Automated Hypersafety Verification

Azadeh Farzan and Anthony Vandikas

University of Toronto

Abstract. We propose an automated verification technique for hypersafety properties, which express sets of valid interrelations between multiple finite runs of a program. The key observation is that constructing a proof for a small representative set of the runs of the product program (i.e. the product of the several copies of the program by itself), called a *reduction*, is sufficient to formally prove the hypersafety property about the program. We propose an algorithm based on a counterexample-guided refinement loop that simultaneously searches for a reduction and a proof of the correctness for the reduction. We demonstrate that our tool Weaver is very effective in verifying a diverse array of hypersafety properties for a diverse class of input programs.

1 Introduction

A hypersafety property describes the set of valid interrelations between multiple finite runs of a program. A k-safety property [7] is a program safety property whose violation is witnessed by at least k finite runs of a program. Determinism is an example of such a property: non-determinism can only be witnessed by two runs of the program on the same input which produce two different outputs. This makes determinism an instance of a 2-safety property.

The vast majority of existing program verification methodologies are geared towards verifying standard (1-)safety properties. This paper proposes an approach to automatically reduce verification of k-safety to verification of 1-safety, and hence a way to leverage existing safety verification techniques for hypersafety verification. The most straightforward way to do this is via *self-composition* [5], where verification is performed on k memory-disjoint copies of the program, sequentially composed one after another. Unfortunately, the proofs in these cases are often very verbose, since the full functionality of each copy has to be captured by the proof. Moreover, when it comes to automated verification, the invariants required to verify such programs are often well beyond the capabilities of modern solvers [26] even for very simple programs and properties.

The more practical approach, which is typically used in manual or automated proofs of such properties, is to compose k memory-disjoint copies of the program *in parallel* (instead of in sequence), and then verify some *reduced* program obtained by removing redundant traces from the program formed in the previous step. This parallel product program can have many such reductions. For example, the program formed from sequential self-composition is one such reduction of the parallel product program. Therefore, care must be taken to choose a "good"

reduction that *admits a simple proof*. Many existing approaches limit themselves to a narrow class of reductions, such as the one where each copy of the program executes in lockstep [3, 10, 24], or define a general class of reductions, but do not provide algorithms with guarantees of covering the entire class [4, 24].

We propose a solution that combines the search for a safety proof with the search for an appropriate reduction, in a counterexample-based refinement loop. Instead of settling on a single reduction in advance, we try to verify the entire (possibly infinite) set of reductions simultaneously and terminate as soon as some reduction is successfully verified. If the proof is not currently strong enough to cover at least one of the represented program reductions, then an appropriate set of counterexamples are generated that guarantee progress towards a proof.

Our solution is language-theoretic. We propose a way to represent sets of reductions using infinite tree automata. The standard safety proofs are also represented using the same automata, which have the desired closure properties. This allows us to check if a candidate proof is in fact a proof for one of the represented program reductions, with reasonable efficiency.

Our approach is not uniquely applicable to hypersafety properties of sequential programs. Our proposed set of reductions naturally work well for concurrent programs, and can be viewed in the spirit of reduction-based methods such as those proposed in [11, 21]. This makes our approach particularly appealing when it comes to verification of hypersafety properties of concurrent programs, for example, proving that a concurrent program is deterministic. The parallel composition for hypersafety verification mentioned above and the parallel composition of threads inside the multi-threaded program are treated in a uniform way by our proof construction and checking algorithms. In summary:

- We present a counterexample-guided refinement loop that simultaneously searches for a proof and a program reduction in Section 7. This refinement loop relies on an efficient algorithm for proof checking based on the antichain method of [8], and strong theoretical progress guarantees.
- We propose an automata-based approach to representing a class of program reductions for k-safety verification. In Section 5 we describe the precise class of automata we use and show how their use leads to an effective proof checking algorithm incorporated in our refinement loop.
- We demonstrate the efficacy of our approach in proving hypersafety properties of sequential and concurrent benchmarks in Section 8.

2 Illustrative Example

We use a simple program Mult, that computes the product of two non-negative integers, to illustrate the challenges of verifying hypersafety properties and the type of proof that our approach targets. Consider the multiplication program in Figure 1(i), and assume we want to prove that it is distributive over addition.

In Figure 1 (ii), the parallel composition of Mult with two copies of itself is illustrated. The product program is formed for the purpose of proving distributivity, which can be encoded through the postcondition $x_1 = x_2 + x_3$. Since a, b, and

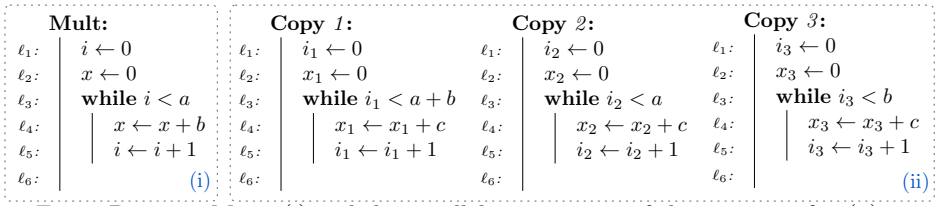

Fig. 1: Program MULT (i) and the parallel composition of three copies of it (ii).

c are not modified in the program, the same variables are used across all copies. One way to prove MULT is distributive is to come up with an inductive invariant ϕ_{ijk} for each location in the product program, represented by a triple of program locations (ℓ_i, ℓ_j, ℓ_k), such that $\mathit{true} \implies \phi_{111}$ and $\phi_{666} \implies x_1 = x_2 + x_3$. The main difficulty lies in finding assignments for locations such as ϕ_{611} that are points in the execution of the program where one thread has finished executing and the next one is starting. For example, at (ℓ_6, ℓ_1, ℓ_1) we need the assignment $\phi_{611} \leftarrow x_1 = (a+b) * c$ which is non-linear. However, the program given in Figure 1(ii) can be verified with simpler (linear) reasoning.

The program on the right is a semantically equivalent *reduction* of the full composition of Figure 1(ii). Consider the program $P = (\text{Copy 1} \mid\mid (\text{Copy 2; Copy 3}))$. The program on the right is equivalent to a lockstep execution of the two parallel components of P. The validity of this reduction is derived from the fact that the statements in each thread are *independent* of the statements in the other. That is, reordering the statements of different threads in an execution leads to an equivalent execution. It is easy to see that $x_1 = x_2 + x_3$ is an invariant of both while loops in the reduced program, and therefore, linear reasoning is sufficient to prove the postcondition for this program. Conceptually, this reduction (and its soundness proof) together with the proof of correctness for the reduced program constitute a proof that the original program MULT is distributive. Our proposed approach can come up with reductions like this and their corresponding proofs fully automatically. Note that a lockstep reduction of the program in Figure 1(ii) would not yield a solution for this problem and therefore the discovery of the right reduction is an integral part of the solution.

$i_1 \leftarrow 0,\ i_2 \leftarrow 0,\ i_3 \leftarrow 0$
$x_1 \leftarrow 0,\ x_2 \leftarrow 0,\ x_3 \leftarrow 0$
while $i_2 < a$
  $x_1 \leftarrow x_1 + c$
  $x_2 \leftarrow x_2 + c$
  $i_1 \leftarrow i_1 + 1$
  $i_2 \leftarrow i_2 + 1$
while $i_3 < b$
  $x_1 \leftarrow x_1 + c$
  $x_3 \leftarrow x_3 + c$
  $i_1 \leftarrow i_1 + 1$
  $i_3 \leftarrow i_3 + 1$

3 Programs and Proofs

A non-deterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A deterministic finite automaton (DFA) is an NFA whose transition relation is a function $\delta : Q \times \Sigma \to Q$. The language of an NFA or DFA A is denoted $\mathcal{L}(A)$, which is defined in the standard way [18].

3.1 Program Traces

$\mathcal{S}t$ denotes the (possibly infinite) set of *program states*. For example, a program with two integer variables has $\mathcal{S}t = \mathbb{Z} \times \mathbb{Z}$. $\mathcal{A} \subseteq \mathcal{S}t$ is a (possibly infinite) set of *assertions* on program states. Σ denotes a finite alphabet of program *statements*. We refer to a finite string of statements as a (program) *trace*. For each statement $a \in \Sigma$ we associate a *semantics* $[\![a]\!] \subseteq \mathcal{S}t \times \mathcal{S}t$ and extend $[\![-]\!]$ to traces via (relation) composition. A trace $x \in \Sigma^*$ is said to be *infeasible* if $[\![x]\!](\mathcal{S}t) = \emptyset$, where $[\![x]\!](\mathcal{S}t)$ denotes the image of $[\![x]\!]$ under $\mathcal{S}t$. To abstract away from a particular program syntax, we define a *program* as a regular language of traces. The semantics of a program P is simply the union of the semantics of its traces $[\![P]\!] = \bigcup_{x \in P} [\![x]\!]$. Concretely, one may obtain programs as languages by interpreting their edge-labelled control-flow graphs as DFAs: each vertex in the control flow graph is a state, and each edge in the control flow graph is a transition. The control flow graph entry location is the initial state of the DFA and all its exit locations are final states.

3.2 Safety

There are many equivalent notions of program safety; we use non-reachability. A program P is *safe* if all traces of P are infeasible, i.e. $[\![P]\!](\mathcal{S}t) = \emptyset$. Standard partial correctness specifications are then represented via a simple encoding. Given a precondition ϕ and a postcondition ψ, the validity of the Hoare-triple $\{\phi\}P\{\psi\}$ is equivalent to the safety of $[\phi] \cdot P \cdot [\neg\psi]$, where [] is a standard assume statement (or the singleton set containing it), and $\cdot$ is language concatenation.

Example 3.1. We use determinism as an example of how k-safety can be encoded in the framework defined thus far. If P is a program then determinism of P is equivalent to safety of $[\phi] \cdot (P_1 \sqcup\!\sqcup P_2) \cdot [\neg\phi]$ where P_1 and P_2 are copies of P operating on disjoint variables, $\sqcup\!\sqcup$ is a shuffle product of two languages, and $[\phi]$ is an assume statement asserting that the variables in each copy of P are equal.

A *proof* is a finite set of assertions $\Pi \subseteq \mathcal{A}$ that includes *true* and *false*. Each Π gives rise to an NFA $\Pi_{NFA} = (\Pi, \mathcal{S}t, \delta_\Pi, \mathit{true}, \{\mathit{false}\})$ where $\delta_\Pi(\phi_{pre}, a) = \{\phi_{post} \mid [\![a]\!](\phi_{pre}) \subseteq \phi_{post}\}$. We abbreviate $\mathcal{L}(\Pi_{NFA})$ as $\mathcal{L}(\Pi)$. Intuitively, $\mathcal{L}(\Pi)$ consists of all traces that can be proven infeasible using only assertions in Π. Thus the following proof rule is sound [12, 13, 17]:

$$\frac{\exists \Pi \subseteq \mathcal{A}.\, P \subseteq \mathcal{L}(\Pi)}{P \text{ is safe}} \qquad (\textsc{Safe})$$

When $P \subseteq \mathcal{L}(\Pi)$, we say that Π is a proof for P. A proof does not uniquely belong to any particular program; a single Π may prove many programs correct.

4 Reductions

The set of assertions used for a proof is usually determined by a particular language of assertions, and a safe program may not have a (safety) proof in that

particular language. Yet, a subset of the program traces may have a proof in that assertion language. If it can be proven that the subset of program runs that have a safety proof are a faithful representation of all program behaviours (with respect to a given property), then the program is correct. This motivates the notion of *program reductions*.

Definition 4.1 (semantic reduction). *If for programs P and P', P' is safe implies that P is safe, then P' is a* semantic reduction *of P (written $P' \preceq P$).*

The definition immediately gives rise to the following proof rule for proving program safety:

$$\frac{\exists P' \preceq P, \Pi \subseteq \mathcal{A}.\, P' \subseteq \mathcal{L}(\Pi)}{P \text{ is safe}} \qquad (\textsc{SafeRed}1)$$

This generic proof rule is not automatable since, given a proof Π, verifying the existence of the appropriate reduction is *undecidable*. Observe that a program is safe if and only if $\emptyset$ is a valid reduction of the program. This means that discovering a semantic reduction and proving safety are mutually reducible to each other. To have decidable premises for the proof rule, we need to formulate an easier (than proving safety) problem in discovering a reduction. One way to achieve this is by restricting the set of reductions under consideration from all reductions (given in Definition 4.1) to a proper subset which more amenable to algorithmic checking. Fixing a set $\mathcal{R}$ of (semantic) reductions, we will have the rule:

$$\frac{\exists P' \in \mathcal{R}.\, P' \subseteq \mathcal{L}(\Pi) \qquad \forall P' \in \mathcal{R}.\, P' \preceq P}{P \text{ is safe}} \qquad (\textsc{SafeRed}2)$$

Proposition 4.2. *The proof rule* SafeRed2 *is sound.*

The core contribution of this paper is that it provides an algorithmic solution inspired by the above proof rule. To achieve this, two subproblems are solved: (1) Given a set $\mathcal{R}$ of reductions of a program P and a candidate proof Π, can we check if there exists a reduction $P' \in \mathcal{R}$ which is covered by the proof Π? In section 5, we propose a new semantic interpretation of an existing notion of infinite tree automata that gives rise to an algorithmic check for this step. (2) Given a program P, is there a general sound set of reductions $\mathcal{R}$ that be effectively represented to accommodate step (1)? In section 6, we propose a construction of an effective set of reductions, representable by our infinite tree automata, using inspirations from existing partial order reduction techniques [15].

5 Proof Checking

Given a set of reductions $\mathcal{R}$ of a program P, and a candidate proof Π, we want to check if there exists a reduction $P' \in \mathcal{R}$ which is covered by Π. We call this *proof checking*. We use tree automata to represent certain classes of languages (i.e sets of sets of strings), and then use operations on these automata for the purpose of proof checking.

The set Σ^* can be represented as an infinite tree. Each $x \in \Sigma^*$ defines a path to a unique node in the tree: the root node is located at the empty string ϵ, and for all $a \in \Sigma$, the node located at xa is a child of the node located at x. Each node is then identified by the string labeling the path leading to it. A language $L \subseteq \Sigma^*$ (equivalently, $L : \Sigma^* \to \mathbb{B}$) can consequently be represented as an infinite tree where the node at each x is labelled with a boolean value $B \equiv (x \in L)$. An example is given in Figure 2.

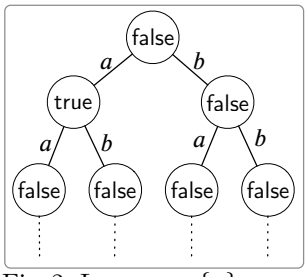

Fig. 2: Language $\{a\}$ as an infinite tree.

It follows that a set of languages is a set of infinite trees, which can be represented using automata over infinite trees. Looping Tree Automata (LTAs) are a subclass of Büchi Tree Automata where all states are accept states [2]. The class of Looping Tree Automata is closed under intersection and union, and checking emptiness of LTAs is decidable. Unlike Büchi Tree Automata, emptiness can be decided in linear time [2].

Definition 5.1. *A Looping Tree Automaton (LTA) over $|\Sigma|$-ary, $\mathbb{B}$-labelled trees is a tuple $M = (Q, \Delta, q_0)$ where Q is a finite set of states, $\Delta \subseteq Q \times \mathbb{B} \times (\Sigma \to Q)$ is the transition relation, and q_0 is the initial state.*

Intuitively, an LTA $M = (Q, \Delta, q_0)$ performs a parallel and depth-first traversal of an infinite tree L while maintaining some local state. Execution begins at the root ϵ from state q_0 and non-deterministically picks a transition $(q_0, B, \sigma) \in \Delta$ such that B matches the label at the root of the tree (i.e. $B = (\epsilon \in L)$). If no such transition exists, the tree is rejected. Otherwise, M recursively works on each child a from state $q' = \sigma(a)$ in parallel. This process continues infinitely, and L is accepted if and only if L is never rejected.

Formally, M's execution over a tree L is characterized by a *run* $\delta^* : \Sigma^* \to Q$ where $\delta^*(\epsilon) = q_0$ and $(\delta^*(x), x \in L, \lambda a.\, \delta^*(xa)) \in \Delta$ for all $x \in \Sigma^*$. The set of languages accepted by M is then defined as $\mathcal{L}(M) = \{L \mid \exists \delta^*.\, \delta^* \text{ is a run of } M \text{ on } L\}$.

Theorem 5.2. *Given an LTA M and a regular language L, it is decidable whether $\exists P \in \mathcal{L}(M).\, P \subseteq L$.*

The proof, which appears in [14], reduces the problem to deciding whether $\mathcal{L}(M) \cap \mathcal{P}(L) \neq \emptyset$. LTAs are closed under intersection and have decidable emptiness checks, and the lemma below is the last piece of the puzzle.

Lemma 5.3. *If L is a regular language, then $\mathcal{P}(L)$ is recognized by an LTA.*

Counterexamples. Theorem 5.2 effectively states that proof checking is decidable. For automated verification, beyond checking the validity of a proof, we require counterexamples to fuel the development of the proof when the proof does not check. Note that in the simple case of the proof rule SAFE, when $P \not\subseteq \mathcal{L}(\Pi)$ there exists a counterexample trace $x \in P$ such that $x \notin \mathcal{L}(\Pi)$.

With our proof rule SafeRed2, things get a bit more complicated. First, note that unlike the classic case (Safe), where a failed proof check coincides with the non-emptiness of an intersection check (i.e. $P \cap \overline{\mathcal{L}(\Pi)} \neq \emptyset$), in our case, a failed proof check coincides with the emptiness of an intersection check (i.e. $\mathcal{R} \cap \mathcal{P}(\mathcal{L}(\Pi)) = \emptyset$). The sets $\mathcal{R}$ and $\mathcal{P}(\mathcal{L}(\Pi))$ are both sets of languages. What does the witness to the emptiness of the intersection look like? Each language member of $\mathcal{R}$ contains at least one string that does not belong to any of the subsets of our proof language. One can collect all such witness strings to guarantee progress across the board in the next round. However, since LTAs can represent an infinite set of languages, one must take care not end up with an infinite set of counterexamples following this strategy. Fortunately, this will not be the case.

Theorem 5.4. *Let M be an LTA and let L be a regular language such that $P \not\subseteq L$ for all $P \in \mathcal{L}(M)$. There exists a finite set of counterexamples C such that, for all $P \in \mathcal{L}(M)$, there exists some $x \in C$ such that $x \in P$ and $x \notin L$.*

The proof appears in [14]. This theorem justifies our choice of using LTAs instead of more expressive formalisms such as Büchi Tree Automata. For example, the Büchi Tree Automaton that accepts the language $\{\{x\} \mid x \in \Sigma^*\}$ would give rise to an infinite number of counterexamples with respect to the empty proof (i.e. $\Pi = \emptyset$). The finiteness of the counterexample set presents an alternate proof that LTAs are strictly less expressive than Büchi Tree Automata [27].

6 Sleep Set Reductions

We have established so far that (1) a set of assertions gives rise to a regular language proof, and (2) given a regular language proof and a set of program reductions recognizable by an LTA, we can check the program (reductions) against the proof. The last piece of the puzzle is to show that a useful class of program reductions can be expressed using LTAs.

Recall our example from Section 2. The reduction we obtain is sound because, for every trace in the full parallel-composition program, an equivalent trace exists in the reduced program. By equivalent, we mean that one trace can be obtained from the other by swapping independent statements. Such an equivalence is the essence of the theory of Mazurkiewicz traces [9].

We fix a reflexive symmetric *dependence relation* $D \subseteq \Sigma \times \Sigma$. For all $a, b \in \Sigma$, we say that a and b are *dependent* if $(a, b) \in D$, and say they are *independent* otherwise. We define $\sim_D$ as the smallest congruence satisfying $xaby \sim_D xbay$ for all $x, y \in \Sigma^*$ and independent $a, b \in \Sigma$. The closure of a language $L \subseteq \Sigma^*$ with respect to $\sim_D$ is denoted $[L]_D$. A language L is *$\sim_D$-closed* if $L = [L]_D$. It is worthwhile to note that all input programs considered in this paper correspond to regular languages that are $\sim_D$-closed.

An equivalence class of $\sim_D$ is typically called a (Mazurkiewicz) trace. We avoid using this terminology as it conflicts with our definition of traces as strings of statements in Section 3.1. We assume D is *sound*, i.e. $[\![ab]\!] = [\![ba]\!]$ for all independent $a, b \in \Sigma$.

Definition 6.1 (D-reduction). *A program P' is a* D-reduction *of a program P, that is $P' \preceq_D P$, if $[P']_D = P$.*

Note that the equivalence relation on programs induced by $\sim_D$ is a refinement of the semantic equivalence relation used in Definition 4.1.

Lemma 6.2. *If $P' \preceq_D P$ then $P' \preceq P$.*

Ideally, we would like to define an LTA that accepts all D-reductions of a program P, but unfortunately this is not possible in general.

Proposition 6.3 (corollary of Theorem 67 of [9]). *For arbitrary regular languages $L_1, L_2 \in \Sigma^*$ and relation D, the proposition $\exists L \preceq_D L_1 . L \subseteq L_2$ is undecidable.*

The proposition is decidable only when $\overline{D}$ is transitive, which does not hold for a semantically correct notion of independence for a parallel program encoding a k-safety property, since statements from the same thread are dependent and statements from different program copies are independent. Therefore, we have:

Proposition 6.4. *Assume P is a $\sim_D$-closed program and Π is a proof. The proposition $\exists P' \preceq_D P . P' \subseteq \mathcal{L}(\Pi)$ is undecidable.*

In order to have a decidable premise for proof rule SafeRed2 then, we present an approximation of the set of D-reductions, inspired by sleep sets [15]. The idea is to construct an LTA that recognizes *a class of D-reductions* of an input program P, whose language is assumed to be $\sim_D$-closed. This automaton intuitively makes non-deterministic choices about what program traces to *prune* in favour of other $\sim_D$-equivalent program traces for a given reduction. Different non-deterministic choices lead to different D-reductions.

Consider two statements $a, b \in \Sigma$ where $(a, b) \notin D$. Let $x, y \in \Sigma^*$ and consider two program runs $xaby$ and $xbay$. We know $[\![xbay]\!] = [\![xaby]\!]$. If the automaton makes a non-deterministic choice that the successors of xa have been explored, then the successors of xba need not be explored (can be pruned away) as illustrated in Figure 3. Now assume $(a, c) \in D$, for some $c \in \Sigma$. When the node xbc is being explored, we can no longer safely ignore a-transitions, since the equality $[\![xbcay]\!] = [\![xabcy]\!]$ is not guaranteed. Therefore, the a successor of xbc has to be explored. The nondeterministic choice of what child node to explore is modelled by a choice of order in which we explore each node's children. Different orders yield different reductions. Reductions are therefore characterized as an assignment $R : \Sigma^* \to \mathcal{L}in(\Sigma)$ from nodes to linear orderings on Σ, where $(a, b) \in R(x)$ means we explore child xa after child xb.

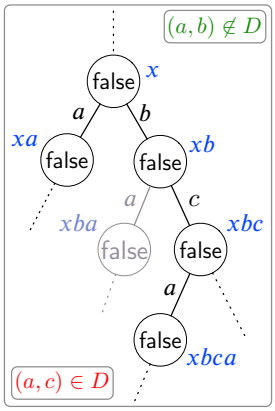

Fig. 3: Exploring from x with sleep sets.

Given $R : \Sigma^* \to \mathcal{L}in(\Sigma)$, the *sleep set* $\text{sleep}_R(x) \subseteq \Sigma$ at node $x \in \Sigma^*$ defines the set of transitions that can be ignored at x:

$$\text{sleep}_R(\epsilon) = \emptyset \tag{1}$$
$$\text{sleep}_R(xa) = (\text{sleep}_R(x) \cup R(x)(a)) \setminus D(a) \tag{2}$$

Intuitively, (1) no transition can be ignored at the root node, since nothing has been explored yet, and (2) at node x, the sleep set of xa is obtained by adding the transitions we explored before a ($R(x)(a)$) and then removing the ones that conflict with a (i.e. are related to a by D). Next, we define the nodes that are ignored. The set of ignored nodes is the smallest set $\text{ignore}_R : \Sigma^* \to \mathbb{B}$ such that

$$x \in \text{ignore}_R \implies xa \in \text{ignore}_R \tag{1}$$
$$a \in \text{sleep}_R(x) \implies xa \in \text{ignore}_R \tag{2}$$

Intuitively, a node xa is ignored if (1) any of its ancestors is ignored ($\text{ignore}_R(x)$), or (2) a is one of the ignored transitions at node x ($a \in \text{sleep}_R(x)$).

Finally, we obtain an actual reduction of a program P from a characterization of a reduction R by removing the ignored nodes from P, i.e. $P \setminus \text{ignore}_R$.

Lemma 6.5. *For all $R : \Sigma^* \to \mathcal{L}in(\Sigma)$, if P is a $\sim_D$-closed program then $P \setminus \text{ignore}_R$ is a D-reduction of P.*

The set of all such reductions is $\text{reduce}_D(P) = \{P \setminus \text{ignore}_R \mid R : \Sigma^* \to \mathcal{L}in(\Sigma)\}$.

Theorem 6.6. *For any regular language P, $\text{reduce}_D(P)$ is accepted by an LTA.*

Interestingly, every reduction in $\text{reduce}_D(P)$ is optimal in the sense that each reduction contains at most one representative of each equivalence class of $\sim_D$.

Theorem 6.7. *Fix some $P \subseteq \Sigma^*$ and $R : \Sigma^* \to \mathcal{L}in(\Sigma)$. For all $(x, y) \in P \setminus \text{ignore}_R$, if $x \sim_D y$ then $x = y$.*

7 Algorithms

Figure 4 illustrates the outline of our verification algorithm. It is a counterexample-guided abstraction refinement loop in the style of [12, 13, 17]. The key difference is that instead of checking whether some proof Π is a proof for the program P, it checks if there exists a reduction of the program P that Π proves correct.

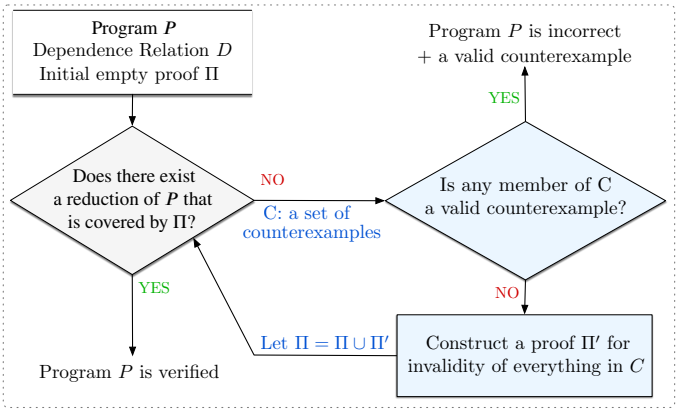

Fig. 4: Counterexample-guided refinement loop.

The algorithm relies on an oracle INTERPOLATE that, given a finite set of program traces C, returns a proof Π', if one exists, such that $C \subseteq \mathcal{L}(\Pi')$. In our tool, we use Craig interpolation to implement the oracle INTERPOLATE. In general, since program traces are the simplest form of sequential programs (loop and branch free), any automated program prover that can handle proving them may be used.

The results presented in Sections 5 and 6 give rise to the proof checking sub routine of the algorithm in Figure 4 (i.e. the light grey test). Given a program DFA $A_P = (Q_P, \Sigma, \delta_P, q_{P0}, F_P)$ and a proof DFA $A_\Pi = (Q_\Pi, \Sigma, \delta_\Pi, q_{\Pi 0}, F_\Pi)$ (obtained by determinizing Π_{NFA}), we can decide $\exists P' \in \text{reduce}_D(\mathcal{L}(A_P)).\, P' \subseteq \mathcal{L}(A_\Pi)$ by constructing an LTA $M_{P\Pi}$ for $\text{reduce}_D(\mathcal{L}(A_P)) \cap \mathcal{P}(\mathcal{L}(A_\Pi))$ and checking emptiness (Theorem 5.2).

7.1 Progress

The algorithm corresponding to Figure 4 satisfies a weak progress theorem: none of the counterexamples from a round of the algorithm will ever appear in a future counterexample set. This, however, is not strong enough to guarantee termination. Alternatively, one can think of the algorithm's progress as follows. In each round new assertions are discovered through the oracle INTERPOLATE, and one can optimistically hope that one can finally converge on an existing target proof Π^*. The success of this algorithm depends on two factors: (1) the counterexamples used by the algorithm belong to $\mathcal{L}(\Pi^*)$ and (2) the proof that INTERPOLATE discovers for these counterexamples coincide with Π^*. The latter is a typical known wild card in software model checking, which cannot be guaranteed; there is plenty of empirical evidence, however, that procedures based on Craig Interpolation do well in approximating it. The former is a new problem for our refinement loop.

In a standard algorithm in the style of [12, 13, 17], the verification proof rule dictates that every program trace must be in $\mathcal{L}(\Pi^*)$. In our setting, we only require a subset (corresponding to some reduction) to be in $\mathcal{L}(\Pi^*)$. This means one cannot simply rely on program traces as *appropriate* counterexamples. Theorem 5.4 presents a solution to this problem. It ensures that we always feed INTERPOLATE some counterexample from Π^* and therefore guarantee progress.

Theorem 7.1 (Strong Progress). *Assume a proof Π^* exists for some reduction $P^* \in \mathcal{R}$ and* INTERPOLATE *always returns some subset of Π^* for traces in $\mathcal{L}(\Pi^*)$. Then the algorithm will terminate in at most $|\Pi^*|$ iterations.*

Theorem 7.1 ensures that the algorithm will never get into an infinite loop due to a bad choice of counterexamples. The condition on INTERPOLATE ensures that divergence does not occur due to the wrong choice of assertions by INTERPOLATE and without it any standard interpolation-based software model checking algorithm may diverge. The assumption that there exists a proof for a reduction of the program in the fixed set $\mathcal{R}$ ensures that the proof checking procedure can verify the target proof Π^* once it is reached. Note that, in general, a proof may exist for a reduction of the program which is not in $\mathcal{R}$. Therefore, the algorithm

is not complete with respect to all reductions, since checking the premises of SafeRed1 is undecidable as discussed in Section 4.

7.2 Faster Proof Checking through Antichains

The state set of $M_{P\Pi}$, the intersection of program and proof LTAs, has size $|Q_P \times \mathbb{B} \times \mathcal{P}(\Sigma) \times Q_\Pi|$, which is exponential in $|\Sigma|$. Therefore, even a linear emptiness test for this LTA can be computationally expensive. Antichains have been previously used [8] to optimize certain operations over NFAs that also suffer from exponential blowups, such as deciding universality and inclusion tests. The main idea is that these operations involve computing downwards-closed and upwards-closed sets according to an appropriate subsumption relation, which can be represented compactly as antichains. We employ similar techniques to propose a new emptiness check algorithm.

Antichains. The set of maximal elements of a set X with respect to some ordering relation $\sqsubseteq$ is denoted $\max(X)$. The downwards-closure of a set X with respect to $\sqsubseteq$ is denoted $\lfloor X \rfloor$. An antichain is a set X where no element of X is related (by $\sqsubseteq$) to another. The maximal elements $\max(X)$ of a finite set X is an antichain. If X is downwards-closed then $\lfloor \max(X) \rfloor = X$.

The emptiness check algorithm for LTAs from [2] computes the set of *inactive* states (i.e. states which generate an empty language) and checks if the initial state is inactive. The set of inactive states of an LTA $M = (Q, \Delta, q_0)$ is defined as the smallest set $\text{inactive}(M)$ satisfying

$$\frac{\forall (q, B, \sigma) \in \Delta.\, \exists a.\, \sigma(a) \in \text{inactive}(M)}{q \in \text{inactive}(M)} \qquad \text{(Inactive)}$$

Alternatively, one can view $\text{inactive}(M)$ as the least fixed-point of a monotone (with respect to $\subseteq$) function $F_M : \mathcal{P}(Q) \to \mathcal{P}(Q)$ where

$$F_M(X) = \{q \mid \forall (q, B, \sigma) \in \Delta.\, \exists a.\, \sigma(a) \in X\}.$$

Therefore, $\text{inactive}(M)$ can be computed using a standard fixpoint algorithm.

If $\text{inactive}(M)$ is downwards-closed with respect to some *subsumption relation* $(\sqsubseteq) \subseteq Q \times Q$, then we need not represent all of $\text{inactive}(M)$. The antichain $\max(\text{inactive}(M))$ of maximal elements of $\text{inactive}(M)$ (with respect to $\sqsubseteq$) would be sufficient to represent the entirety of $\text{inactive}(M)$, and can be exponentially smaller than $\text{inactive}(M)$, depending on the choice of relation $\sqsubseteq$.

A trivial way to compute $\max(\text{inactive}(M))$ is to first compute $\text{inactive}(M)$ and then find the maximal elements of the result, but this involves doing strictly more work than the baseline algorithm. However, observe that if F_M also preserves downwards-closedness with respect to $\sqsubseteq$, then

$$\begin{aligned} \max(\text{inactive}(M)) &= \max(\text{lfp}(F_M)) \\ &= \max(\text{lfp}(F_M \circ \lfloor - \rfloor \circ \max)) = \text{lfp}(\max \circ F_M \circ \lfloor - \rfloor) \end{aligned}$$

That is, $\max(\text{inactive}(M))$ is the least fixed-point of a function $F_M^{\max} : \mathcal{P}(Q) \to \mathcal{P}(Q)$ defined as $F_M^{\max}(X) = \max(F_M(\lfloor X \rfloor))$. We can calculate $\max(\text{inactive}(M))$ efficiently if we can calculate $F_M^{\max}(X)$ efficiently, which is true in the special case of the intersection automaton for the languages of our proof $\mathcal{P}(\mathcal{L}(\Pi))$ and our program $\text{reduce}_D(P)$, which we refer to as $M_{P\Pi}$.

We are most interested in the state space of $M_{P\Pi}$, which is $Q_{P\Pi} = (Q_P \times \mathbb{B} \times \mathcal{P}(\Sigma)) \times Q_\Pi$. Observe that states whose $\mathbb{B}$ part is $\top$ are always active:

Lemma 7.2. *$((q_P, \top, S), q_\Pi) \notin \text{inactive}(M_{P\Pi})$ for all $q_P \in Q_P$, $q_\Pi \in Q_\Pi$, and $S \subseteq \Sigma$.*

The state space can then be assumed to be $Q_{P\Pi} = (Q_P \times \{\bot\} \times \mathcal{P}(\Sigma)) \times Q_\Pi$ for the purposes of checking inactivity. The subsumption relation defined as the smallest relation $\sqsubseteq_{P\Pi}$ satisfying

$$S \subseteq S' \implies ((q_P, \bot, S), q_\Pi) \sqsubseteq_{P\Pi} ((q_P, \bot, S'), q_\Pi)$$

for all $q_P \in Q_P$, $q_\Pi \in Q_\Pi$, and $S, S' \subseteq \Sigma$, is a suitable one since:

Lemma 7.3. *$F_{M_{P\Pi}}$ preserves downwards-closedness with respect to $\sqsubseteq_{P\Pi}$.*

The function $F_{M_{P\Pi}}^{\max}$ is a function over relations

$$F_{M_{P\Pi}}^{\max} : \mathcal{P}((Q_P \times \{\bot\} \times \mathcal{P}(\Sigma)) \times Q_\Pi) \to \mathcal{P}((Q_P \times \{\bot\} \times \mathcal{P}(\Sigma)) \times Q_\Pi)$$

but in our case it is more convenient to view it as a function over functions

$$F_{M_{P\Pi}}^{\max} : (Q_P \times \{\bot\} \times Q_\Pi \to \mathcal{P}(\mathcal{P}(\Sigma))) \to (Q_P \times \{\bot\} \times Q_\Pi \to \mathcal{P}(\mathcal{P}(\Sigma)))$$

Through some algebraic manipulation and some simple observations, we can define $F_{M_{P\Pi}}^{\max}$ functionally as follows.

Lemma 7.4. *For all $q_P \in Q_P$, $q_\Pi \in Q_\Pi$, and $X : Q_P \times \{\bot\} \times Q_\Pi \to \mathcal{P}(\mathcal{P}(\Sigma))$,*

$$F_{M_{P\Pi}}^{\max}(X)(q_P, \bot, q_\Pi) = \begin{cases} \{\Sigma\} & \text{if } q_P \in F_P \wedge q_\Pi \notin F_\Pi \\ \bigsqcap_{R \in \mathcal{L}in(\Sigma)} \bigsqcup_{a \in \Sigma} S' & \text{otherwise} \\ \qquad\qquad\quad S \in X(q'_P, \bot, q'_\Pi) & \end{cases}$$

where

$$q'_P = \delta_P(q_P, a) \qquad X \sqcap Y = \max\{x \cap y \mid x \in X \wedge y \in Y\}$$
$$q'_\Pi = \delta_\Pi(q_\Pi, a) \qquad X \sqcup Y = \max(X \cup Y)$$

$$S' = \begin{cases} \{(S \cup D(a)) \setminus \{a\}\} & \text{if } R(a) \setminus D(a) \subseteq S \\ \emptyset & \text{otherwise} \end{cases}$$

A full justification appears in [14]. Formulating $F_{M_{P\Pi}}^{\max}$ as a higher-order function allows us to calculate $\max(\text{inactive}(M_{P\Pi}))$ using efficient fixpoint algorithms like the one in [22]. Algorithm 1 outlines our proof checking routine. $\textsc{Fix} : ((A \to B) \to (A \to B)) \to (A \to B)$ is a procedure that computes the least fixpoint of its input. The algorithm simply computes the fixpoint of the function $F_{M_{P\Pi}}^{\max}$ as defined in Lemma 7.4, which is a compact representation of $\text{inactive}(M_{P\Pi})$ and checks if the start state of $M_{P\Pi}$ is in it.

```
function Check(A_P, A_Π, D)
    (Q_P, Σ, δ_P, q_0P, F_P) ← A_P
    (Q_Π, Σ, δ_Π, q_0Π, F_Π) ← A_Π
    function FMax(X)((q_P, ⊥, q_Π))
        if q_P ∈ F_P ∧ q_Π ∉ F_Π
            return {Σ}
        X^⊓ ← {Σ}
        for R ∈ Lin(Σ)
            X^⊔ ← ∅
            for a ∈ Σ, S ∈ X((δ_P(q_P, a), ⊥, δ_Π(q_Π, a)))
                if R(a) \ D(a) ⊆ S
                    X^⊔ ← X^⊔ ⊔ {(S ∪ D(a)) \ {a}}
            X^⊓ ← X^⊓ ⊓ X^⊔
        return X^⊓
    return Fix(FMax)((q_0P, ⊥, q_0Π)) ≠ ∅
```

Algorithm 1: Proof checking algorithm

Counterexamples. Theorem 5.4 states that a finite set of counterexamples exists whenever $\exists P' \in \text{reduce}_D(P).\, P' \subseteq \mathcal{L}(\Pi)$ does not hold. The proof of emptiness for an LTA, formed using rule Inactive above, is a finite tree. Each edge in the tree is labelled by an element of Σ (obtained from the existential in the rule) and the paths through this tree form the counterexample set. To compute this set, then, it suffices to remember enough information during the computation of inactive(M) to reconstruct the proof tree. Every time a state q is determined to be inactive, we must also record the witness $a \in \Sigma$ for each transition $(q, B, \sigma) \in \Delta$ such that $\sigma(a) \in \text{inactive}(M)$.

In an antichain-based algorithm, once we determine a state q to be inactive, we simultaneously determine everything it subsumes (i.e. $\sqsubseteq q$) to be inactive as well. If we record unique witnesses for each and every state that q subsumes, then the space complexity of our antichain algorithm will be the same as the unoptimized version. The following lemma states that it is sufficient to record witnesses only for q and discard witnesses for states that q subsumes.

Lemma 7.5. *Fix some states q, q' such that $q' \sqsubseteq_{P\Pi} q$. A witness used to prove q is inactive can also be used to prove q' is inactive.*

Note that this means that the antichain algorithm soundly returns potentially fewer counterexamples than the original one.

7.3 Partition Optimization

The LTA construction for $\text{reduce}_D(P)$ involves a nondeterministic choice of linear order at each state. Since $|\mathcal{L}in(\Sigma)|$ has size $|\Sigma|!$, each state in the automaton would have a large number of transitions. As an optimization, our algorithm selects ordering relations out of $\mathcal{P}art(\Sigma)$ (instead of $\mathcal{L}in(\Sigma)$), defined as $\mathcal{P}art(\Sigma) = \{\Sigma_1 \times \Sigma_2 \mid \Sigma_1 \uplus \Sigma_2 = \Sigma\}$ where $\uplus$ is disjoint union. This leads to a sound

algorithm which is not complete with respect to sleep set reductions and trades the factorial complexity of computing $\mathcal{L}in(\Sigma)$ for an exponential one.

8 Experimental Results

To evaluate our approach, we have implemented our algorithm in a tool called Weaver written in Haskell. Weaver accepts a program written in a simple imperative language as input, where the property is already encoded in the program in the form of *assume* statements, and attempts to prove the program correct. The dependence relation for each input program is computed using a heuristic that ensures $\sim_D$-closedness. It is based on the fact that the shuffle product (i.e. parallel composition) of two $\sim_D$-closed languages is $\sim_D$-closed.

Weaver employs two verification algorithms: (1) The total order algorithm presented in Algorithm 1, and (2) the variation with the partition optimization discussed in Section 7.3. It also implements multiple counterexample generation algorithms: (1) *Naive:* selects the first counterexample in the difference of the program and proof language. (2) *Progress-Ensuring:* selects a set of counterexamples satisfying Theorem 5.4. (3) *Bounded Progress-Ensuring:* selects a few counterexamples (in most cases just one) from the set computed by the progress-ensuring algorithm. Our experimentation demonstrated that in the vast majority of the cases, the bounded progress ensuring algorithm (an instance of the partition algorithm) is the fastest of all options. Therefore, all our reports in this section are using this instance of the algorithm.

For the larger benchmarks, we use a simple sound optimization to reduce the proof size. We declare the basic blocks of code as atomic, so that internal assertions need not be generated for them as part of the proof. This optimization is incomplete with respect to sleep set reductions.

Benchmarks. We use a set of sequential benchmarks from [24] and include additional sequential benchmarks that involve more interesting reductions in their proofs. We have a set of parallel benchmarks, which are beyond the scope of previous hypersafety verification techniques. We use these benchmarks to demonstrate that our technique/tool can seamlessly handle concurrency. These involve proving concurrency specific hypersafety properties such as determinism and equivalence of parallel and sequential implementations of algorithms. Finally, since the proof checking algorithm is the core contribution of this paper, we have a contrived set of instances to stress test our algorithm. These involve proving determinism of simple parallel-disjoint programs with various numbers of threads and statements per thread. These benchmarks have been designed to cause a combinatorial explosion for the proof checker and counterexample generation routines. More information on the benchmarks can be found in [14].

Evaluation

Due to space restrictions, it is not feasible to include a detailed account of all our experiments here, for over 50 benchmarks. A detailed table can be found in [14]. Table 1 includes a summary in the form of averages, and here, we discuss our top findings.

Benchmark Group	Group Count	Proof Size	Number of Refinement Rounds	Proof Construction Time	Proof Checking Time	Total Time
Looping programs of [24] 2-safety properties	5	63	12	46.69s	0.1s	47.03s
Looping programs of [24] 3-safety properties	8	155	22	475.78s	11.79s	448.36s
Loop-free programs of [24]	27	5	2	0.13s	0.0004s	0.15s
Our sequential benchmarks	13	30	9	14.27s	2.5s	17.94s
Our parallel benchmarks	7	31	8	17.95	0.56s	18.63s

Table 1: Experimental results averages for benchmark groups.

Proof construction time refers to the time spent to construct $\mathcal{L}(\Pi)$ from a given set of assertions Π and excludes the time to produce proofs for the counterexamples in a given round. **Proof checking time** is the time spent to check if the current proof candidate is strong enough for a reduction of the program. In the fastest instances (total time around 0.01 seconds), roughly equal time is spent in proof checking and proof construction. In the slowest instances, the total time is almost entirely spent in proof construction. In contrast, in our stress tests (designed to stress the proof checking algorithm) the majority of the time is spent in proof checking. The time spent in proving counterexamples correct is negligible in all instances. **Proof sizes** vary from 4 assertions to 298 for the most complicated instance. Verification times are *correlated* with the final proof size; larger proofs tend to cause longer verification times.

Numbers of refinement rounds vary from 2 for the simplest to 33 for the most complicated instance. A small number of refinement rounds (e.g. 2) implies a fast verification time. But, for the higher number of rounds, a strong positive correlation between the number of rounds and verification time does not exist.

For our **parallel programs** benchmarks (other than our stress tests), the tool spends the majority of its time in proof construction. Therefore, we designed specific (unusual) parallel programs to stress test the proof checker. **Stress test** benchmarks are trivial tests of determinism of disjoint parallel programs, which can be proven correct easily by using the atomic block optimization. However, we force the tool to do the unnecessary hard work. These instances simulate the worst case theoretical complexity where the proof checking time and number of counterexamples grow exponentially with the number of threads and the sizes of the threads. In the largest instance, more than 99% of the total verification time is spent in proof checking. Averages are not very informative for these instances, and therefore are not included in Table 1.

Finally, Weaver is only slow for verifying 3-safety properties of large looping benchmarks from [24]. Note that unlike the approach in [24], which starts from a default lockstep reduction (that is incidentally sufficient to prove these instances), we do not assume any reduction and consider them all. The extra time is therefore expected when the product programs become quite large.

9 Related Work

The notion of a k-safety hyperproperty was introduced in [7] without consideration for automatic program verification. The approach of reducing k-safety to 1-safety by self-composition is introduced in [5]. While theoretically complete, self-composition is not practical as discussed in Section 1. Product programs generalize the self-composition approach and have been used in verifying translation validation [20], non-interference [16, 23], and program optimization [25]. A product of two programs P_1 and P_2 is semantically equivalent to $P_1 \cdot P_2$ (sequential composition), but is made easier to verify by allowing parts of each program to be interleaved. The product programs proposed in [3] allow lockstep interleaving exclusively, but only when the control structures of P_1 and P_2 match. This restriction is lifted in [4] to allow some non-lockstep interleavings. However, the given construction rules are non-deterministic, and the choice of product program is left to the user or a heuristic.

Relational program logics [6, 28] extend traditional program logics to allow reasoning about relational program properties, however automation is usually not addressed. Automatic construction of product programs is discussed in [10] with the goal of supporting procedure specifications and modular reasoning, but is also restricted to lockstep interleavings. Our approach does not support procedure calls but is fully automated and permits non-lockstep interleavings.

The key feature of our approached is the automation of the discovery of an appropriate program reduction and a proof combined. In this case, the only other method that compares is the one based on Cartesian Hoare Logic (CHL) proposed in [24] along with an algorithm for automatic verification based on CHL. Their proposed algorithm implicitly constructs a product program, using a heuristic that favours lockstep executions as much as possible, and then prioritizes certain rules of the logic over the rest. The heuristic nature of the search for the proof means that no characterization of the search space can be given, and no guarantees about whether an appropriate product program will be found. In contrast, we have a formal characterization of the set of explored product programs in this paper. Moreover, CHL was not designed to deal with concurrency.

Lipton [19] first proposed reduction as a way to simplify reasoning about concurrent programs. His ideas have been employed in a semi-automatic setting in [11]. Partial-order reduction (POR) is a class of techniques that reduces the state space of search by removing redundant paths. POR techniques are concerned with finding a single (preferably minimal) reduction of the input program. In contrast, we use the same underlying ideas to explore many program reductions simultaneously. The class of reductions described in Section 6 is based on the sleep set technique of Godefroid [15]. Other techniques exist [15, 1] that are used in conjunction with sleep sets to achieve minimality in a normal POR setting. In our setting, reductions generated by sleep sets are already optimal (Theorem 6.7). However, employing these additional POR techniques may propose ways of optimizing our proof checking algorithm by producing a smaller reduction LTA.

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. Journal of the ACM (JACM) **64**(4), 25 (2017)
2. Baader, F., Tobies, S.: The inverse method implements the automata approach for modal satisfiability. In: International Joint Conference on Automated Reasoning. pp. 92–106. Springer (2001)
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: International Symposium on Formal Methods. pp. 200–214. Springer (2011)
4. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: International Symposium on Logical Foundations of Computer Science. pp. 29–43. Springer (2013)
5. Barthe, G., D'argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: ACM SIGPLAN Notices. vol. 39, pp. 14–25. ACM (2004)
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 21st IEEE Computer Security Foundations Symposium. pp. 51–65. IEEE (2008)
8. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: International Conference on Computer Aided Verification. pp. 17–30. Springer (2006)
9. Diekert, V., Métivier, Y.: Partial commutation and traces. In: Handbook of formal languages, pp. 457–533. Springer (1997)
10. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: European Symposium on Programming. pp. 502–529. Springer (2018)
11. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: ACM SIGPLAN Notices. vol. 44, pp. 2–15. ACM (2009)
12. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: ACM SIGPLAN Notices. vol. 48, pp. 129–142. ACM (2013)
13. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: ACM SIGPLAN Notices. vol. 50, pp. 407–420. ACM (2015)
14. Farzan, A., Vandikas, A.: Reductions for automated hypersafety verification (2019)
15. Godefroid, P., Van Leeuwen, J., Hartmanis, J., Goos, G., Wolper, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem, vol. 1032. Springer Heidelberg (1996)
16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy, 1982 IEEE Symposium on. pp. 11–11. IEEE (1982)
17. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: International Static Analysis Symposium. pp. 69–85. Springer (2009)
18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
19. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Communications of the ACM **18**(12), 717–721 (1975)
20. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 151–166. Springer (1998)
21. Popeea, C., Rybalchenko, A., Wilhelm, A.: Reduction for compositional verification of multi-threaded programs. In: Formal Methods in Computer-Aided Design (FMCAD), 2014. pp. 187–194. IEEE (2014)

22. Pottier, F.: Lazy least fixed points in ml
23. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications **21**(1), 5–19 (2003)
24. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: ACM SIGPLAN Notices. vol. 51, pp. 57–69. ACM (2016)
25. Sousa, M., Dillig, I., Vytiniotis, D., Dillig, T., Gkantsidis, C.: Consolidation of queries with user-defined functions. In: ACM SIGPLAN Notices. vol. 49, pp. 554–564. ACM (2014)
26. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: International Static Analysis Symposium. pp. 352–367. Springer (2005)
27. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information & Computation **115**(1), 1–37 (1994)
28. Yang, H.: Relational separation logic. Theoretical Computer Science **375**(1-3), 308–334 (2007)