Reductions for Automated Hypersafety Verification

AZADEH FARZAN, University of Toronto
ANTHONY VANDIKAS, University of Toronto

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 [25] 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, 23], or define a general class of reductions, but do not provide algorithms with guarantees of covering the entire class [4, 23].

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, 20]. 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:

```
Mult:
ℓ1: i ← 0
ℓ2: x ← 0
ℓ3: while i < a
ℓ4:     x ← x + b
ℓ5:     i ← i + 1
ℓ6:                         (i)
```

```
Copy 1:                   Copy 2:                 Copy 3:
ℓ1: i1 ← 0                ℓ1: i2 ← 0              ℓ1: i3 ← 0
ℓ2: x1 ← 0                ℓ2: x2 ← 0              ℓ2: x3 ← 0
ℓ3: while i1 < a + b      ℓ3: while i2 < a        ℓ3: while i3 < b
ℓ4:     x1 ← x1 + c       ℓ4:     x2 ← x2 + c     ℓ4:     x3 ← x3 + c
ℓ5:     i1 ← i1 + 1       ℓ5:     i2 ← i2 + 1     ℓ5:     i3 ← i3 + 1
ℓ6:                       ℓ6:                     ℓ6:                 (ii)
```

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

- 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 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 $true \implies \phi_{111}$ 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.

```
i1 ← 0, i2 ← 0, i3 ← 0
x1 ← 0, x2 ← 0, x3 ← 0
while i2 < a
    x1 ← x1 + c
    x2 ← x2 + c
    i1 ← i1 + 1
    i2 ← i2 + 1
while i3 < b
    x1 ← x1 + c
    x3 ← x3 + c
    i1 ← i1 + 1
    i3 ← i3 + 1
```

The program on the right is a semantically equivalent *reduction* of the full composition of Figure 1(ii). Consider the program P = (Copy 1 || (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.

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 [17].

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, 16]:

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

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:

$$\frac{\exists P' \preceq P, \Pi \subseteq \mathcal{A}.\, P' \subseteq \mathcal{L}(\Pi)}{P \text{ is safe}} \qquad (\textsc{SafeRed1})$$

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:

$$\frac{\exists P' \in \mathcal{R}.\, P' \subseteq \mathcal{L}(\Pi) \qquad \forall P' \in \mathcal{R}.\, P' \preceq P}{P \text{ is safe}} \qquad (\textsc{SafeRed2})$$

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

Proof. By the left precondition, there exists some $P' \in \mathcal{R}$ such that $P' \subseteq \mathcal{L}(\Pi)$, which implies $[\![P']\!] = \emptyset$. By the right precondition, $[\![P]\!] = [\![P']\!]$, and therefore P is safe. □

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 known partial order reduction techniques [14].

5 PROOF CHECKING

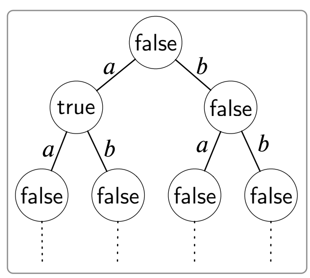


Fig. 2. $L = \{a\}$ as an infinite tree.

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. 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 $\exists P \in \mathcal{L}(M).\, P \subseteq L$.*

PROOF. The proposition $\exists P \in \mathcal{L}(M).\, P \subseteq L$ is equivalent to the proposition $\mathcal{L}(M) \cap \mathcal{P}(L) = \emptyset$. LTA languages are closed under intersection (Lemma 5.3), $\mathcal{P}(L)$ is recognized by an LTA (Lemma 5.4), and LTA emptiness is decidable [2], so $\mathcal{L}(M) \cap \mathcal{P}(L) = \emptyset$ (and therefore $\exists P \in \mathcal{L}(M).\, P \subseteq L$) is decidable. □

LEMMA 5.3. *The set of languages accepted by an LTA is closed under intersection.*

PROOF. The standard construction for Büchi tree automata intersection also works for LTAs. There is a simpler construction specifically for LTAs, which we include here.

Let $M_1 = (Q_1, \Delta_1, q_{01})$ and $M_2 = (Q_2, \Delta_2, q_{02})$ be LTAs. Define $M_\cap = (Q_1 \times Q_2, \Delta_\cap, (q_{01}, q_{02}))$, where

$$\Delta_\cap = \{((q_1, q_2), B, \lambda a.\, (\sigma_1(a), \sigma_2(a))) \mid (q_1, B, \sigma_1) \in \Delta_1 \land (q_2, B, \sigma_2) \in \Delta_2\}$$

Then $\mathcal{L}(M_\cap) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.

This proof has been mechanically checked. □

LEMMA 5.4. *If L is a regular language, then $\mathcal{P}(L)$ is recognized by an LTA.*

PROOF. Since L is a regular language, there exists a DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(A) = L$. Define $M_{\mathcal{P}(L)} = (Q, \Delta_{\mathcal{P}(L)}, q_0)$ where

$$\Delta_{\mathcal{P}(L)} = \{(q, B, \lambda a.\, \delta(q, a)) \mid B \implies q \in F\}$$

Then $\mathcal{L}(M_{\mathcal{P}(L)}) = \mathcal{P}(L)$.

This proof has been mechanically checked. □

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 $P \nsubseteq \mathcal{L}(\Pi)$ 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 5.5. *Let M be an LTA and let L be a regular language such that $P \nsubseteq L$ 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 $x \notin L$.*

PROOF. Assume $M = (Q_M, \Delta_M, q_{0M})$ and let $A = (Q_A, \Sigma, \delta_A, q_{0A}, F_A)$ be an automaton that accepts L.

Assume $P \nsubseteq L$ for all $P \in \mathcal{L}(M)$. Then $\mathcal{L}(M) \cap \mathcal{P}(L) = \emptyset$, so the root node of any automaton accepting $\mathcal{L}(M) \cap \mathcal{P}(L)$ is *inactive* [2]. The set of inactive states is the smallest set satisfying

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

If we instantiate this rule for $M_\cap$ (from Lemma 5.3) and $M_{\mathcal{P}(L)}$ (from Lemma 5.4), we get

$$\frac{\forall (q_M, B, \sigma) \in \Delta_M.\ (B \implies q_A \in F_A) \implies \exists a.\ (\sigma(a), \delta_A(q_A, a)) \in \text{inactive}(M_\cap)}{(q_M, q_A) \in \text{inactive}(M_\cap)}$$

A proof of inactivity is essentially a finite tree where every node is labelled by a pair of states (q_M, q_A) (the root node labelled by (q_{0M}, q_{0A})) and contains an outgoing edge labelled by some $a \in \Sigma$ for every transition. We define our counterexample set C as the set of all strings labelling the path from the root node to any leaf node in this tree. Such a set is clearly finite, so it remains to show that every $P \in \mathcal{L}(M)$ contains some element of C that is not in L.

Fix some $P \in \mathcal{L}(M)$ and let $\delta^* : \Sigma^* \to Q_M$ be the corresponding accepting run. Since the every node in the tree described above contains an outgoing edge for every transition, there must exist a sequence of nodes through the tree labelled by a sequence of states $(q_{0M}, q_{0A}) \ldots (q_{nM}, q_{nA})$ and string $a = a_1 \ldots a_n \in C$ such that $\delta^*(a_1 \ldots a_i) = q_{iM}$ for all $0 \leq i \leq n$. The state (q_{nM}, q_{nA}) must have no outgoing transitions or else it would not label a leaf node. However, both q_{nM} and q_{nA} must have outgoing transitions in their own respective automata: q_{nM} is part of an accepting run, and every state in $M_\subseteq$ has an outgoing transition by definition. Thus, $(q_{nM}, B, \sigma) \in \Delta_M$ implies $B = \top$ and $q_{nA} \notin F_A$, and therefore $a \in P$ and $a \notin L$.

This proof has been mechanically checked. □

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 [26].

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 $P' \preceq P$.*

PROOF. Since D is sound, then $x \sim_D y$ implies $[\![x]\!] = [\![y]\!]$ for all $x, y \in \Sigma^*$. Then

$$
\begin{aligned}
(a, b) \in [\![P]\!] &\iff \exists x \in P.\, (a, b) \in [\![x]\!] \\
&\iff \exists x \in [P']_D.\, (a, b) \in [\![x]\!] \\
&\iff \exists x.\, \exists y \in P'.\, x \sim_D y \wedge (a, b) \in [\![x]\!] \\
&\iff \exists x.\, \exists y \in P'.\, x \sim_D y \wedge (a, b) \in [\![y]\!] \\
&\iff \exists y \in P'.\, (a, b) \in [\![y]\!] \\
&\iff (a, b) \in [\![P']\!]
\end{aligned}
$$

for all $a, b \in \mathcal{S}t$, so $[\![P']\!] = [\![P]\!]$ and therefore $P' \preceq P$. □

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.*

PROOF. Assume that we can decide whether $\exists L \preceq_D L_1.\, L \subseteq L_2$. Then we can decide whether $[L']_D = \Sigma^*$ for any regular language $L' \subseteq \Sigma^*$ (by instantiating $L_1 = \Sigma^*$ and $L_2 = L'$), which is known to be generally undecidable [9]. □

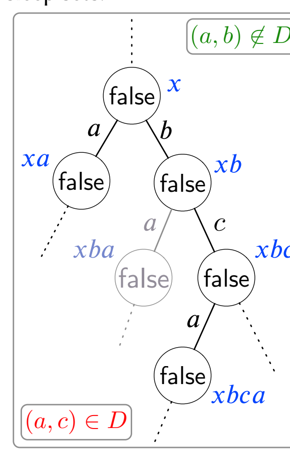
The proposition is decidable only when $\overline{D}$ is transitive, which does not hold for a semantically correct notion of independence for a parallel program encoding a k-safety property, since statements from the same thread are dependent and statements from different program copies are independent. 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 [14]. 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 make precise which nodes 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.*

Proof. If $[\overline{\text{ignore}_R}]_D = \Sigma^*$, then

$$\begin{aligned}[P \setminus \text{ignore}_R]_D &= [P \cap \overline{\text{ignore}_R}]_D \\ &= P \cap [\overline{\text{ignore}_R}]_D \\ &= P \cap \Sigma^* \\ &= P\end{aligned}$$

so it is sufficient to show $[\overline{\text{ignore}_R}]_D = \Sigma^*$. More specifically, it is sufficient to show that for all $x \in \text{ignore}_R$ there exists some $y \notin \text{ignore}_R$ such that $x \sim_D y$.

First, observe that

$$\text{ignore}_R = \{x_1 a x_2 b x_3 \mid (a, b) \in R(x_1) \land (\forall c \in a x_2.\, (c, b) \notin D)\}$$

Second, define the following ordering on traces

$$xay <_R xbz \iff (b, a) \in R(x) \land |y| = |z|$$

The $|y| = |z|$ condition enforces that only strings of the same length are related. Since each $R(x)$ is a linear order on a finite set, it follows that $<_R$ is well-founded.

Assume $x \in \text{ignore}_R$. We proceed by well-founded induction on x using $<_R$. By the above observation, we have $x = x_1 a x_2 b x_3$ for some $x_1, x_2, x_3 \in \Sigma$ and $(a, b) \in R(x_1)$ such that $(c, b) \notin D$ for all $c \in a x_2$. Define $y = x_1 b a x_2 x_3$. Thus $x \sim_D y$ and $y <_R x$. If $y \notin \text{ignore}_R$, then we are done. Otherwise, we have $y \in \text{ignore}_R$ and $y <_R x$, and so induction completes the proof.

This proof has been mechanically checked. □

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

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

Proof. Since P is regular, there exists a DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(A) = P$. Define $M_D = (Q \times \mathbb{B} \times \mathcal{P}(\Sigma), \Delta_D, (q_0, \bot, \emptyset))$ where

$$\Delta_D = \{((q, \iota, S), q \in F \land \neg\iota, \lambda a.\, (\delta(q, a), \iota \lor a \in S, (S \cup R(a)) \setminus D(a)) \mid R \in \mathcal{L}in(\Sigma)\}$$

The values of $\text{sleep}_R(x)$ and $x \in \text{ignore}_R$ can be computed by a simple left-to-right traversal of the input string x. Intuitively, M simulates this computation for a nondeterministically chosen $R : \Sigma^* \to \mathcal{L}in(\Sigma)$. Partial computations of sleep_R (ignore_R) are stored in the $\mathcal{P}(\Sigma)$ ($\mathbb{B}$) part of the state. Thus $\mathcal{L}(M_D) = \text{reduce}_D(P)$.

This proof has been mechanically checked. □

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 $x = y$.*

Proof. It is sufficient to show that $\overline{\text{ignore}_R}$ contains at most one representative of each equivalence class of $\sim_D$, i.e. for all $x, y \notin \text{ignore}_R$, if $x \sim_D y$ then $x = y$.

Assume $x, y \notin \text{ignore}_R$ and $x \sim_D y$. For a contradiction, assume $x \neq y$. Then x and y must differ at some character, so $x = x_1 a x_2$ and $y = x_1 b x_3$ for $x_i \in \Sigma^*$ and $a \neq b$. Assume (by symmetry) that $(a, b) \in R$. We have $x \sim_D y$, so b must appear somewhere in x_2 after a run of elements independent of b, i.e. $x = x_1 a x_{21} b x_{22}$ where $(b, c) \in D$ for every $c \in x_{21}$. However, this implies $b \in \text{sleep}_R(x_1 a x_{21})$, which implies $x_1 a x_{21} b \in \text{ignore}_R$, and therefore $x \in \text{ignore}_R$, which is a contradiction.

This proof has been mechanically checked. □

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, 16]. 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.

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 the proof NFA 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).

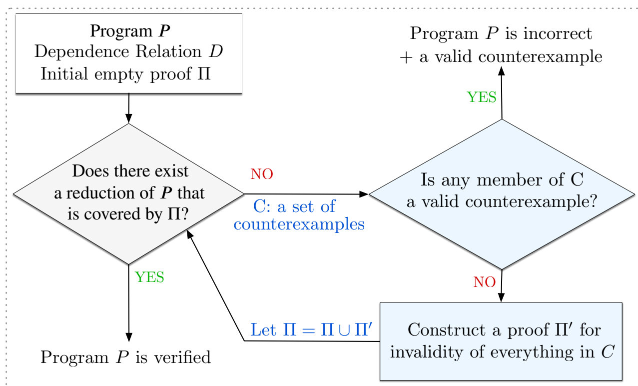


Fig. 4. Counterexample-guided refinement loop.

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, 16], 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 5.5 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.*

Proof. It is sufficient to show that we learn at least one new assertion in Π^* every iteration. Assume we have received a counterexample set C such that, for all $P' \in \mathcal{R}$, there exists some $x \in C$ such that $x \in P'$ and $x \notin \mathcal{L}(\Pi)$ (Theorem 5.5 ensures C exists). Let $x^* \in C$ be the counterexample for P^*. Then Interpolate(x) will return new assertions $\Pi' \subseteq \Pi^*$ satisfying $x^* \in \mathcal{L}(\Pi')$. If $\Pi' \subseteq \Pi$

then x^* would not have been returned as a counterexample, so there must exist some $\phi \in \Pi'$ (and therefore $\phi \in \Pi^*$) such that $\phi \notin \Pi$. □

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

Proof. Let $A_P = (Q_P, \Sigma, \delta_P, q_{0P}, F_P)$ and $A_\Pi = (Q_\Pi, \Sigma, \delta_\Pi, q_{0\Pi}, F_\Pi)$ be automata recognizing P and $\mathcal{L}(\Pi)$, respectively. Then

$$
\begin{aligned}
& F_{M_{P\Pi}}(X) \\
&= \{q \mid \forall (q, B, \sigma) \in \Delta_{P\Pi}.\ \exists a.\ \sigma(a) \in X\} \\
&= \{q \mid \forall (q, B, \sigma) \in \Delta_{\cap}.\ \exists a.\ \sigma(a) \in X\} && (M_{P\Pi} \text{ is an intersection construction}) \\
&= \{\ (q_P, q_\Pi) \mid && (\text{Expanding } \Delta_\cap \text{ from Lemma 5.3}) \\
&\quad \forall (q_P, B, \sigma_1) \in \Delta_D, (q_\Pi, B, \sigma_2) \in \Delta_{\mathcal{P}(\mathcal{L}(\Pi))}. \\
&\quad \exists a.\ (\sigma_1(a), \sigma_2(a)) \in X\} \\
&= \{\ (q_P, q_\Pi) \mid && (\text{Expanding } \Delta_{\mathcal{P}(\mathcal{L}(\Pi))} \text{ from Lemma 5.4}) \\
&\quad \forall (q_P, B, \sigma_1) \in \Delta_D.\ (B \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a.\ (\sigma_1(a), \delta_\Pi(q_\Pi, a)) \in X\} \\
&= \{\ ((q_P, \iota, S), q_\Pi) \mid && (\text{Expanding } \Delta_D \text{ from Lemma 6.6}) \\
&\quad \forall R \in \mathcal{L}in(R).\ (q_P \in F_P \land \neg\iota \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a.\ ((q'_P, \iota \lor a \in S, (S \cup R(a)) \setminus D(a)), \delta_\Pi(q_\Pi, a)) \in X\}
\end{aligned}
$$

Note that if $\iota = \top$ then the body of the set comprehension simplifies to

$$\forall R \in \mathcal{L}in(R).\ \exists a.\ ((q'_P, \top, (S \cup R(a)) \setminus D(a)), \delta_\Pi(q_\Pi, a)) \in X$$

In other words, a state where $\iota = \top$ always has a transition to another state with $\iota = \top$. Therefore such states cannot be inactive.

This proof has been mechanically checked. □

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 $\sqsubseteq_{P\Pi}$.*

Proof. Let $A_P = (Q_P, \Sigma, \delta_P, q_{0P}, F_P)$ and $A_\Pi = (Q_\Pi, \Sigma, \delta_\Pi, q_{0\Pi}, F_\Pi)$ be automata recognizing P and $\mathcal{L}(\Pi)$, respectively. Then

$$
\begin{aligned}
& F_{M_{P\Pi}}(X) \\
&= \{q \mid \forall (q, B, \sigma) \in \Delta_{P\Pi}.\ \exists a.\ \sigma(a) \in X\} \\
&= \{q \mid \forall (q, B, \sigma) \in \Delta_{\cap}.\ \exists a.\ \sigma(a) \in X\} && (M_{P\Pi} \text{ is an intersection construction}) \\
&= \{\ (q_P, q_\Pi) \mid && (\text{Expanding } \Delta_\cap \text{ from Lemma 5.3}) \\
&\quad \forall (q_P, B, \sigma_1) \in \Delta_D, (q_\Pi, B, \sigma_2) \in \Delta_{\mathcal{P}(\mathcal{L}(\Pi))}. \\
&\quad \exists a.\ (\sigma_1(a), \sigma_2(a)) \in X\}
\end{aligned}
$$

$$
\begin{aligned}
&= \{\, (q_P, q_\Pi) \mid && \text{(Expanding } \Delta_{\mathcal{P}(\mathcal{L}(\Pi))} \text{ from Lemma 5.4)}\\
&\quad \forall (q_P, B, \sigma_1) \in \Delta_D.\, (B \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a.\, (\sigma_1(a), \delta_\Pi(q_\Pi, a)) \in X\} \\
&= \{\, ((q_P, \iota, S), q_\Pi) \mid && \text{(Expanding } \Delta_D \text{ from Lemma 6.6)}\\
&\quad \forall R \in \mathcal{L}in(R).\, (q_P \in F_P \wedge \neg\iota \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a.\, ((q'_P, \iota \vee a \in S, (S \cup R(a)) \setminus D(a)), \delta_\Pi(q_\Pi, a)) \in X\} \\
&= \{\, ((q_P, \bot, S), q_\Pi) \mid && \text{(Restricting } \mathbb{B}\text{-part of the domain to } \{\bot\})\\
&\quad \forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a \notin S.\, ((q'_P, \bot, (S \cup R(a)) \setminus D(a)), \delta_\Pi(q_\Pi, a)) \in X\}
\end{aligned}
$$

where

$$
q'_P = \delta_P(q_P, a) \qquad\qquad q'_\Pi = \delta_\Pi(q_\Pi, a)
$$

Recall the subsumption relation $\sqsubseteq_{P\Pi}$:

$$
S \subseteq S' \implies ((q_P, \bot, S), q_\Pi) \sqsubseteq ((q_P, \bot', S'), q_\Pi)
$$

Assume $X \subseteq (Q_P \times \{\bot\} \times \mathcal{P}(\Sigma)) \times Q_\Pi$ is downwards-closed with respect to $\sqsubseteq_{P\Pi}$. For any $S, S' \subseteq \Sigma$ such that $S \subseteq S'$, we have

$$
(S \cup R(a)) \setminus D(a) \subseteq S' \cup R(a) \setminus D(a)
$$

for all $R \in \mathcal{L}in(\Sigma)$, so $F_{P\Pi}(X)$ is also downwards-closed with respect to $\sqsubseteq_{P\Pi}$.

This proof has been mechanically checked. □

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

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

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

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

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

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

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

where

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

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

Proof. From the proof of Lemma 7.3 we have

$$\begin{aligned}
&F_{M_{P\Pi}}(X) \\
&= \{ ((q_P, \bot, S), q_\Pi) \mid \\
&\quad \forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a \notin S.\, ((q'_P, \bot, (S \cup R(a)) \setminus D(a)), q'_\Pi) \in X\}
\end{aligned}$$

where

$$q'_P = \delta_P(q_P, a) \qquad\qquad q'_\Pi = \delta_\Pi(q_\Pi, a)$$

Since $\mathcal{P}((Q_\Pi \times \{\bot\} \times \mathcal{P}(\Sigma)) \times Q_P) \simeq Q_\Pi \times \{\bot\} \times Q_P \to \mathcal{P}(\mathcal{P}(\Sigma))$ we can reformulate $F_{M_{P\Pi}}$ as a function

$$\begin{aligned}
&F_{M_{P\Pi}}(X)(q_P, \bot, q_\Pi) \\
&= \{ S \mid \forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a \notin S.\, (S \cup R(a)) \setminus D(a) \in X(q'_P, \bot, q'_\Pi)\}
\end{aligned}$$

and therefore

$$\begin{aligned}
&F^{\max}_{M_{P\Pi}}(X)(q_P, \bot, q_\Pi) \\
&= \max\{ S \mid \forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&\qquad \exists a \notin S.\, (S \cup R(a)) \setminus D(a) \in \lfloor X(q'_P, \bot, q'_\Pi) \rfloor\} \\
&= \begin{cases} \{\Sigma\} & \text{if } q_P \in F_P \wedge q_\Pi \notin F_\Pi \\ \max\{ S \mid \forall R \in \mathcal{L}in(R).\, \exists a \notin S. & \text{otherwise} \\ \qquad (S \cup R(a)) \setminus D(a) \in \lfloor X(q'_P, \bot, q'_\Pi) \rfloor\} & \end{cases}
\end{aligned}$$

The first case is already in the form we want, so we focus on the second case:

$$\begin{aligned}
&\max\{ S \mid \forall R \in \mathcal{L}in(R).\, \exists a \notin S. \\
&\qquad (S \cup R(a)) \setminus D(a) \in \lfloor X(q'_P, \bot, q'_\Pi) \rfloor\} \\
&= \max\{ S \mid \forall R \in \mathcal{L}in(R).\, \exists a \in \Sigma, S^\uparrow \in X(q'_P, \bot, q'_\Pi). \\
&\qquad a \notin S \wedge (S \cup R(a)) \setminus D(a) \subseteq S^\uparrow\} \\
&= \bigsqcap_{R \in \mathcal{L}in(R)} \bigsqcup_{\substack{a \in \Sigma \\ S^\uparrow \in X(q'_P, \bot, q'_\Pi)}} \max\{a \notin S \wedge (S \cup R(a)) \setminus D(a) \subseteq S^\uparrow\} \\
&= \bigsqcap_{R \in \mathcal{L}in(R)} \bigsqcup_{\substack{a \in \Sigma \\ S^\uparrow \in X(q'_P, \bot, q'_\Pi)}} \max\{a \notin S \wedge S \cup R(a) \subseteq S^\uparrow \cup D(a)\} \\
&= \bigsqcap_{R \in \mathcal{L}in(R)} \bigsqcup_{\substack{a \in \Sigma \\ S^\uparrow \in X(q'_P, \bot, q'_\Pi)}} \max\{a \notin S \wedge S \subseteq S^\uparrow \cup D(a) \wedge R(a) \subseteq S^\uparrow \cup D(a)\} \\
&= \bigsqcap_{R \in \mathcal{L}in(R)} \bigsqcup_{\substack{a \in \Sigma \\ S^\uparrow \in X(q'_P, \bot, q'_\Pi)}} \max\{S \subseteq (S^\uparrow \cup D(a)) \setminus \{a\} \wedge R(a) \subseteq S^\uparrow \cup D(a)\} \\
&= \bigsqcap_{R \in \mathcal{L}in(R)} \bigsqcup_{\substack{a \in \Sigma \\ S \in X(q'_P, \bot, q'_\Pi)}} S'
\end{aligned}$$

This proof has been mechanically checked. □

Formulating $F^{\max}_{M_{P\Pi}}$ as a higher-order function allows us to calculate $\max(\text{inactive}(M_{P\Pi}))$ using efficient fixpoint algorithms like the one in [21]. Algorithm 1 outlines our proof checking routine. 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 $F^{\max}_{M_{P\Pi}}$ 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)
  $(Q_P, \Sigma, \delta_P, q_{0P}, F_P) \leftarrow A_P$
  $(Q_\Pi, \Sigma, \delta_\Pi, q_{0\Pi}, F_\Pi) \leftarrow A_\Pi$
  function FMax(X)$((q_P, \bot, q_\Pi))$
    if $q_P \in F_P \land q_\Pi \notin F_\Pi$
      return $\{\Sigma\}$
    $X^\sqcap \leftarrow \{\Sigma\}$
    for $R \in \mathcal{L}in(\Sigma)$
      $X^\sqcup \leftarrow \emptyset$
      for $a \in \Sigma, S \in$ X$((\delta_P(q_P, a), \bot, \delta_\Pi(q_\Pi, a)))$
        if $R(a) \setminus D(a) \subseteq S$
          $X^\sqcup \leftarrow X^\sqcup \sqcup \{(S \cup D(a)) \setminus \{a\}\}$
      $X^\sqcap \leftarrow X^\sqcap \sqcap X^\sqcup$
    return $X^\sqcap$
  return Fix(FMax)$((q_{0P}, \bot, q_{0\Pi})) \neq \emptyset$

Algorithm 1: Proof checking algorithm

Complexity. Antichain methods do not generally improve worst case time complexity, as the size of the largest antichain in $\mathcal{P}(\Sigma)$ is exponential in $|\Sigma|$. Therefore, our fixpoint algorithm can perform up to $|Q_P||Q_\Pi|2^{|\Sigma|}$ iterations in the worst case. At each iteration of the fixpoint algorithm, $F^{\max}(X)(q_P, \bot, q_\Pi)$ must be recalculated for each $q_P \in Q_P$ and $q_\Pi \in Q_\Pi$, where $X : Q_P \times \bot \times Q_\Pi \to \mathcal{P}(\mathcal{P}(\Sigma))$ is the current assignment calculated by the fixpoint algorithm.

To analyze the complexity of $F^{\max}$, first, note that antichain meet ($\sqcap$) and join ($\sqcup$) can be computed in $O((n_1 n_2)^2)$ and $O(n_1 n_2)$ time, respectively, where n_1 and n_2 are the cardinalities of the left and right arguments, respectively. The complexities of iterated antichain meet ($\bigsqcap$) and join ($\bigsqcup$) over a set of n elements with cardinality at most m are therefore

$$O((mm)^2 + (m^2 m)^2 + \cdots + (m^{n-1} m)^2) = O\left(m^2 \frac{1 - m^{2n}}{1 - m^2}\right) = O(m^{2n})$$

and

$$O(mm + (2m)m + \cdots + ((n-1)m)m) = O(n^2 m^2)$$

respectively.

The inner join of $F^{\max}$ is over at most $k = |\Sigma||Q_P||Q_\Pi|2^{|\Sigma|}$ antichains of size ≤ 1, and is therefore computed in $O(k^2)$ and computes an antichain with size k. The outer meet is over $|\Sigma|!$ antichains produced by the inner join, and thus has complexity $O(k^{2|\Sigma|!})$. Therefore $F^{\max}$ can be computed in $O(k^{2|\Sigma|!} k^2) = O(k^{2|\Sigma|!+2})$. This brings the total complexity of our fixpoint algorithm to $O(|Q_P|^2 |Q_\Pi|^2 2^{|\Sigma|} k^{2|\Sigma|!+2})$. Since $|\Sigma|$ is typically linear in the program size, we simplify this to $O(|P||\Pi|(|P|^2|\Pi|2^{|P|})^{2|P|!+3})$, where $|P|$ is the program size and $|\Pi|$ is the proof size.

Counterexamples. Theorem 5.5 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 $\text{inactive}(M)$ 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.*

Proof. From the proof of Lemma 7.3 we have

$$\begin{aligned}
&F_{M_{P\Pi}}(X) \\
&= \{\, ((q_P, \bot, S), q_\Pi) \mid \\
&\quad \forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&\quad \exists a \notin S.\, ((q'_P, \bot, (S \cup R(a)) \setminus D(a)), q'_\Pi) \in X \}
\end{aligned}$$

where

$$q'_P = \delta_P(q_P, a) \qquad\qquad q'_\Pi = \delta_\Pi(q_\Pi, a)$$

Thus, if we have states $q' = ((q_P, \bot, S'), q_\Pi)$ and $q = ((q_P, \bot, S), q_\Pi)$ with $S \subseteq S'$ such that $q' \in \text{inactive}(M_{P\Pi})$ (and therefore $q \in \text{inactive}(M_{P\Pi})$), our witness for q' is a function $f : \mathcal{L}in(R) \to \Sigma$ such that

$$\begin{aligned}
&\forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&f(R) \notin S' \wedge ((q'_P, \bot, (S' \cup R(a)) \setminus D(a)), q'_\Pi) \in \text{inactive}(M_{P\Pi})
\end{aligned}$$

We must show

$$\begin{aligned}
&\forall R \in \mathcal{L}in(R).\, (q_P \in F_P \implies q_\Pi \in F_\Pi) \implies \\
&f(R) \notin S \wedge ((q'_P, \bot, (S \cup R(a)) \setminus D(a)), q'_\Pi) \in \text{inactive}(M_{P\Pi})
\end{aligned}$$

which is indeed the case since $f(R) \notin S' \implies f(R) \notin S$ and $(S \cup R(a)) \setminus D(a) \subseteq (S' \cup R(a)) \setminus D(a)$. □

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 EXPERIMENTS

8.1 Implementation

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 5.5. (3) *Bounded Progress-Ensuring:* selects a limited subset of the set computed by the progress-ensuring algorithm. We experimented with multiple strategies for the selection of the limited subset:

- *RR:* selects a lockstep trace of the program. In other words, it selects a single counterexample by choosing successive statements from each thread in a round robin fashion, e.g. the first statement is from thread 1, the second statement is from thread 2, and so on.
- *Ln:* selects the n leftmost counterexamples from the tree of counterexamples. When $n = 1$, this counterexample strategy effectively chooses sequential composition traces of the program.
- *Mn:* selects the n middlemost counterexamples from the tree of counterexamples. The middlemost counterexamples do not always contain lockstep counterexamples, so this strategy is distinct from the RR strategy.

Fixing a counterexample strategy does not mean committing to a particular reduction. For example, it is possible for assertions learned from a sequential composition trace to generalize to lockstep traces. In some of our benchmarks, we found that our algorithm converged using the L1 strategy, despite the fact that no full proof exists for the sequential composition reduction.

Our experimentation demonstrated that in all 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. Weaver and all our benchmarks can be found at github.com/weaver-verifier/weaver.

8.2 Benchmarks

We use a set of sequential benchmarks from [23] and include additional sequential benchmarks that involve more interesting reductions in their proofs. We also 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 proof checking 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. The set of hypersafety properties used in all our experimental results are depicted in Figure 5.

Property	Formula	Description
CompSymm	$\forall a, b, c.\, \mathrm{sign}(P(a, b)) = -\mathrm{sign}(P(b, a))$	Comparator symmetry
CompTrans	$\forall a, b, c.\, P(a, b) > 0 \land P(b, c) > 0 \implies P(a, c) > 0$	Comparator transitivity
CompSubst	$\forall a, b.\, P(a, b) = 0 \implies \mathrm{sign}(P(a, c)) = \mathrm{sign}(P(b, c))$	Comparator resp. equality
Symm	$\forall a, b.\, P(a, b) \implies P(b, a)$	Symmetry
Trans	$\forall a, b, c.\, P(a, b) \land P(b, c) \implies P(a, c)$	Transitivity
Dist	$\forall a, b, c.\, P(a + b, c) = P(a, c) + P(b, c)$	Distributivity
Sec	$\forall h_1, h_2, l.\, P(h_1, l) = P(h_2, l)$	Information flow security
Det	$\forall x, x'_1, x'_2.\, (x, x'_1) \in P \land (x, x'_2) \in P \implies x'_1 = x'_2$	Determinism
Equiv	$\forall a.\, P_1(a) = P_2(a)$	Equivalence

Fig. 5. Hypersafety properties for our experiments. In the case of Det, P is a relation between inputs and outputs rather than a function.

Benchmarks from [23]. Each example implements a comparator function that returns a negative number if the first argument is less than the second, a positive number if the first argument is greater than the second, and zero if they are equal. We check that each comparator satisfies CompSubst, CompTrans, and CompSymm.

Our Sequential Benchmarks. We verify the Mult example given in Section 2 satisfies Dist (i.e. $\textsc{Mult}(a + b, c) = \textsc{Mult}(a, c) + \textsc{Mult}(b, c)$). Since Mult iterates on its first argument only, we also verify that the property holds when its arguments are flipped (i.e. $\textsc{Mult}(c, a + b) = \textsc{Mult}(c, a) + \textsc{Mult}(c, b)$). We also verify that an array equality procedure ArrayEq satisfies symmetry and transitivity, and include a simple information flow security example. Lastly, we include examples of loop unrolling: each UnrollN example involves a loop iterating a multiple of N times, and each UnrollCondN example unrolls a loop N times, with cleanup code for extra iterations.

Our Parallel Benchmarks. We use the following parallel benchmarks:

- The Barrier example checks that a simple loop-free barrier computation is deterministic.
- The Lamport example verifies the correctness of a locking algorithm by checking whether executing a non-atomic operation in two threads within the confines of the lock is equivalent to executing the operation twice sequentially. The lock is implemented using Lamport's bakery algorithm.
- The ParallelSum examples implement parallel summation over a queue. The ParallelSum1 example has two threads atomically sum into a shared variable, while each thread in the ParallelSum2 example sums into a local variable. We verify that they are deterministic. We also verify that ParallelSum1 is equivalent to a single-threaded version of the same program.
- The SimpleInc example verifies that atomically incrementing an integer in two different threads is equivalent to non-atomically incrementing it twice in a single thread. The Spaghetti example verifies the determinism of a program that performs an arbitrary computation before setting its output to a fixed value.
- The ExpNxM benchmarks verify that a program computing an exponential term is deterministic. Each program is replicated over N threads, and each thread contains M statements.

8.3 Evaluation

Detailed results of our experiments are included in Appendix A. 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 [23] 2-safety properties	5	63	12	46.69s	0.1s	47.03s
Looping programs of [23] 3-safety properties	8	155	22	475.78s	11.79s	448.36s
Loop-free programs of [23]	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.

The bounded progress ensuring algorithm achieves its best time, in most cases, by taking 1 counterexample at each refinement round. There are two exceptions which respectively require 5 counterexamples from the left (the parallel sum) and middle (transitivity of integer array comparator) per refinement round for the best overall verification time. The former is negligible because the 1 counterexample variation produces a runtime within 12% of this optimal answer. The latter is significant, since other instances take substantially more time to verify.

In the vast majority of the cases, choosing a counterexample from the left or the middle makes a negligible difference in the verification time. The exceptions are the hardest sequential instance and the hardest parallel one. In these two cases, a left choice leads to a timeout while the middle one succeeds.

To gauge the effect of the counterexample selection strategy on the overall performance of our approach, we used all three counterexample selection modes on all our benchmarks: RR, Ln, and Mn , for $n \in \{1, 5, 10\}$. In most cases, the fastest configuration is 1-2x faster than the next fastest. For the more complicated benchmarks, the difference is much larger; for example, Weaver converges 13x faster using the RR compared to M1 counterexample mode for the substitutivity of the integer array comparator.

Our conclusion is that in the cases that the proof is complicated and can grow large, the choice of counterexamples can have a big impact on verification time. One can simply run a few instantiations of the algorithm with different counterexample choices in parallel and wait for the best one to return.

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 [23]. Note that unlike the approach in [23], 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.

Antichains. To measure the effectiveness of our antichain optimization, we compare the times for proof checking of the final (correct) proof for all our benchmarks with and without the antichain optimization. We discard the cases where the proof checking time is under 0.01s without optimization as the times are too small to be statistically relevant. The antichain-based algorithm outperforms the unoptimized one in all these instances. Of the remaining (27 benchamrks), the minimum speedup is a factor of 2, and the maximum is a factor of 44 (not considering the one instance that times out without the optimization). The average speedup is 12, and larger speedups (on average) are observed for the parallel benchmarks compared to the sequential ones.

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 [19], non-interference [15, 22], and program optimization [24]. 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, 27] 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 [23] 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 [18] 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 [14]. Other techniques exist [1, 14] 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] 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)

[15] Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy, 1982 IEEE Symposium on. pp. 11–11. IEEE (1982)

[16] Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: International Static Analysis Symposium. pp. 69–85. Springer (2009)

[17] 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)

[18] Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Communications of the ACM **18**(12), 717–721 (1975)

[19] 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)

[20] 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)

[21] Pottier, F.: Lazy least fixed points in ml

[22] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications **21**(1), 5–19 (2003)

[23] Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: ACM SIGPLAN Notices. vol. 51, pp. 57–69. ACM (2016)

[24] 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)

[25] Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: International Static Analysis Symposium. pp. 352–367. Springer (2005)

[26] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information & Computation **115**(1), 1–37 (1994)

[27] Yang, H.: Relational separation logic. Theoretical Computer Science **375**(1-3), 308–334 (2007)

A RESULTS

Benchmarks were run on a Proliant DL980 G7 with eight eight-core Intel X6550 processors (64 cores, 64 threads) and 256G of RAM, running 64-bit Ubuntu.

Benchmark	Property	Algorithm	Proof Size	Refinement Rounds	Proof Construction Time	Proof Checking Time	Total Time
Our Sequential Benchmarks							
ArrayEq	Symm	BPE(RR)	17	10	1.17s	0.16s	1.38s
ArrayEq	Trans	BPE(M5)	97	15	156.17s	30.71s	201.2s
Mult	Dist	BPE(M1)	21	11	1.34s	1.32s	2.94s
Mult	Dist (Flipped)	BPE(L1)	27	12	2.54s	1.38s	4s
Security	Sec	BPE(RR)	12	6	0.41s	0.052s	0.49s
Unroll2	Equiv	BPE(RR)	21	9	1.42s	0.017s	1.48s
Unroll3	Equiv	BPE(RR)	25	9	1.99s	0.02s	2.06s
Unroll4	Equiv	BPE(L1)	31	11	3.31s	0.05s	3.43s
Unroll5	Equiv	BPE(L1)	36	12	4.87s	0.06s	5.01s
UnrollCond2	Equiv	BPE(RR)	22	7	0.85s	0.01s	0.88s
UnrollCond3	Equiv	BPE(RR)	30	9	2.28s	0.02s	2.35s
UnrollCond4	Equiv	BPE(L1)	38	9	4.7s	0.02s	4.8s
UnrollCond5	Equiv	BPE(L1)	45	10	7.06s	0.02s	7.18s
Our Parallel Benchmarks							
Barrier	Det	BPE(RR)	21	6	4.05s	1.98s	6.1s
Lamport	Equiv	BPE(RR)	28	8	3.24s	0.44s	3.77s
ParallelSum1	Equiv	BPE(M5)	32	7	6.01s	0.19s	6.45s
ParallelSum1	Det	BPE(RR)	37	13	10.38s	0.22s	10.72s
ParallelSum2	Det	BPE(RR)	86	15	101.89s	1.07s	103.23s
SimpleInc	Equiv	BPE(M1)	7	2	0.05s	0.0008s	0.06s
Spaghetti	Det	BPE(RR)	4	2	0.03s	0.02s	0.05s
Stress Tests							
Exp1x3	Det	BPE(L1)	10	5	0.13s	0.005s	0.15s
Exp2x3	Det	BPE(L1)	17	8	0.88s	3.03s	3.95s
Exp2x4	Det	BPE(RR)	18	8	1.11s	4.23s	5.39s
Exp2x6	Det	BPE(RR)	19	8	1.32s	8.26s	9.65s
Exp2x9	Det	BPE(RR)	19	8	1.34s	19.19s	20.61s
Exp3x3	Det	BPE(L5)	26	11	3.77s	1672.05s	1676.61s

Table 2. The detailed results of the winning algorithm performed on our benchmarks. TO = total-order, P = partition, N = naive, BPE(?n) = bounded-progress-ensuring (with n counterexamples from left (L), middle (M), or round-robin (RR)).

Benchmark	Property	Algorithm	Proof Size	Refinement Rounds	Proof Construction Time	Proof Checking Time	Total Time
ArrayInt	CompSubst	BPE(RR)	146	20	517.33s	5.22	523.82
ArrayInt	CompSymm	BPE(RR)	29	8	6.21s	0.02s	6.36s
ArrayInt	CompTrans	BPE(RR)	148	21	390.15s	2.27s	393.66s
Chromosome	CompSubst	BPE(L1)	136	18	211.34s	11.39s	223.58s
Chromosome	CompSymm	BPE(RR)	99	16	61.68s	0.15s	62.2s
Chromosome	CompTrans	BPE(L1)	179	23	274s	16.56s	291.22s
CollItem	CompSubst	BPE(M1)	5	2	0.5s	0.0004s	0.55s
CollItem	CompSymm	BPE(M1)	4	2	0.09s	0.0003s	0.12s
CollItem	CompTrans	BPE(M1)	5	2	0.21s	0.0003s	0.24s
Container	CompSubst	BPE(M1)	5	2	0.39s	0.0004s	0.43s
Container	CompSymm	BPE(M1)	4	2	0.09s	0.001s	0.12s
Container	CompTrans	BPE(M1)	5	2	0.22s	0.0005s	0.25s
ExpTerm	CompSubst	BPE(L1)	5	2	0.16s	0.0005s	0.18s
ExpTerm	CompSymm	BPE(M1)	4	2	0.07s	0.0003s	0.08s
ExpTerm	CompTrans	BPE(M1)	4	2	0.08s	0.0003s	0.09s
FileItem	CompSubst	BPE(RR)	5	2	0.14s	0.0004s	0.16s
FileItem	CompSymm	BPE(RR)	4	2	0.07s	0.0003s	0.08s
FileItem	CompTrans	BPE(L1)	5	2	0.12s	0.0004s	0.13s
Match	CompSubst	BPE(RR)	5	2	0.11s	0.0004s	0.12s
Match	CompSymm	BPE(M1)	4	2	0.06s	0.0005s	0.07s
Match	CompTrans	BPE(RR)	5	2	0.07s	0.0004s	0.08s
NameComparator	CompSubst	BPE(L1)	75	14	70.41s	0.37s	71.03s
NameComparator	CompSymm	BPE(L1)	49	9	15.5s	0.026s	15.66s
NameComparator	CompTrans	BPE(L1)	114	18	114.88s	3.56s	118.84s
Node	CompSubst	BPE(L1)	5	2	0.27s	0.0004s	0.3s
Node	CompSymm	BPE(L1)	4	2	0.06s	0.0002s	0.08s
Node	CompTrans	BPE(RR)	5	2	0.1s	0.0004s	0.11s
NzbFile	CompSubst	BPE(RR)	298	33	2026.73s	28.51s	2056.9s
NzbFile	CompSymm	BPE(RR)	111	19	144.96s	0.27s	145.72s
NzbFile	CompTrans	BPE(RR)	219	29	751.71s	43.97s	796.54s
SimplStr	CompSubst	BPE(RR)	5	2	0.12s	0.0003s	0.13s
SimplStr	CompSymm	BPE(L1)	4	2	0.03s	0.0002s	0.04s
SimplStr	CompTrans	BPE(RR)	5	2	0.07s	0.0003s	0.07s
Sponsored	CompSubst	BPE(RR)	5	2	0.07s	0.0003s	0.08s
Sponsored	CompSymm	BPE(M1)	4	2	0.03s	0.0002s	0.03s
Sponsored	CompTrans	BPE(M1)	5	2	0.03s	0.0003s	0.04s
Time	CompSubst	BPE(M1)	5	2	0.17s	0.0005s	0.19s
Time	CompSymm	BPE(M1)	4	2	0.03s	0.0005s	0.04s
Time	CompTrans	BPE(M1)	5	2	0.07s	0.0004s	0.08s
Word	CompSubst	BPE(RR)	103	18	179.29s	2.59s	182.28s
Word	CompSymm	BPE(RR)	26	8	5.14s	0.017s	5.22s
Word	CompTrans	BPE(RR)	129	22	221.94s	3.48s	225.78s

Table 3. The detailed results of the winning algorithm performed on benchmarks from [23]. TO = total-order, P = partition, N = naive, BPE(?n) = bounded-progress-ensuring (with *n* counterexamples from left (L), middle (M), or round-robin (RR))

Benchmark	Property	Threads	Proof Size	Optimized Time	Unoptimized Time
Our Sequential Benchmarks					
ArrayEq	Symm	2	17	0.03s	0.13s
ArrayEq	Trans	3	97	5.99s	75.53s
Mult	Dist	3	21	0.42s	5.21s
Mult	Dist (Flipped)	3	27	0.33s	3.11s
Security	Sec	2	12	0.01s	0.06s
Unroll2	Equiv	2	21	0.003s	0.007s
Unroll3	Equiv	2	25	0.005s	0.01s
Unroll4	Equiv	2	31	0.01s	0.05s
Unroll5	Equiv	2	36	0.01s	0.05s
UnrollCond2	Equiv	2	22	0.002s	0.003s
UnrollCond3	Equiv	2	30	0.003s	0.005s
UnrollCond4	Equiv	2	38	0.007s	0.02s
UnrollCond5	Equiv	2	45	0.006s	0.01s
Our Parallel Benchmarks					
Barrier	Det	4	21	0.47s	11.46s
Lamport	Equiv	3	28	0.09s	0.36s
ParallelSum1	Equiv	3	32	0.09s	1.10s
ParallelSum1	Det	4	37	0.12s	1.17s
ParallelSum2	Det	4	86	0.1s	1.21s
SimpleInc	Equiv	3	7	0.0004s	0.001s
Spaghetti	Det	4	4	0.004s	2.28s
Stress Tests					
Exp1x3	Det	2	10	0.002s	0.005s
Exp2x3	Det	4	17	1.15s	27.2s
Exp2x4	Det	4	18	1.33s	36.009s
Exp2x6	Det	4	19	2.34s	80.98s
Exp2x9	Det	4	19	5.045s	222.45s
Exp3x3	Det	6	26	910.03s	T/O
Benchmarks from [23]					
ArrayInt	CompSubst	3	146	0.54s	2.47s
ArrayInt	CompSymm	2	29	0.004s	0.005s
ArrayInt	CompTrans	3	148	0.13s	0.71s
Chromosome	CompSubst	3	136	2.81s	32.63s
Chromosome	CompSymm	2	99	0.02s	0.02s
Chromosome	CompTrans	3	179	3.25s	24.55s
NameComparator	CompSubst	3	75	0.02s	0.31s
NameComparator	CompSymm	2	49	0.004s	0.009s
NameComparator	CompTrans	3	114	0.22s	1.79s
NzbFile	CompSubst	3	298	6.25s	34.38s
NzbFile	CompSymm	2	111	0.041s	0.082s
NzbFile	CompTrans	3	219	9.39s	50.74s
Word	CompSubst	3	103	0.42s	2.11s
Word	CompSymm	2	26	0.005s	0.007s
Word	CompTrans	3	129	0.39s	2.038s

Table 4. The detailed results of the last round of proof checking for all benchmarks, with and without the antichain optimization. The value of the Threads column is the number of threads in the encoded program. Benchmarks whose proofs take less than 0.01 seconds to check are omitted.