h_1 > 0$, $h_2 > 0$ , $low_1 > h_1$,
  $low_2 > h_2$, $i_1 < h_1$ ,$i_2 < h_2$,
  $i_1 < low_1$, $i_2 < low_2$, $v_1 = 1$,
  $v_2 = 1$, $y_1 = y_2$, $i_1 = i_2$,
  $low_1 = low_2$


composition function=
  if (($v_1 == 0$ && $i_1 \geq h_1$)
        & ($i_2 < h_2$ || $v_2 == 1$))
      step(1);
  else if (($v_2 \neq 1$ && $i_2 \geq h_2$)
          & ($i_1 < h_1$ || $v_1 == 1$))
      step(2);
  else
      step(1,2);
```

Figure 5.3: A program that computes $\frac{low^2}{2}$; the computation is not continuous and depends on a secret variable h.

In the program presented in Figure 5.3 we consider the non-interference property, with pre-condition $low_1 = low_2$ (low input) and post-condition $y_1 = y_2$ (low output). The high input h has no constraints, as implied by the pre-condition. Intuitively, the difficulty of proving non-interference for this program arises from the need to align the computations such that $y_1 = y_2$ at every state along the execution. This is not a trivial alignment since it must "skip" the statement between the two loops. The inferred composition function aligns the computations such that they proceed simultaneously only when both are at either loops, which results in the invariant $i_1 = i_2 \wedge y_1 = y_2$ for the self composed program.

ArrayIntMod

The example in Figure 5.4 is a comparator program, based on a Java program from the comparator evaluation examples (Section 5.2.2). The comparator is modified to contain a loop that may perform two steps in a single iteration, depending on the first value in the first array (the condition is saved to $flag$). The 2-safety property of interest is anti-symmetry, i.e., the pre-condition is $o1_1 = o2_2 \wedge o1_2 = o2_1$ and the post-condition is $sgn(compare(o1_1, o2_1)) = -sgn(compare(o1_2, o2_2))$. The figure also contains the corresponding predicate language and the composition function inferred by PDSC that aligns the loops according to the value of $flag$. This yields a composed program that has an invariant that proves the desired property.

```
pre($o1_1 == o2_2$ && $o2_1 == o1_2$)

int compare(AInt o1, AInt o2){
     if(o1.len != o2.len){
        return 0;
     }
     boolean flag = (o1.get(0)>0);
     int i, aentry, bentry,last1,last2;
     i = 0;

     while ((i < o1.len) && (i<o2.len)) {
       aentry = o1.get(i);
       bentry = o2.get(i);
       if (aentry < bentry) {
           return -1;
       }
       if (aentry > bentry) {
           return 1;
       }
       i++;

     if(flag && ((i < o1.len) && (i < o2.len))){
          aentry = o1.get(i);
          bentry = o2.get(i);
          if (aentry < bentry) {
           return -1;
            }
            if (aentry > bentry) {
           return 1;
            }
          i++;
       }

     }
     return 0;
   }
}

post(sgn(compare($o1_1$,$o2_1$)) = -sgn(compare($o1_2$,$o2_2$))
```

```
predicates:
  $len1_1 = len2_1$, $len1_1 = len2_2$,
  $o1_1 = o2_2$, $o2_1 = o1_2$,
  $len1_1 = len2_2$, $len2_1 = len1_2$,
  $ret_1 = -ret_2$, $i_1 < len1_1$,
  $i_2 < len1_2$, $i_1 = i_2$,
  $i_1 = i_2 - 1$, $i_2 = i_1 - 1$,
  $o1_1[i_1] = o2_1[i_1]$,
  $o1_2[i_2] = o2_2[i_2]$,
  $o1_1[i_1 + 1] = o2_1[i_1 + 1]$,
  $o1_2[i_2 + 1] = o2_2[i_2 + 1]$
```

```
composition function=
  if (($i_2 = i_1 + 1$ && $i_2 < len_2$) || ($i_2 = i_1$ && $i_2 < len_2$ && $o1_2[i_2] = o2_2[i_2]$ &&
        $o1_2[i_2 + 1] \neq o2_2[i_2 + 1]$ && $flag_2$ && !$flag_1$)
      step(1);
  else if (($i_1 = i_2 + 1$ && $i_1 < len_1$) || ($i_1 = i_2$ && $i_1 < len_1$ && $o1_1[i_1] = o2_1[i_1]$ &&
             $o1_1[i_1 + 1] \neq o2_1[i_1 + 1]$ && $flag_1$ !$flag_2$))
      step(2);
  else
      step(1,2);
```

Figure 5.4: Comparator example with potentially unbalanced loops.

5.2.2 Comparator examples

Next we compare Pdsc to Synonym on programs that were considered in previous work, and show that its performance on such programs is comparable to Synonym. To this end we consider 34 Java comparator programs from [26, 29] that are based on real programs that appeared on Stackoverflow. For each program we check the 3 properties that are required from a method named **compare**:

P1 $\forall x, y.\ sgn(compare(x, y)) = -sgn(compare(y, x))$ (Anti-symmetry)
P2 $\forall x, y, z.\ \big(compare(x, y) > 0 \land compare(y, z) > 0\big) \rightarrow compare(x, z) > 0$ (Transitivity)
P3 $\forall x, y, z.\ compare(x, y) = 0 \rightarrow \big(sgn(compare(x, z)) = sgn(compare(y, z))\big)$

Therefore, a total of 102 verification problems are considered, where each problem is a pair of a program and a property. The verification problems (and accordingly the results) are divided to 63 safe problems – in which the program satisfies the property, and 39 unsafe problems – in which the program violates the property. For unsafe problems, when Synonym converges it returns a concrete counterexample trace, whereas Pdsc may either find a concrete counterexample or only determine that no composition-invariant pair exists in the given predicate language. To run Pdsc, we manually converted the Java programs to C, and implemented a pre-processing procedure that automatically converts the C programs to SMT2, using SeaHorn, and mines the predicates used by Pdsc.

Predicate Mining For all but 3 programs (out of 34), only 2 types of predicates that we mined automatically were sufficient for verification of the safe instances:

1. Relational predicates derived from the pre- and post-conditions. E.g., for anti-symmetry, a predicate for each equality expression in the property.
2. 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 (the evaluated programs have a single function, therefore cut-points are only loop heads). For the remaining 3 programs, we manually added 2–4 predicates.

Results Figure 5.5 plots the running times of Pdsc and Synonym over all verification problems (safe and unsafe). The results are also detailed in Tables A.1 to A.3.

With the exception of 3 problems, all the safe problems were solved by Pdsc with a lock-step composition function. The problems that were not solved with lock-step composition are all instances of the same program. 2 out of the 3 instances timed out while the third was successfully solved with a different composition. Overall, the results show that on safe

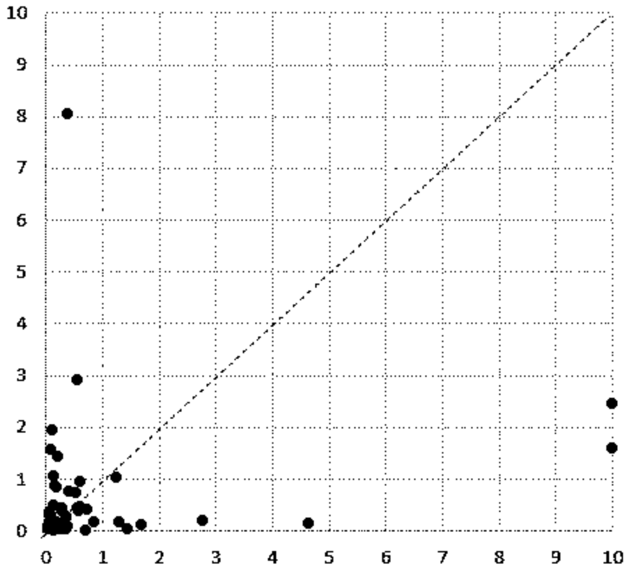

Figure 5.5: Runtime comparison: Pdsc (x-axis) and Synonym (y-axis).

problems where a simple composition function (lock-step) suffices, Pdsc performs similarly to Synonym.

For 12 out of 39 unsafe problems Pdsc only determines that no composition-invariant pair exists, and does not provide a concrete counterexample. However, it does terminate within time similar to Synonym for unsafe problems as well.

Chapter 6

Related work

This thesis 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, 32]). Below we focus on works that use self-composition.

Self-Composition. Previous work such as [4, 2, 3, 16, 31, 15] 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.

Cartesian Hoare Logic. The work most closely related to ours is [29] 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 [29] 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.

Modular Product Program. In [15], modular k-product programs are introduced in order to enable modular proofs of arbitrary k-safety properties. These programs use Boolean activation variables that indicate at every statement which of the copies of the duplicated program should perform the statement. While adding activation variables does not result in a composition function that is pre-defined completely, the activation variables are added statically based on the control flow structure of the program, which is similar in spirit to syntactic composition. This is in contrast to our approach, which takes advantage of a fully dynamic composition. Yet, [15] leverages the activation variable representation to define modular proofs, which we do not attempt in this work.

Synchronized Constrained Horn Clauses. There have been several results in the context of harnessing Constraint Horn Clauses (CHC) solvers for verification of relational properties [11, 25]. 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. Checking equivalence of programs 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 [33, 28, 10, 19]. 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, 19] identify possible synchronization points syntactically, and try to match them during the construction of a simulation relation between programs.

Regression verification. The problem of regression verification also requires the ability to show equivalence between different versions of a program [16, 17, 30]. More precisely, given two programs and a mapping between their functions, regression verification aims to verify the equivalence of the two programs in terms of the computations results. In [30], regression verification in the presence of unbalanced recursive function calls is addressed. To allow synchronization of recursive calls, the user can specify different unrolling parameters for the different copies. Synchronization of the recursive calls in this case resembles synchronization of unbalanced loops that our work addresses. In contrast to the method presented in [30], our approach relies only on user supplied predicates that are needed to establish correctness, while synchronization is handled automatically.

Chapter 7

Conclusion and Future Work

This work formulates the problem of inferring a semantic self composition function together with an inductive invariant for the composed program in a given language. Considering the composition function and the inductive invariant together, where both are restricted to a given language, captures the interplay between the self composition and the difficulty of verifying the resulting composed program, as reflected by the expressive power needed to express an inductive invariant for the composed program. To address the inference problem we present PDSC– a *property directed* algorithm for inferring a composition-invariant pair in a given language of predicates. We implement PDSC and show that it manages to find nontrivial self compositions that are beyond reach of existing tools. When evaluated on programs that require only a trivial composition, PDSC is comparable to existing tools.

In future work, we are interested in further improving PDSC by extending it with more sophisticated (possibly lazy) predicate discovery schemes. We believe that the need for a user to specify the predicate language used by the tool is a major hurdle for the usability of PDSC for large and complex programs. Our current procedure for automatic predicate discovery is rather naive. Automatically mining a greater range of predicates has the potential to both improve the performance of PDSC and verify a wider range of programs. Another promising direction is to consider an iterative procedure in which predicate discovery is intertwined with the inference procedure in a lazy fashion. In addition, we consider improving the performance of PDSC by exploring further generalization techniques as described in Section 4.5.2.

Another direction we wish to explore is embedding the inference of a semantic composition function within other efficient k-safety verification methods (such as methods based on Cartesian Hoare Logic [26, 29]).

Bibliography

[1] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 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*, pages 362–375, 2017.

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, pages 200–214, 2011.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*, pages 29–43, 2013.

[4] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114, 2004.

[5] Aaron R. Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87, 2011.

[6] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 modulo theories via implicit predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 46–61, 2014.

[7] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[8] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 51–65, 2008.

[9] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[10] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*, pages 127–147, 2017.

[11] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through horn clause transformation. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 147–169, 2016.

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[13] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[14] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134, 2011.

[15] Marco Eilers, Peter Müller, and Samuel Hitz. Modular product programs. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 502–529, 2018.

[16] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 349–360, 2014.

[17] Benny Godlin and Ofer Strichman. Regression verification. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pages 466–471, 2009.

[18] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

[19] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 365–382, 2018.

[20] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[22] Jaber Karimpour, Ayaz Isazadeh, and Ali A. Noroozi. Verifying observational determinism. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, pages 82–93, 2015.

[23] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 17–34, 2014.

[24] Shuvendu K. Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, pages 214–229, 2009.

[25] Dmitry Mordvinov and Grigory Fedyukovich. Synchronizing constrained horn clauses. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pages 338–355, 2017.

[26] Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. Exploiting synchrony and symmetry in relational verification. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 164–182, 2018.

[27] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pages 443–454, 1999.

[28] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 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*, pages 391–406, 2013.

[29] Marcelo Sousa and Isil Dillig. 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*, pages 57–69, 2016.

[30] Ofer Strichman and Maor Veitsman. Regression verification for unbalanced recursive functions. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 645–658, 2016.

[31] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, pages 352–367, 2005.

[32] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. Lazy self-composition for security verification. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 136–156, 2018.

[33] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 35–51, 2008.

Appendix A

Running time tables for evaluated comparator programs

Tables A.1 to A.3 present the performance of predicate mining (performed before running Pdsc), as well as the running time of Pdsc itself and the running time of Synonym on the Java comparator programs and the 3 properties listed in Section 5.2.2. The columns "Iterations" and "Predicate count" are the number of calls to Spacer (for checking `Abs_Reach` with a candidate self composition function) and the number of predicates used, respectively. We split the verification time of each problem to "Time" and "Predicate mining". The former refers to the actual running time of Pdsc, while the latter is the time spent on generating a predicate set to run Pdsc with. The values under "Manually supplied predicate" are the number of predicates out of the total "Predicate count" that were supplied manually, and not mined automatically. We denote timeout (10 seconds) with TO.

Program	**Pdsc**					**Synonym** (s)
	Iterations	Time (s)	Predicate mining (s)	Predicate count	Manually supplied predicates	
Safe						
ArrayInt-false	1	0.102801	0.386401	8	0	0.051
ArrayInt-true	1	0.109133	0.41706	8	0	0.05
Chromosome-false	1	0.361742	1.581703	7	0	0.063
Chromosome-true	1	0.087003	0.399591	8	0	0.038
ColItem-true	1	0.062242	0.210289	9	0	0.033
Contact-false	1	0.08527	0.275478	7	0	0.071
Container-true	1	0.063571	0.193904	9	0	0.062
FileItem-false	1	0.044894	0.112472	5	0	0.021
FileItem-true	1	0.04735	0.139799	5	0	0.029
Match-true	1	0.052631	0.152487	7	0	0.023
NameComparator-true	1	0.170584	0.828861	8	0	0.044
Node-false	1	0.074997	0.378679	11	0	0.021
Node-true	1	0.09925	0.454168	11	0	0.023
NzbFile-true	1	0.18299	1.570949	14	4	0.058
PokerHand-false	1	0.842343	5.064878	23	4	0.175
PokerHand-true	5	2.7699	6.94377	18	0	0.188
Solution-false	1	0.078602	0.26656	7	0	0.082
Solution-true	1	0.0635	0.243874	7	0	0.162
TextPosition-false	1	0.065248	0.218703	9	0	0.051
TextPosition-true	1	0.071053	0.2486	11	0	0.067
Time-true	1	0.041927	0.090919	5	0	0.047
Word-true	1	0.102203	0.558408	10	0	0.045
Unsafe						
CatBPos-false	6	0.350095	0.554562	11	0	0.04
ColItem-false	6	0.297965	0.428963	9	0	0.041
Container-false-v1	6	0.279431	0.386224	9	0	0.024
Container-false-v2	6	0.287247	0.406656	9	0	0.025
DataPoint-false	6	0.294873	0.447905	7	0	0.059
IsoSprite-false-v1	7	0.215831	0.286406	1	0	0.026
IsoSprite-false-v2	6	0.348504	0.593785	13	0	0.252
Match-false	6	0.269434	0.379308	7	0	0.024
NameComparator-false	20	1.444409	1.629696	6	0	0.039
NzbFile-false	6	0.392863	1.10813	9	0	0.076
Time-false	6	0.217103	0.268009	5	0	0.04
Word-false	36	4.647284	5.248239	11	0	0.142

Table A.1: Running results for comparator property P1 - Antisymmetry.

Program	**PDSC**					**SYNONYM** (s)
	Iterations	Time (s)	Predicate mining (s)	Predicate count	Manually supplied predicates	
Safe						
ArrayInt-false	1	0.361299	0.946831	14	0	0.286
ArrayInt-true	1	0.405376	1.026988	14	0	0.757
Chromosome-true	1	0.391074	1.033354	14	0	8.058
ColItem-true	1	0.092281	0.424563	15	0	0.152
Container-true	1	0.091462	0.388688	15	0	1.577
FileItem-false	1	0.064125	0.219827	9	0	0.037
FileItem-true	1	0.085879	0.304044	9	0	0.044
Match-false	1	0.077703	0.326495	12	0	0.04
Match-true	1	0.072908	0.300427	12	0	0.053
NameComparator-false	1	0.279183	0.680973	12	0	0.387
NameComparator-true	1	0.734568	1.450714	15	0	0.413
Node-false	1	0.119951	0.707396	18	0	0.03
Node-true	1	0.145707	0.851567	18	0	0.051
NzbFile-false	1	0.186473	1.608238	15	0	0.08
NzbFile-true	1	0.523781	1.963365	23	4	0.746
PokerHand-true	TO	TO	TO	29	4	1.596
Solution-false	1	0.134059	0.619103	13	0	0.497
Solution-true	1	0.132646	0.547795	13	0	1.048
TextPosition-true	1	0.113323	0.515613	18	0	1.95
Time-false	1	0.058409	0.173289	9	0	0.359
Time-true	1	0.057856	0.169371	9	0	0.305
Word-true	1	0.282044	1.208389	17	0	0.448
Unsafe						
CatBPos-false	2	0.100266	0.575184	20	0	0.04
Chromosome-false	2	0.300962	2.676827	12	0	0.087
ColItem-false	2	0.076132	0.371777	15	0	0.022
Contact-false	2	0.084389	0.517546	12	0	0.04
Container-false-v1	2	0.069804	0.309855	15	0	0.021
Container-false-v2	2	0.071401	0.346139	15	0	0.021
DataPoint-false	2	0.077918	0.434505	12	0	0.09
IsoSprite-false-v1	2	0.053593	0.247729	9	0	0.023
IsoSprite-false-v2	2	0.105165	0.632595	21	0	0.025
PokerHand-false	2	1.244157	9.458459	37	0	1.018
TextPosition-false	2	0.084009	0.428122	15	0	0.071
Word-false	2	0.69956	1.966766	19	0	0.018

Table A.2: Running results for comparator property P2 - Transitivity.

Program	**Pdsc**					**Synonym** (s)
	Iterations	Time (s)	Predicate mining (s)	Predicate count	Manually supplied predicates	
Safe						
ArrayInt-true	1	0.606869	1.285321	17	0	0.956
Chromosome-true	1	0.556879	1.22019	17	0	2.91
ColItem-true	1	0.152798	0.494219	18	0	0.155
Container-false-v1	1	0.11331	0.361224	18	0	0.041
Container-true	1	0.161147	0.465932	18	0	0.873
FileItem-true	1	0.093718	0.339993	12	0	0.054
IsoSprite-false-v2	1	0.194566	0.730789	24	0	0.853
Match-true	1	0.125692	0.360273	15	0	0.059
NameComparator-false	1	0.560919	0.970041	15	0	0.436
NameComparator-true	1	0.615013	1.292094	18	0	0.462
Node-true	1	0.27561	0.98983	21	0	0.073
NzbFile-false	1	0.264231	1.707929	18	0	0.147
PokerHand-true	TO	TO	TO	32	0	2.454
Solution-true	1	0.205432	0.626999	16	0	1.421
TextPosition-true	1	0.187039	0.597422	21	0	0.179
Time-false	1	0.037231	0.159684	12	0	0.081
Time-true	1	0.085826	0.203187	12	0	0.309
Word-false	1	0.139223	1.433379	22	0	0.019
Word-true	1	0.568253	1.50883	20	0	0.377
Unsafe						
ArrayInt-false	2	0.178949	0.900308	17	0	0.145
CatBPos-false	2	0.111095	0.602127	23	0	0.04
Chromosome-false	2	0.268251	2.676413	15	0	0.094
ColItem-false	2	0.127957	0.431028	18	0	0.068
Contact-false	2	0.094413	0.537394	15	0	0.028
Container-false-v2	2	0.099635	0.381602	18	0	0.027
DataPoint-false	2	0.144215	0.507173	15	0	0.068
FileItem-false	2	0.064935	0.226448	12	0	0.024
IsoSprite-false-v1	2	0.059607	0.259633	12	0	0.028
Match-false	2	0.095143	0.353735	15	0	0.043
Node-false	2	0.133625	0.727429	21	0	0.021
NzbFile-true	2	1.291413	2.822463	26	4	0.158
PokerHand-false	2	1.692465	10.009726	40	4	0.107
Solution-false	2	0.142166	0.625398	16	0	0.133
TextPosition-false	2	0.094398	0.447467	18	0	0.106

Table A.3: Running results for comparator property P3.

תקציר

בעבודה זו אנו חוקרים את בעיית האימות של תכונות k-בטיחות: תכונות אשר מתייחסות ל-k ריצות של תכנית. גישה ידועה לאימות תכונות אלו היא גישת ההרכבה העצמית. בגישה זו בעיית האימות של k-בטיחות עבור תכנית נתונה מתורגמת לבדיקת תכונת בטיחות "רגילה" עבור תכנית המריצה (בסידור כלשהו) k עותקים של התכנית המקורית. הדרך בה העותקים הללו מורכבים לתכנית אחת קובעת למעשה כמה מסובך יהיה לאמת את התכנית המורכבת. אנו רואים הרכבה זו כפונקציית הרכבה סמנטית אשר ממפה מצב בתכנית המורכבת אל העותקים שיבצעו את הצעד הבא בריצה.

מכיוון ש"איכות" פונקציית ההרכבה העצמית נמדדת על ידי היכולת לאמת את התכנית המורכבת, אנו מגדירים את הבעיה כהסקת פונקציית ההרכבה העצמית יחד עם אינווריאנט אינדוקטיבי הנחוץ לצורך אימות התכנית המורכבת, כאשר שניהם מוגבלים לשפה נתונה. אנו מפתחים אלגוריתם הסקה מוכוון תכונה אשר בהינתן קבוצת פרדיקטים, מסיק זוגות הרכבה-אינווריאנט הניתנים לביטוי על ידי שילובים בוליאניים של הפרדיקטים הנתונים, או לחילופין קובע שלא קיים זוג כזה המאפשר את אימות התכונה. האלגוריתם מומש והתוצאות מדגימות את יכולתו למצוא הרכבות עצמיות מעבר ליכולותיהם של כלי אימות מובילים.

אוניברסיטת תל-אביב TEL AVIV UNIVERSITY

הפקולטה למדעים מדויקים ע״ש ריימונד ובברלי סאקלר

בית הספר למדעי המחשב ע״ש בלבטניק

הרכבה עצמית
מוכוונת תכונה

חיבור זה הוגש כחלק מהדרישות לקבלת התואר

"מוסמך האוניברסיטה"(M.Sc.)

על ידי

רון שמר

עבודת המחקר בוצעה בהנחייתה של

דר' שרון שוהם