Reductions for Safety Proofs [*]

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

Program reductions are used widely to simplify reasoning about the correctness of concurrent and distributed programs. In this paper, we propose a general approach to proof simplification of concurrent programs based on exploring *generic* classes of reductions. We introduce two classes of sound program reductions, study their theoretical properties, show how they can be effectively used in algorithmic verification, and demonstrate that they are very effective in producing proofs of a diverse class of programs without targeting specific syntactic properties of these programs. The most novel contribution of this paper is the introduction of the concept of *context* in the definition of program reductions. We demonstrate how *commutativity* of program steps in some program contexts can be used to define a generic class of sound reductions which can be used to automatically produce proofs for programs whose complete Floyd-Hoare style proofs are theoretically beyond the reach of automated verification technology of today.

CCS Concepts: • **Theory of computation → Automated reasoning**; *Verification by model checking*; Automata over infinite objects; Tree languages.

Additional Key Words and Phrases: Concurrency, Automated Verification, Reductions, Automata

ACM Reference Format:
Azadeh Farzan and Anthony Vandikas. 2020. Reductions for Safety Proofs . *Proc. ACM Program. Lang.* 4, POPL, Article 13 (January 2020), 28 pages. https://doi.org/10.1145/3371081

13

1 INTRODUCTION

A *reduction* of a program is generally another program, with a subset of the behaviours of the original program, that faithfully represents it. Program reductions have been studied extensively [Desai et al. 2014; Elmas et al. 2009; Genest et al. 2007; Hawblitzel et al. 2015; Lipton 1975; von Gleissenthall et al. 2019] in the context of simplifying reasoning about concurrent and distributed programs. The earliest and perhaps most well-known approach to reduction is due to Lipton [Lipton 1975] who proposed to simplify concurrent program proofs by inferring large atomic blocks of code (when possible) in order to reap the benefits of sound sequential reasoning inside these blocks. The inference of the large atomic blocks is carried out based on commutativity specifications of individual program statements. In the past 40 years, Lipton's work has inspired many reduction schemes for concurrent program analysis [Flanagan and Qadeer 2003; Flanagan et al. 2005] and verification [Elmas et al. 2009; Hawblitzel et al. 2015]. In a different context, commutativity specification of program statements have been used for an entirely different type of reduction. There, the aim is to reduce the sizes of the communication buffers used in message-passing programs. The equivalent program with the smallest buffer sizes can be viewed as an *almost synchronous* variation of the original asynchronous program. The key insight is that the proof of

[*]Formal proofs of all theorems can be found in the extended version [Farzan and Vandikas 2019b].

Authors' addresses: Azadeh Farzan, University of Toronto, Canada; Anthony Vandikas, University of Toronto, Canada.

2475-1421/2020/1-ART13
https://doi.org/10.1145/3371081

correctness for the synchronous program is *simpler*; for programs with bounded buffers the proof need not include complex invariants such as those that universally quantify over unbounded buffer contents.

The two groups of reduction approaches strive for seemingly contradictory targets. Lipton's approach opts for reductions in which the threads try not to *yield* for as long as possible, while synchronous reductions would force a yield right after each *send* operation in order to execute its matching *receive*. The former seems appropriate for shared memory concurrent programs and the latter for message-passing concurrent and distributed programs. This sparks several interesting questions: is this truly a rigid dichotomy? Can shared memory concurrent programs benefit from certain types of synchronous reductions where arbitrary program statements (other than just sends and receives on channels) are synchronized? What sort of reductions can deal with message-passing concurrent programs where reasoning has to be extended to the part of the program that manipulates the data? Can we not commit to a particular reduction scheme in advance and let the verifier pick the ideal reduction for the input program, depending on what is required for the the specific combination of the program and the property? In this paper, we provide some initial answers to these questions.

We propose an automated verification approach that combines the search for a proof with the search for a sound reduction of the program. The high level idea is to give automated verification a chance to succeed by finding a correctness proof for a reduction of the program, where attempting to prove the original program correct would otherwise fail. The key distinction with regards to most of the relevant literature is that instead of fixing a particular reduction in advance, we propose to let a new automated verification algorithm search for an ideal reduction within a generic universe of (infinitely many) sound reductions. The simple insight is that committing to the wrong reduction in advance, for example attempting to infer large atomic blocks for a distributed message-passing program, could set one up for failure. Our target programs are principally those where *proof simplification is the difference between the existence and nonexistence of a safety proof* within a fixed (decidable) language of assertions commonly used in automated verification. Without simplification, the proof involves complicated invariants with elements such as quantification over arrays and buffers or non-linear arithmetic for data variables which are currently the Achilles heel of automated verification techniques. Therefore, the main accomplishment of our methodology is to leverage the proof simplification power of reductions to expand the reach of automated verification to instances that are theoretically out of its scope.

Our refinement loop maintains a proof candidate at each round, and checks if there exists a reduction of the input program that is proved correct by this proof. To be able to implement this subsumption test algorithmically, one needs an effective way of representing the set of all program reductions. We introduce two novel classes of (infinitely many) program reductions and use finite state (tree) automata, with nice algorithmic properties, to represent each class. The first class is inspired by semi-trace monoids [Diekert and Rozenberg 1995] defined by a semi-commutativity relation between program statements. Unfortunately, checking whether a proof subsumes a reduction of the program according to such a semi-trace monoid is in general undecidable (more on this in Section 4). Therefore, the contribution of this paper critical to algorithmic verification is devising a subclass, which we call *S-reductions*, with a decidable subsumption check.

The most significant contribution of this paper is the second proposed class of reductions, namely *contextual reductions*. For this class, the commutativity properties of the program statements depend on the context from which the corresponding statements are executed. Two statements may commute in one *context* and not in another. Contexts have been exploited in special cases for proofs before. For example, in message-passing programs, a *receive* can be commuted to the left of a *send* operation that is not its matching *send*, determined by the context.

To the best of our knowledge, general contexts have never been exploited for program proofs before, and certainly not for automated verification.

Inspired by a language-theoretic notion of context from *generalized Mazurkiewicz traces*, we define a set of (infinitely many) contextual reductions that is recognized by finite state automata. The elegance of this definition is that it does not commit to a particular contextual commutativity relation in advance. The automaton models a universe of reductions based on a universe of contextual commutativity specifications. Our proposed algorithm then decides if there exists a sound contextual commutativity specification in this universe, which induces a sound contextual reduction of the program that is covered by a current valid proof candidate. Therefore, beyond making progress in proof construction, the refinement loop also infers assertions that substantiate the soundness of a larger contextual commutativity relation through refinement. This goes against the classical approach to automated program verification where one first chooses a (static) mostly non-contextual commutativity specification, *then* a reduction induced by the chosen specification, and *finally then* tries to search for a proof for the given reduction. Our proposed refinement loop performs all these three searches simultaneously. In summary, the following are the contributions of this paper:

- We introduce a class of semi-commutative reductions and present the theoretical properties of this class that make it a good candidate for algorithmic proof simplification (Section 4).
- We introduce a novel class of contextually commutative reductions essential to proof simplification for both shared-memory and message-passing concurrent programs, and theoretically argue why they are suitable for algorithmic use (Section 5).
- We present a counterexample-guided refinement loop for verification which can incorporate the above classes of reductions. This algorithm effectively performs a search for a triple consisting of *a contextual commutativity relation, a program reduction induced by it, and a proof of correctness for the reduction*. We discuss the soundness, completeness, and convergence conditions for the algorithm. Moreover, we present two interesting insights that accommodate the development of a novel algorithm for proof checking with an improved time complexity upper bound. (Sections 7 and 8).
- We provide an in-depth comparison of the reductions presented in this paper and the two most well-known reduction schemes from the concurrent program verification literature (Section 6): (1) Lipton's reduction for the inference of large atomic blocks, and (2) reductions based on the idea of existential boundedness [Genest et al. 2007] which use commutativity-based transformations to reduce message buffer sizes to simplify proofs of message-passing concurrent/distributed programs.
- Our approach is implemented in a tool, called Siever . Using a rich set of benchmarks, mostly with required invariants beyond the reach of previous automated verification tools, we demonstrate how the technique is effective in producing (automatically generated) proofs for these benchmarks (Section 9).

2 MOTIVATING EXAMPLES

We start by motivating the two classes of reductions proposed in this paper through two examples. In our first example, proof simplification is not essential, in that a proof for the program exists and can be discovered using a standard verification algorithm [Heizmann et al. 2009]. By using the class of semi-commutative reductions in this paper, however, one can produce a simpler proof (about half the number of distinct assertions in the proof) in less than one third of the time. We then make a small modification to the code to get our second example, for which a proof for the whole program does not exist in the decidable assertion language of linear integer arithmetic (LIA). Moreover, even though the program does admit a semi-commutative reduction, a proof does not exist for that reduction either. This will motivate our *contextual reduction* class, which includes a sound reduction of the program with a simpler proof that is quickly discovered by SIEVER .

Consider the simple methods `inc()` and `dec()` defined in Figure 1(a,b), and a simple concurrent program using them listed in Figure 1(c), along with its corresponding pre/post-conditions. Note that `dec()` is a blocking statement and therefore not all program runs terminate. For a safety proof, it suffices to show that those that do, satisfy the given pre/post-condition. It is straightforward to see that a full Floyd-Hoare style proof of this program exists in the decidable logical language of linear integer arithmetic.

```
inc(x: int)
{
  atomic {
    x = x + 1
  }
}
```
(a)

```
dec(x: int)
{
  atomic {
    assume(x > 0);
    x = x - 1;
  }
}
```
(b)

Precondition: $\{N = M \land y = 0\}$

```
i = 0;                  ||  j = 0;
while (i < N) {         ||  while (j < M) {
   inc(y);              ||     dec(y);
   i++;                 ||     j++;
}                           }
```

Postcondition: $\{y = 0\}$ (c)

Fig. 1. Semi-commutativity example.

Observe that `inc()` soundly semi-commutes with `dec()` in the sense that it is sound to swap a `dec()` statement to the right of a following `inc()` statement, without changing program behaviour. The inverse, however, is not true. Swapping a decrement to the left of an increment *may* make it block in some program runs where it was not blocking in its original position. Adding this to the fact that `i` and `j` are thread-local and therefore all statements referencing them commute (against the statements of the other thread), indicates that a full proof for the program is not strictly necessary. It is sufficient to provide a proof for a subset of the program runs that soundly represent the program, and this subset may admit a strictly simpler proof. The discovery of this simple proof is the goal of the methodology presented in this paper.

```
i = 0;
while (i < N) {
   inc(y);
   i++;
}
j = 0;
while (j < M) {
   dec(y);
   j++;
}
```

Fig. 2. A reduction.

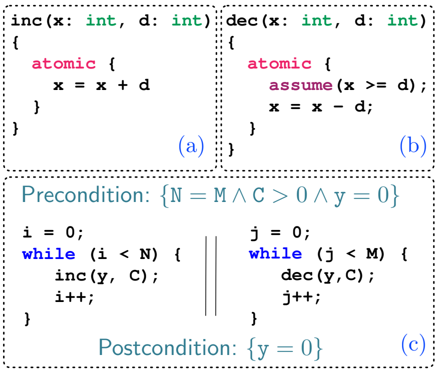
The sequential program illustrated in Figure 2 is a sound *reduction* of the program in Figure 1. In this reduction, *all* decrements are postponed to the end using the semi-commutativity of `dec()` and `inc()`. The full commutativity of the rest of the actions is then used to bring relevant steps of each thread together. Our proposed set of *S-reductions* includes this sequential program as well as (infinitely) many other reductions that are equivalent to the program up to the aforementioned (semi-) commutativity properties of the statements. Our proposed algorithm attempts to verify at least one member of the entire set in a refinement loop. Our tool, SIEVER , discovers a simpler proof (about half the number of distinct assertions in the proof) in about a third of the time of the original (without reductions). Note that the reduction, for which SIEVER discovers a proof, may not be the one in Figure 2.

The reader familiar with Lipton's reductions [Lipton 1975] and the concept of left/right movers would be curious about the connection between this transformation and Lipton's atomic blocks reductions. Note that `dec()` is a right-mover, and respectively, `inc()` is a left mover, and every other statement is both [1]. Therefore, one can soundly declare each thread as one atomic block and end up with a reduced program that runs these two atomic blocks in parallel. This program has additional behaviours compared to the (sequentialized) reduction of Figure 2. The difference is not substantial in this case. Next, we will look at a slight modification of this program which would render both Lipton style reductions and our S-reductions entirely useless for proof simplification.

Consider the modified code illustrated in Figure 3. The methods `inc()` and `dec()` operate as before, but now take an extra parameter determining the increment/decrement *delta*. The program uses a global (uninitialized) positive constant `C` as this delta, and otherwise operates as before. Note

[1]In fact, since Lipton's original definition in [Lipton 1975] is quantified over all reachable program contexts, `inc()` and `dec()` would be both-movers according to his original definition. But, folklore usage of his technique, which quantifies over all contexts (reachable or not), would declare them only left and right mover respectively.

that this program admits the same sequential reduction in the style of Figure 2, since the new `inc()` and `dec()` methods satisfy the same (semi-) commutative properties as in the previous example. The problem is, however, that this sequential reduction does not admit a proof in the decidable LIA fragment. The proof needs to establish at the end of the first loop that $\mathsf{y} = \mathsf{N} \times \mathsf{C}$, so that by the end of the second loop, `y` can be proved to go back to zero. This requires a non-linear loop invariant $\mathsf{y} = \mathsf{i} \times \mathsf{C}$ for the first loop. Lipton's reductions are also not effective for the same reason. Luckily, there is another reduction of this program that does admit a proof in the LIA fragment.



```
inc(x: int, d: int)
{
  atomic {
    x = x + d
  }
}
```
(a)

```
dec(x: int, d: int)
{
  atomic {
    assume(x >= d);
    x = x - d;
  }
}
```
(b)

Precondition: $\{\mathrm{N} = \mathrm{M} \land \mathrm{C} > 0 \land \mathrm{y} = 0\}$

```
i = 0;                   ||   j = 0;
while (i < N) {          ||   while (j < M) {
   inc(y, C);            ||      dec(y,C);
   i++;                  ||      j++;
}                        ||   }
```

Postcondition: $\{\mathrm{y} = 0\}$ (c)

Fig. 3. Contextual commutativity example.

Any program trace that has a different number of increments and decrements is infeasible, and this can be reflected in the proof by simple invariants relating `i`, `j`, `M`, and `N`. All feasible program traces will have an equal number of increments and decrements. To avoid having to use multiplicative assertions like $\mathsf{y} = \mathsf{N} \times \mathsf{C}$, of all the equivalent feasible interleavings of the program, the one in which increments and decrements appear in alternate order is the preferred one:

`inc(y,C)` ... `dec(y,C)` ... `inc(y,C)` ... `dec(y,C)` ....

For these interleavings, the invariants need to only capture the fact that `y` goes up by `C` and then comes back to zero when the matching decrement happens. The reduction that only includes these interleavings is in some sense *the opposite* of the sequential reduction of Figure 2. In the sequential one, an entire thread is executed as an atomic block, while in this one, threads are forced to yield after each increment to let the matching decrement execute. It is interesting how a small change in the program can have a big impact on the appropriate reduction for proving it correct.

Let us now argue why the suggested reduction is sound. The key is the concept of *contextual commutativity*. Note that `inc(y)` and `dec(y)` fully commute if $\mathsf{y} \geq \mathsf{C}$. Only for values of $\mathsf{y} < \mathsf{C}$, do they semi-commute (as discussed above). Under this contextual commutativity relation, one can show that the interleaving proposed above is equivalent to all other interleavings of the program. The high level argument is: we already know that all decrements can be postponed to the end, due to the (non-contextual) semi-commutativity relation. Therefore, every interleaving is equivalent to one with all the decrements appearing at the end. Starting from that interleaving, the decrements can be pulled forward one by one to appear next to a (matching) increment, because we know $\mathsf{y} \geq \mathsf{C}$ is true before each decrement (that has a matching increment in the prefix of the run). Note that this last step cannot be performed under the static semi-commutativity assumption. A decrement does not commute to the left of an increment.

The main observation is that *contexts matter*. At the beginning, before any increments or decrements have been executed, the two operations do not commute (when $\mathsf{y} = 0$). Once an increment is executed, then $\mathsf{y} \geq \mathsf{C}$ is established and then the operations commute.

Our proposed set of contextual reductions, called *C-reductions*, includes this preferred reduction and (infinitely) many more equivalent ones. In a refinement loop, our algorithm infers such contextual commutativity information, and uses it to discover a sound contextual reduction of the program that can be proved correct. The proof is in the pudding: the algorithm decides which reductions are sound and among those which can be proved correct by actually producing proofs of soundness of reductions and correctness of at least one specific reduction. SIEVER can discover a proof for the program in Figure 3 in a few seconds.

3 BACKGROUND

3.1 Programs and Proofs

Programs as Regular Languages. $\mathcal{S}t$ denotes the (possibly infinite) set of *program states*. For example, we have $\mathcal{S}t = \mathbb{Z}\times\mathbb{Z}$ for a program with two integer variables. Let $\mathcal{A} \subseteq \mathcal{P}(\mathcal{S}t)$ be a (possibly infinite) set of *assertions*. Σ denotes a finite alphabet of program *statements*. For multithreaded programs, statements are annotated with thread identifiers to distinguish the same statement of different threads. We assume a bounded number of threads.

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 $\tau \in \Sigma^*$ is said to be *infeasible* if $[\![\tau]\!](\mathcal{S}t) = \emptyset$, where $[\![\tau]\!](\mathcal{S}t)$ denotes the image of $[\![\tau]\!]$ under $\mathcal{S}t$. Note that the set of program traces is a superset of the set of concrete program executions (i.e. feasible program traces).

Without loss of generality, we define a *program* as a 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 the language of program traces by interpreting the edge-labelled control-flow graph of the program as a deterministic finite automaton (DFA): each location in the control flow graph is a DFA state, and each edge in the control flow graph is a DFA transition. The control flow graph entry location is the initial state of the DFA and all its exit locations are the DFA final states. We do not define programs to necessarily be *regular* languages, but we do require our input programs to be regular and many important results require this.

Program Safety. In the context of this paper, a program P is *safe* if all traces of P are infeasible, i.e. $[\![P]\!](\mathcal{S}t) = \emptyset$. Standard partial correctness specifications can be represented as safety 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 language containing it), and $\cdot$ is language concatenation.

A *proof* is defined based on a finite set of assertions $\Pi \subseteq \mathcal{A}$ that includes *true* and *false*. One can associate a regular language to each set of assertions Π by defining the NFA $\Pi_{NFA} = (\Pi, \Sigma, \delta_\Pi, \mathit{true}, \{\mathit{false}\})$ where

$$\delta_\Pi(\phi_{pre}, a) = \{\phi_{post} \mid [\![a]\!](\phi_{pre}) \subseteq \phi_{post}\}.$$

We refer to $\mathcal{L}(\Pi_{NFA})$, abbreviated as $\mathcal{L}(\Pi)$, as a *proof*. Intuitively, $\mathcal{L}(\Pi)$ consists of traces that can be proven infeasible using only assertions in Π. The following proof rule is therefore sound [Farzan et al. 2013, 2015; Heizmann et al. 2009]:

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

When $P \subseteq \mathcal{L}(\Pi)$, we say that $\mathcal{L}(\Pi)$ is a proof for P. A proof does not uniquely belong to any particular program; a single language $\mathcal{L}(\Pi)$ may prove many programs correct. When both P and $\mathcal{L}(\Pi)$ are regular, this check is decidable and polynomial on the sizes of their corresponding DFAs.

3.2 Reductions

A safe program may not admit a safety proof in a given language of assertions, or it may admit one but the proof may be prohibitively complex. This has inspired the notion of *program reductions*. The reduction of a program P is a simpler program P' that may be soundly proved safe in place of the original program P. Below is a very general definition of program reductions.

Definition 3.1 (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 sound proof rule for proving safety:

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

A program is safe if and only if $\emptyset$ is a valid reduction of the program, which means discovering a semantic reduction and proving safety are mutually reducible to each other. Therefore, verifying the existence of a *semantic reduction* is in general *undecidable*. Therefore, a very particular choice of reduction is often used [Desai et al. 2014; Lipton 1975; von Gleissenthall et al. 2019].

There are instances in the literature [Farzan and Vandikas 2019a; Lipton 1975] where a restricted class of reductions have been used instead. For example, Lipton's reductions are technically a family of choices of atomic blocks based on left/write-movers in the program. If one restricts the set of possible reductions from all reductions (given in Definition 3.1) to a proper subset which more amenable to algorithmic checking, then the rule becomes more amenable to automation. Fixing a set $\mathcal{R}$ of (semantic) reductions will change the rule to:

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

In [Farzan and Vandikas 2019a] one candidate for $\mathcal{R}$ was presented in the form of a set of syntactic reductions which are called *sleep set* reductions. In this paper, we take a major step in defining a far more general (yet decidable) set of semantic reductions as a candidate for $\mathcal{R}$.

3.3 Tree Automata for Classes of Languages

It is possible to automate the checking of the first premise of the rule SafeRed2 through automata theoretic techniques.

An infinite tree can encode a (potentially infinite) language of finite words. Consider the tree on the right where nodes are labeled with booleans and arcs are labeled with alphabet letters. A word belongs to the language represented by such a tree if it labels a path from the root to a *true* labeled node of the tree. A set of languages can then be encoded as a set of infinite trees. Certain sets of infinite trees are recognized by (finite state) automata over infinite trees. Looping Tree Automata (LTAs) are a subclass of Büchi Tree Automata where all states are accept states [Baader and Tobies 2001]. 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 [Baader and Tobies 2001].

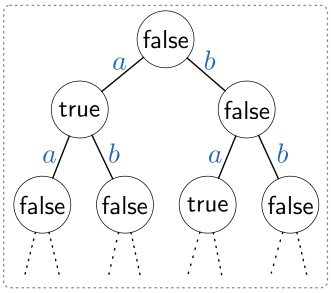


Fig. 4. Infinite tree representing the language $\{a, ba\}$.

Definition 3.2. A Looping Tree Automaton (LTA) over $|\Sigma|$-ary, $\mathbb{B}$-labelled trees is a tuple $M = (Q, \Sigma, \Delta, q_0)$ 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.

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 3.3 (from [Farzan and Vandikas 2019a]). *Given an LTA M and a regular language L, it is decidable whether $\exists P \in \mathcal{L}(M).\, P \subseteq L$.*

Note that Theorem 3.3 is effectively providing an automation recipe for the proof rule SafeRed2. In [Farzan and Vandikas 2019a], a construction was given for a Looping Tree Automaton (LTA) that recognizes a specific family of reductions, called *sleep-set reductions* of an input program P, which were shown to be useful in proving hypersafety properties of programs. In this paper, we will provide two extensions: a family of *static semi-commutative* reductions and a family of *contextual* reductions which are specifically useful for simplification of concurrent program proofs.

4 SEMI-COMMUTATIVE REDUCTIONS

We introduce a class of reductions inspired by semi-commutative *Mazurkiewicz* traces (aka semi-trace monoids) and Lipton's [Lipton 1975] left/right-movers.

Let $I \subseteq \Sigma \times \Sigma$ be an irreflexive (but not necessarily symmetric) *semi-independence relation.* Let $\sqsubseteq_I$ be the smallest preorder satisfying $\sigma ab\rho \sqsubseteq_I \sigma ba\rho$ for all $\sigma, \rho \in \Sigma^*$ and $(a, b) \in I$. The upwards and downwards closures of a language $L \subseteq \Sigma^*$ with respect to $\sqsubseteq_I$ are respectively denoted by $\lceil L \rceil_I$ and $\lfloor L \rfloor_I$ and defined as:

$$\lceil L \rceil_I = \{u \mid \exists v \in L.\, v \sqsubseteq_I u\} \qquad \lfloor L \rfloor_I = \{u \mid \exists v \in L.\, u \sqsubseteq_I v\}$$

A language L is *upwards-closed* (resp. *downwards-closed*) with respect to $\sqsubseteq_I$ if $L = \lceil L \rceil_I$ (resp. $L = \lfloor L \rfloor_I$).

If I is a symmetric relation, then $\sqsubseteq_I$ becomes an equivalence relation and its equivalence classes are known as *Mazurkiewicz traces* [Diekert and Métivier 1997]. As is the case with Mazurkiewicz traces, relation I is of interest in program verification when it is *sound*, i.e. $[\![ab]\!] \subseteq [\![ba]\!]$ for all $(a, b) \in I$.

Definition 4.1 (semi-commutative reduction). A program P' is a semi-commutative reduction of a program P, denoted by $P' \preceq_I P$, if $P \subseteq \lfloor P' \rfloor_I$.

Intuitively, in the reduction P', it is safe to remove smaller traces (with respect to $\sqsubseteq_I$) in favour of larger ones. I is *sound* if $\sigma \sqsubseteq_I \rho \implies [\![\sigma]\!] \subseteq [\![\rho]\!]$. Sound relations define sound reductions for safety verification. Formally:

Lemma 4.2. *If I is a sound semi-independence relation and $P' \preceq_I P$ then $P' \preceq P$.*

Example 4.3. Recall the example from Section 2 illustrated in Figure 1. `inc()` semi-commutes with `dec()` in the sense that it would be sound to have (`dec()`, `inc()`) $\in I$. But, the inverse is not true: (`inc()`, `dec()`) $\notin I$. The sequential program of Figure 2 is a sound semi-commutative reduction of the program in Figure 1.

Ideally, the set of all (sound) semi-commutative reductions of a program would replace $\mathcal{R}$ in the premise of the rule SafeRed2. Unfortunately, this is not possible. It has already been argued in [Farzan and Vandikas 2019a] that the premise check $\exists P' \in \mathcal{R}. P' \subseteq \mathcal{L}(\Pi)$ is undecidable for an arbitrary Π for the special case where I is symmetric. Considering our scenario is strictly more general, the undecidability result follows straightforwardly. Fortunately, there exists a suitable approximation of the set of *semi-commutative* reductions that can be used as a candidate for $\mathcal{R}$ rendering the premise decidable.

4.1 A Representable Class of Semi-Commutative Reductions

Recall from Section 3.3 that a language can be represented by an infinite labelled tree, where the arcs are labelled with program statements. To reduce a language in a constructive way (in contrast to Definition 4.1), one can prune this infinite tree in a style inspired by partial order reduction [Godefroid 1996]. Pruning the tree is equivalent to removing words from the language, which defines a reduction.

Consider the tree depicted in Figure 5(i) which corresponds to the language of all traces of a simple program $a \parallel bcd$. Assume that we have $I = \{(b, a), (d, a)\}$. Imagine a (depth-first) traversal of the tree in prefix order starting from the root. Once the left-most branch of the tree is explored, which corresponds to the program run *abcd*, the algorithm explores the right branch at the root. Here, the algorithm chooses not to explore the run *bacd*, since $(b, a) \in I$ (and therefore $bacd \sqsubseteq_I abcd$) and *abcd* has already been explored. This branch is greyed out in Figure 5(ii) to indicate that it is pruned. The algorithm continues its exploration and decides to prune *bcda* since $bcda \sqsubseteq_I bcad$ and *bcad* is explored beforehand.

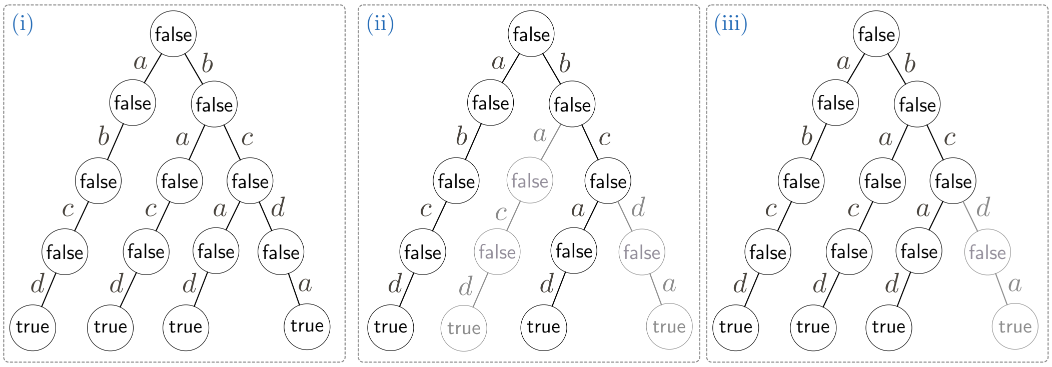


Fig. 5. An example illustrating reductions as prunings of the tree representing the program language.

Now let us slightly tweak the algorithm's traversal strategy. At the root, we choose to go right first (instead of left as before). At every other internal node, we do prefix traversal as before. Now, the algorithm sees *bacd* first, *bcad* second, and as before, prunes *bcda* since $bcda \sqsubseteq_I bcad$. This is illustrated in Figure 5(iii). Then finally, it gets to the leftmost branch from the root and explores *abcd*. Note that *abcd* cannot be pruned. We have $bacd \sqsubseteq_I abcd$, but the inverse is not true, that is $abcd \not\sqsubseteq_I bacd$. The change of the traversal strategy changes the reduction that is acquired.

A particular reduction is parametric on the non-deterministic choices made about which branch to explore first. They determine what program traces are *pruned* in favour of others visited before them which are larger with respect to $\sqsubseteq_I$. Different non-deterministic choices lead to different reductions. Two such reductions are depicted in Figure 5(ii,iii) for two different choices of exploration strategy at the root. Note that one can change the exploration strategy at every internal node (with more than one successor) to enumerate more reductions of this particular language. Reductions are then characterized by an assignment $O : \Sigma^* \to \mathcal{L}in(\Sigma)$ of nodes to linear orderings on Σ, where $(a, b) \in O(\sigma)$ means that at node σ (i.e. the node labeled by string σ from the root), we explore the child σa after the child σb. Each O combined with the semi-independence relation I defines a reduction $P{\downarrow_{I,O}}$ of the program P:

$$P{\downarrow_{I,O}} = P \setminus \{\rho a \sigma b \tau \mid \rho, \sigma, \tau \in \Sigma^* \wedge (a, b) \in O(\rho) \wedge \forall c \in a\sigma.\, (c, b) \in I\}$$

where smaller (with respect to $\sqsubseteq_I$) strings are pruned away in favour of the larger ones. If program P is upwards closed, then the aggressive pruning defined above is sound:

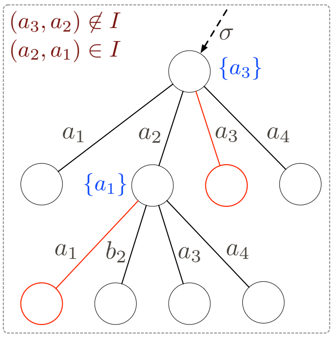
Lemma 4.4. *For all $O : \Sigma^* \to \mathcal{L}in(\Sigma)$, if P is upwards-closed then $P{\downarrow_{I,O}} \preceq_I P$.*

The set of all such reductions for a program and a fixed semi-independence relation I can then be defined by enumerating all such order relations.

Definition 4.5 (S-Reduction). For a sound semi-independence relation I and an upwards closed program P, the set of S-reductions of P is defined as

$$\text{SRed}_I(P) = \{P{\downarrow_{I,O}} \mid O : \Sigma^* \to \mathcal{L}in(\Sigma)\}.$$

The good news is that S-reductions can be effectively represented as the language of an LTA (Looping Tree Automaton) as defined in Section 3.3. The intuition behind the construction of the LTA recognizing $\mathrm{SRed}_I(P)$ is as follows. The state of the LTA keeps track of the set of transitions that can be ignored during the exploration, referred to as *sleep sets*. The idea is that the sleep set at the root of the tree is always empty, since nothing can be ignored there. The child node inherits the sleep set of the parent node, adds to it the transitions that have already been explored from the parent node (which is retrievable from $O(\sigma)$) and removes from it anything that is not semi-independent on the transition taken from the parent to the child. Ignored transitions define ignored nodes in a tree in a straightforward manner: a node is not ignored if there is a path of (all) unignored transitions to it from the root. For example, in the figure on the right, if at node σ, the transition a_3 can be ignored, then it means all the descendants of σa_3 are also ignored. If a_i's are traversed in ascending order of i's, then by the time we get to σa_2, we have already explored σa_1 and its descendants. At σa_2, we can ignore a_1 in addition to a_3 which is already in the sleep set of σ. However, it is assumed that $(a_2, a_3) \notin I$. Therefore, we have to remove a_3 from the inherited sleep set. Therefore, at σa_2, a_1 is the only thing that can be ignored. The LTA effectively accepts all such trees for all possible choices of $O(\sigma)$ at each node σ by maintaining these (finite) sleep sets in its state. The full construction, which is inspired by the one given in [Farzan and Vandikas 2019a] for a symmetric I, appears in the proof of the Theorem below:



Theorem 4.6. *For any regular language P and semi-independence relation I, the set of S-reductions of P defined by I is recognized by an LTA.*

Since the set of S-reductions is parametric on I, it is interesting to explore the connection between two different reduction sets $\mathrm{SRed}_I(P)$ and $\mathrm{SRed}_J(P)$ when $I \subseteq J$. It is tempting to think that $I \subseteq J \implies \forall P.\, \mathrm{SRed}_J(P) \subseteq \mathrm{SRed}_I(P)$. The more liberal semi-independence relation, in this case J, permits more aggressive prunings and hence produces smaller reductions. But its reductions are not reductions of I, specially if $I \subset J$. The following statement is true, which has the same desired positive effect for proof checking:

Proposition 4.7. *Given a program P, two semi-independence relations I and J, and an ordering function $O : \Sigma^* \to \mathcal{L}in(\Sigma)$, if $I \subseteq J$ then $P{\downarrow_{J,O}} \subseteq P{\downarrow_{I,O}}$.*

This means that if the program has a proof up to a reduction with a weaker semi-independence relation, it will always have a proof for a reduction according to a stronger semi-independence relation. It also implies in a straightforward manner that these reductions subsume the reductions proposed in [Farzan and Vandikas 2019a] based on symmetric independence relations.

In the special case where I is symmetric (as is the case in [Farzan and Vandikas 2019a]), each $L \in \mathrm{SRed}_I(P)$ is guaranteed to be optimal in the sense that the elements of L are pairwise incomparable [Farzan and Vandikas 2019a]. Unfortunately, this does not hold for a general (non-symmetric) I. For example, the language defined by the tree in Figure 5(iii) is a strict superset of the language defined by the tree in Figure 5(ii). Specifically, the language of the tree in Figure 5(iii) contains both $abcd$ and $bacd$, and the latter trace is redundant because $bacd \sqsubseteq_I abcd$. Therefore, some program reductions defined by $\mathrm{SRed}_I(P)$ contain redundant traces and are non-optimal.

4.2 Computing a Sound Semi-Independence Relation

LTA-representability of the class of S-reductions (Theorem 4.6) is the key result of Section 4. Soundness of S-reductions relies on Lemmas 4.2 and 4.4. Lemma 4.4 requires that the program P is upwards-closed with respect to the independence relation I and Lemma 4.2 requires that I is sound. Here, we outline how a relation I that satisfies both criteria can be constructed, with the help of a theorem prover. Proposition 4.7 implies that one should try to obtain as large of an independence relation as possible in order to maximize the likelihood that there exists a reduction with a proof. One can show that every program P admits a *maximal semi-independence relation* I_P.

The (Brzozowski) derivative of a language P and a string σ is defined as

$$\sigma^{-1}P = \{\tau \in \Sigma^* \mid \sigma\tau \in P\}.$$

Theorem 4.8. *The relation I_P defined as $I_P = \{(a, b) \mid a \neq b \wedge \forall\sigma.\, (\sigma ab)^{-1}P \subseteq (\sigma ba)^{-1}P\}$ is the largest (with respect to $\subseteq$) semi-independence relation such that $P = \lceil P \rceil_{I_P}$ (upwards closedness of P).*

When P is regular (as is the case for all of our input programs), it is possible to construct I_P directly, as the proposition $\forall\sigma.\, (\sigma ab)^{-1}P \subseteq (\sigma ba)^{-1}P$ is equivalent to a subsumption relation on states of the DFA recognizing P [Diekert and Rozenberg 1995].

Theorem 4.9. *If P is regular, then I_P is computable.*

I_P may be unsound. One can always obtain a *maximal sound semi-independence relation* by removing from I_P all statements a and b that do not satisfy $[\![ab]\!] \subseteq [\![ba]\!]$. This last step needs to be performed by making calls to a theorem prover. Note that since a and b are program statements, the computation of this relation takes place once at the beginning of the verification process by making a quadratic number (in the program size) of calls to a solver.

5 CONTEXTUAL REDUCTIONS

We introduce the notion of *context* for program reductions through the definition of a *contextual semi-independence* relation. The independence relation is strengthened through the consideration of the context information, where statements can be declared (semi-) commutative only in some contexts. Context is typically considered to be a state (or a set of states) from which the transitions are being considered [Godefroid and Pirottin 1993; Katz and Peled 1992]. We propose a different notion of context which is more useful in our language-theoretic setting, where the *history* of the trace is used as context.

Concretely, a *contextual semi-independence relation* is a function $\mathcal{I} : \Sigma^* \to \mathcal{P}(\Sigma \times \Sigma)$ from traces to irreflexive relations. Intuitively $(a, b) \in \mathcal{I}(\sigma)$ should hold for statements a and b and a program trace σ where a can be swapped with b in context σ, that is $[\![\sigma ab]\!] \subseteq [\![\sigma ba]\!]$. Note that contextual semi-independence subsumes normal semi-independence, which can be considered as the special case of a constant function; i.e. the same independence relation is assigned to all contexts.

Define $\sqsubseteq_{\mathcal{I}}$ to be the smallest preorder satisfying $\sigma ab\rho \sqsubseteq_{\mathcal{I}} \sigma ba\rho$ for all $\sigma, \rho \in \Sigma^*$ and $(a, b) \in \mathcal{I}(\sigma)$. Upwards and downwards closure and closedness are defined as before. We say $\mathcal{I}$ is *sound* if $[\![\sigma ab]\!] \subseteq [\![\sigma ba]\!]$ for all $\sigma \in \Sigma^*$ and $(a, b) \in \mathcal{I}(\sigma)$.

Example 5.1. Recall the example of Figure 3, where we discussed the idea of contextual commutativity of `inc()` and `dec()` at the high level in Section 2. Concretely, $(\texttt{inc()}, \texttt{dec()}) \in \mathcal{I}(\sigma)$ and $(\texttt{dec()}, \texttt{inc()}) \in \mathcal{I}(\sigma)$, for all σ where the number of `inc()` statements is strictly larger than the number of `dec()` statements in σ.

Since our contexts are defined language-theoretically, the definition of $\downarrow$ can be naturally extended to support contexts. Define

$$P{\downarrow_{\mathcal{I},O}} = P \setminus \{\sigma a \rho b \upsilon \mid \sigma, \rho, \upsilon \in \Sigma^* \wedge (a, b) \in O(\sigma) \wedge \forall \tau c \leq a\rho.\, (c, b) \in \mathcal{I}(\sigma\tau)\}$$

where $\leq$ is the prefix relation on strings. Similar to the definition of semi-commutative reductions, strings are soundly pruned from the program language to obtain each reduction $P{\downarrow_{\mathcal{I},O}}$, where

O is an order that determines the exploration strategy for the particular reduction. The set of all reductions is then defined as

$$\mathrm{CRed}_{\mathcal{I}}(P) = \{P{\downarrow}_{\mathcal{I},O} \mid O : \Sigma^* \to \mathcal{L}in(\Sigma)\}.$$

When $\mathcal{I}$ is a constant function, which makes the contextual relation collapse into the standard semi-independence relation of Definition 4.1, $\mathrm{CRed}_{\mathcal{I}}(P)$ is representable as an LTA (by Theorem 4.6). This does not hold true for a general $\mathcal{I}$. Since the goal of this paper is the development of algorithms for enumerating reductions effectively, we are strictly interested in cases where for a given $\mathcal{I}$, the set of reductions $\mathrm{CRed}_{\mathcal{I}}(P)$ is LTA representable.

5.1 A Representable Class of Contextual Reductions

A contextual semi-independence relation $\mathcal{I} : \Sigma^* \to \mathcal{P}(\Sigma \times \Sigma)$ can be alternatively viewed as an infinite tree labelled by a standard semi-independence relation. Thus we call $\mathcal{I}$ *regular* if it corresponds to a regular tree. An infinite tree is *regular* iff it contains a finite number of unique subtrees. Equivalently, an infinite tree is regular iff it can be generated by a modified DFA with states marked with arbitrary labels (in our case, semi-independence relations) instead just being labelled as final or non-final. Since $O : \Sigma^* \to \mathcal{L}in(\Sigma)$ can be viewed as an infinite tree, its regularity can be accordingly defined. Program reductions induced by regular contextual independence relations are LTA-representable:

Theorem 5.2. *If $\mathcal{I}$ is regular, then* $\mathrm{CRed}_{\mathcal{I}}(P)$ *is representable as an LTA.*

Similar to the semi-commutative case (stated in Theorem 4.8), every program P has a *maximal contextual semi-independence relation* $\mathcal{I}_P$, defined as

$$\mathcal{I}_P(\sigma) = \{(a, b) \mid a \neq b \wedge (\sigma ab)^{-1}P \subseteq (\sigma ba)^{-1}P\}.$$

One can naturally lift $\subseteq$ to functions, where $\mathcal{I}_1 \subseteq \mathcal{I}_2$ iff $\forall\sigma.\, \mathcal{I}_1(\sigma) \subseteq \mathcal{I}_2(\sigma)$. This provides an order on the set of contextual relations with respect to which one can define maximality. This leads us to the contextual analog of Theorem 4.8:

Theorem 5.3. *$\mathcal{I}_P$ is the largest (with respect to $\subseteq$) semi-independence relation satisfying $P = \lceil P \rceil_{\mathcal{I}_P}$.*

When P is a regular language, $\mathcal{I}_P$ is a computable function. In fact, a stronger result holds:

Theorem 5.4. *If P is regular then so is $\mathcal{I}_P$.*

As in the semi-commutative case, there is no guarantee that $\mathcal{I}_P$ is sound, and one may obtain a maximal sound contextual semi-independence relation $\mathcal{I}_P^{\text{sound}}$ by removing the unsound elements:

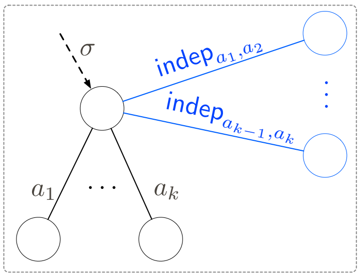
$$\mathcal{I}_P^{\text{sound}}(\sigma) = \mathcal{I}_P(\sigma) \setminus \{(a, b) \mid [\![\sigma ab]\!] \nsubseteq [\![\sigma ba]\!]\}.$$

Unlike the semi-commutative case, where any semi-commutative relation defines an LTA-representable set of reductions, there is no guarantee that $\mathrm{CRed}_{\mathcal{I}_P^{\text{sound}}}(P)$ is LTA-representable.

Theorem 5.5. *The set* $\mathrm{CRed}_{\mathcal{I}_P^{\text{sound}}}(P)$ *generally cannot be represented by an LTA.*

This makes $\mathcal{I}_P^{\text{sound}}$ unsuitable for use in automated verification. Fortunately, we have a solution for this problem. Recall that the ultimate goal is to find some reduction $P{\downarrow}_{\mathcal{I},O}$ that can be proven safe, and to this end, it is not necessary that $\mathcal{I}$ be maximal. It is difficult (if not impossible), without knowing anything about the proof, to make a correct choice about $\mathcal{I}$ in advance. Therefore, we can be clever and handle the choice of $\mathcal{I}$ in the same way that we handle the choice of a particular exploration order O for a reduction: we construct the set of all possible $P{\downarrow}_{\mathcal{I},O}$ over all possible $\mathcal{I}$ and O. It will be left to the verification algorithm to *discover* the right choices for both $\mathcal{I}$ and O.

Since not all independence relations are sound, we ensure our reductions are valid by adding additional *soundness constraints* to each reduction in the form of additional traces that can only be proven correct if the underlying independence relation is sound. This is done via an additional set of *independence statements* $\Sigma_{\text{indep}} = \{\text{indep}_{a,b} \mid a, b \in \Sigma\}$ with the semantics $[\![\text{indep}_{a,b}]\!] = [\![ab]\!] \setminus [\![ba]\!]$. Intuitively, each $\text{indep}_{a,b}$ is infeasible iff it is executed in a state where statement a commutes to the right of b. As illustrated on the right, for every node σ, and each pair of outgoing transitions a_i and a_j, a new independence transition with the label indep_{a_i,a_j} is added to a new fresh state. The states/transitions illustrated in blue are the new additions. The set of soundness constraints for a particular independence relation $\mathcal{I} : \Sigma^* \to \mathcal{P}(\Sigma \times \Sigma)$ is then defined as



$$\text{sound}(\mathcal{I}) = \{\sigma \cdot \text{indep}_{a,b} \mid (a, b) \in \mathcal{I}(\sigma)\}.$$

Intuitively, this corresponds to unreachability of all newly added (blue) states in the schematic figure above, which is formalized in the lemma below:

Lemma 5.6. *$\mathcal{I}$ is sound iff* $[\![\text{sound}(\mathcal{I})]\!] = \emptyset$.

At this point, we can claim P is safe if *both* $P{\downarrow_{\mathcal{I},O}}$ and $\text{sound}(\mathcal{I})$ are safe. Since two programs are safe iff their union is safe, we can treat $P{\downarrow_{\mathcal{I},O}}$ and $\text{sound}(\mathcal{I})$ as a single reduction by taking their union.

Theorem 5.7. *For any independence relation $\mathcal{I} : \Sigma^* \to \mathcal{P}(\Sigma \times \Sigma)$ and upwards-closed program P, $P{\downarrow_{\mathcal{I},O}} \cup \text{sound}(\mathcal{I}) \preceq P$.*

Finally, we are ready to define our set of contextual reductions.

Definition 5.8 (C-Reductions). Given a program P, the set of C-reductions of the program is defined as:

$$\text{CRed}^*(P) = \{P{\downarrow_{\mathcal{I},O}} \cup \text{sound}(\mathcal{I}) \mid O : \Sigma^* \to \mathcal{L}in(\Sigma), \mathcal{I} \subseteq \mathcal{I}_P\}.$$

C-reductions, like S-reductions, are effectively representable for algorithmic verification:

Theorem 5.9. *For any regular program P the set of C-reductions (i.e.* $\text{CRed}^*(P)$*) of P is recognized by an LTA.*

Note that since the LTA represents the set of all reductions with all possible choices of $\mathcal{I}$, it includes the reductions with the specific choice of the maximal sound contextual relation $\mathcal{I}_P^{\text{sound}}$. Therefore, reductions based on $\mathcal{I}_P^{\text{sound}}$ will be considered for the proof without the need for them to be captured by an LTA as a single set.

5.2 Finite Programs

There is research [Wang et al. 2009] that focuses on reductions in the context of bounded model checking (i.e. bug finding) for concurrent programs. It is therefore worthwhile to mention that for this special class of programs, where the program language includes only finitely many traces, an appropriate regular reduction always exists.

Theorem 5.10. *For any finite P, there exists a sound regular independence relation $\mathcal{I} \subseteq \mathcal{I}_P$ such that* $\text{CRed}_{\mathcal{I}}(P) = \text{CRed}_{\mathcal{I}_P}(P)$.

While the "ideal" independence relation $\mathcal{I}_P^{\text{sound}}$ defined previously may not be regular, and therefore may not satisfy the precondition of Theorem 5.2, Theorem 5.10 implies that we can always construct another regular independence relation that produces equivalent reductions. This implies that completeness with respect to reductions can be achieved for bounded programs, while it is not achievable for general (unbounded) programs.

6 RELATIONSHIP TO KNOWN REDUCTION TECHNIQUES

Now that we have introduced C-reductions, it is only natural to ask if they *subsume* some well-understood and widely used reduction techniques specific to certain program classes. To this end, we investigate the relation between C-reductions and the two known approaches of Lipton [Lipton 1975] and Existential Boundedness [Genest et al. 2007].

6.1 Lipton's Atomic Block Reductions

In Lipton's reductions, semi-commutativity properties of statements are used as sufficient conditions to infer *atomic blocks*. Declaring a block of code atomic (and having it executed without interruption) has the effect of reducing the number of thread interleavings that must be proved correct.

Lipton's condition is based on *left movers* and *right movers*. A left mover (respectively right mover) is a statement that soundly commutes to the left (respectively right) of every statement in every other thread. A sequence of statements $a_1 \dots a_n b c_1 \dots c_m$ may be declared atomic if each a_i is a right mover and each c_j is a left mover. First, note that Lipton's original definition for left/right-movers was a *contextual* definition. He defined an action to be a left/right-mover if it left/right commutes from all concrete reachable program contexts. For a program with variables ranging over infinite data domains, this implies infinitely many possible contexts. Therefore, a corresponding contextual commutativity relation will be incomputable in general. Given a general contextual left/right-mover specification, checking the soundness of it according to Lipton's original contextual definition is undecidable since determining the set of reachable states is undecidable in general. This is perhaps why all instances of usage of Lipton's reductions in the literature are non-contextual. One can view our C-reductions as an attempt to provide a decidable approximation of this original definition for program safety verification.

Let us now compare our (non-contextual) S-reductions against the commonly used non-contextual definition of Lipton's reductions. First, observe that Lipton's reductions are not optimal, even if maximal blocks of atomic codes are inferred. For example, consider a *disjoint* parallel program $a_1 a_2 \parallel b_1 b_2$. Since every statement is both a left and a right mover, one can reduce the program to $A \parallel B$ where $A = a_1 a_2$ and $B = b_1 b_2$ are atomic blocks. Note that this reduction includes two execution AB and BA which are equivalent. Therefore, one is redundant. The set of S-reductions of this program will include reductions with no redundancies; there are 4 of them in total, and each has a single trace in it.

Now, let us assume the program $a_1 a_2 \parallel b_1 b_2$ is not disjoint parallel anymore and a_1 and b_1 are right movers. This means that P can still be reduced to $A \parallel B$ with the same atomic blocks. Note that Lipton's definition of a right mover is strictly stronger than (and therefore implies) our definition of semi-commutativity. That is, if a_1 is a right mover, then for all $l \in \Sigma$, $(a_1, l) \in I$. Therefore, we have a sound semi-independence relation

$$I = \{(a_1, b_1), (a_1, b_2), (b_1, a_1), (b_1, a_2)\}.$$

The program is upwards-closed with respect to I. One of the S-reductions of the program is illustrated in Figure 6. Assume that at every relevant node, the a transition is explored first (which would correspond to the prefix order traversal of the depicted tree). The figure illustrates which runs are pruned (greyed out) and which remain in the reduction. Specifically, the *unexpected* trace

$a_1 b_1 b_2 a_2$ has to remain in the reduction because the trace $b_1 b_2 a_1 a_2$ which subsumes it is only visited *later*, and cannot be used for pruning. Therefore, this particular s-reduction has an extra trace compared to Lipton's reduction $A \parallel B$. The rest of the choices of order (other than exploring a's before b's at every point) follow a similar pattern, and include other redundant runs. It is easy to check that all four S-reductions of this program include redundant traces in addition to Lipton's reduction.

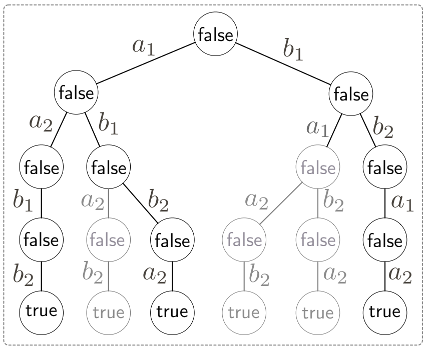


Fig. 6. S-reductions vs Lipton's reductions.

The combination of the two examples demonstrate how our S-reductions and Lipton's (non-contextual) reductions are incomparable and therefore complementary. Note that for any concurrent program (with a bounded number of threads), there are only finitely many choices for atomic blocks. Therefore, one can imagine enumerating them all as a specialized reduction class. Since there are finitely many possible reductions, the class of reductions is trivially recognizable by an LTA. However, our generic C-reduction (or S-reduction) classes do not necessarily include all these (finitely many) reductions.

6.2 Existential Boundedness

Programs operating on unbounded FIFO channels typically require quantified invariants for their correctness proofs. For example, the simple code in Figure 7(a) requires an invariant stating that all elements in transit at any given time are equal to 5. Note, however, that every trace of this program is equivalent to a trace of the program in Figure 7(b) where every receive occurs right after its corresponding send. For each trace in Figure 7(b), there is at most one element in transit at any given time, which makes the program effectively finite state. The proposed approaches in [Desai et al. 2014; von Gleissenthall et al. 2019], for example, verify the program in Figure 7(a) precisely by transforming it to the one in Figure 7(b).

The program in Figure 7(b) is called *universally bounded*, since there is a fixed bound k (= 1 in this case) where no execution of the program has more than k messages in transit at any given time. Conversely, the program in Figure 7(a) is called *existentially bounded*, because every execution of this program is equivalent to some execution where no more than k messages are in transit at any given time. Universally bounded programs are usually much easier to verify because their channels are bounded. We argue that if a program is existentially bounded then there exists a universally bounded C-reduction. Note that our algorithm does not have to be aware of the existential boundedness of its input. The claim is that if the input is existentially bounded, the C-reductions will provide the opportunity for the simpler proof to be found.

Precondition: $\{x = 5\}$

```
i = 0;               || j = 0;
while (i < N) {      || while (j < N) {
   send(5);          ||    x = receive();
   i++;              ||    j++;
}                    || }
```

Postcondition: $\{x = 5\}$ (a)

```
i = 0;
while (i < N) {
   send(5);
   x = receive();
   i++;
}
```

(b)

Fig. 7. A Synchronous Reduction.

To provide the formal result, we use the setup similar to the one in [Genest et al. 2007] (in this section only). We fix a finite set Chan of unbounded FIFO channels. For simplicity, we assume programs can *only* perform send and receive actions on channels. Furthermore, we do not concern ourselves with the particular contents of the channels; we only care about the number of messages in transit at any given time. Formally, we instantiate our state set to $\mathcal{St}$ = (Chan $\rightarrow \mathbb{N}$) and our alphabet to $\Sigma = \{\text{send}(c), \text{recv}(c) \mid c \in \text{Chan}\}$. We assign the semantics

$$[\![\text{send}(c)]\!] = \{(f, f[c \mapsto f(c)+1]) \mid f \in \mathcal{S}t\}$$
$$[\![\text{recv}(c)]\!] = \{(f, f[c \mapsto f(c)-1]) \mid f \in \mathcal{S}t, f(c) > 0\}$$

where $f[x \mapsto y]$ denotes function f with the output for x replaced with y.

A trace τ is said to be *k-bounded* if the number of elements in transit in any given channel never exceeds k. Formally, this means that for any prefix σ of τ, we have

$$(\lambda\_.\,0, f) \in [\![\sigma]\!] \implies f(c) \le k$$

for all $f \in \mathcal{S}t$ and $c \in \text{Chan}$. We say a program P is *existentially k-bounded* if every trace of P is equivalent (with respect to the symmetric subset of $\sqsubseteq_{\mathcal{I}_P^{\text{sound}}}$, which we shall denote by $\sim_{\mathcal{I}_P^{\text{sound}}}$) to a k-bounded trace. We say P is *universally k-bounded* if every trace of P is k-bounded. With these definitions in place, we present the main theorem:

Theorem 6.1. *If P is an existentially k-bounded program, then there exists a universally k-bounded reduction $P{\downarrow}_{\mathcal{I}_P^{\text{sound}}, O} \in \text{CRed}_{\mathcal{I}_P^{\text{sound}}}(P)$ for some exploration ordering $O : \Sigma^* \to \mathcal{L}in(\Sigma)$.*

There are existing methods [Desai et al. 2014; von Gleissenthall et al. 2019], designed specifically to transform asynchronous programs into synchronous ones. The significance of Theorem 6.1 is that it demonstrates that even though our proposed technique is not specialized for this category, it can be effective in discovering appropriate reductions for existentially bounded programs.

It should be noted that $P{\downarrow}_{\mathcal{I}_P^{\text{sound}}, O}$ from Theorem 6.1 can only be used for verification if a soundness proof for $\mathcal{I}_P^{\text{sound}}$ exists in the assertion language of choice and can be discovered through interpolation. This is not always possible because such proofs would generally have to include assertions that somehow "count" the number of elements in each channel, yielding a non-regular language. However, we conjecture that a weaker independence relation may suffice.

Conjecture 6.2. *If P is an existentially k-bounded program, then there exists a universally k-bounded reduction $P{\downarrow}_{\mathcal{I},O} \in \text{CRed}^*(P)$ for some independence relation $\mathcal{I} : \Sigma^* \to \mathcal{P}(\Sigma \times \Sigma)$ and exploration ordering $O : \Sigma^* \to \mathcal{L}in(\Sigma)$ such that there exists a simple proof Π (i.e. Π is in linear integer arithmetic) such that* $\text{sound}(\mathcal{I}) \subseteq \mathcal{L}(\Pi)$.

7 REFINEMENT-STYLE VERIFICATION ALGORITHM

Figure 8 illustrates the outline of our verification algorithm. It is a counterexample-guided abstraction refinement loop in the style of [Farzan and Vandikas 2019a; Farzan et al. 2013, 2015; Heizmann et al. 2009]. The algorithm starts with assertions true and false in the proof and iteratively discovers more assertions until a proof is found. Unlike standard refinement loops, it suffices to discover the proof for a *sound* reduction of the program and not the entire program.

To check the validity of any candidate proof Π, one has to check if there exists a reduction of the program that is covered by Π. The results presented in this paper so far naturally give rise to an algorithm for performing this check:

- The set of program reductions are LTA-representable (Theorems 4.6 and 5.2).
- A regular proof can be constructed based on a candidate set of assertions (see Section 3.1).
- LTA representability of the program and regularity of the proof language imply that the check can be performed through an emptiness test for the intersection of LTA languages (Theorem 3.3). There are known algorithms for both tasks [Baader and Tobies 2001].

When the proof is not sufficient, a *finite* set of counterexamples can be produced as a witness. These counterexamples are sequences of statements. One can check whether they are feasible or not. In a standard setting, a feasible counterexample would be a witness for the violation of

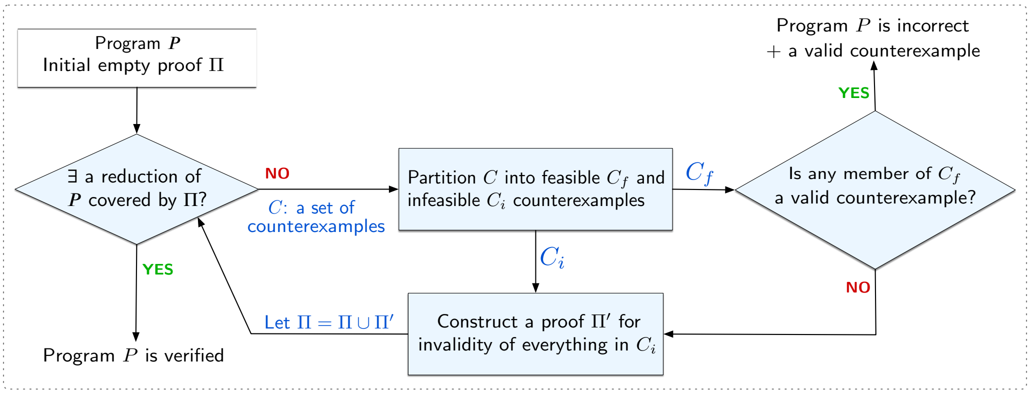

Program P
Initial empty proof Π
$\exists$ a reduction of P covered by Π?
NO
C: a set of counterexamples
YES
Program P is verified
Partition C into feasible C_f and infeasible C_i counterexamples
C_f
C_i
Is any member of C_f a valid counterexample?
YES
Program P is incorrect + a valid counterexample
NO
Construct a proof Π' for invalidity of everything in C_i
Let $\Pi = \Pi \cup \Pi'$

Fig. 8. Counterexample-guided refinement loop.

the property by the program. In our setting, not every counterexample is a program trace. If a program trace counterexample is feasible, then the program is unsafe. We stop the loop and return the counterexample.

The specific construction of our LTA for C-reductions implies that some reductions in the class may be unsound. Any feasible counterexample that is not a program trace is a witness for the unsoundness of (at least) one such reduction. We refer to these as *invalid* counterexamples. These are therefore spurious counterexamples and are ignored. We address how this does not affect the termination of the refinement loop in Section 7.1.

If no true counterexample is found, then one can produce proofs of infeasibility of all the infeasible counterexamples with the aid of any program prover. All new assertions discovered through this process are then added to the current proof conjecture, and the refinement loop restarts. Note that proofs of infeasibility of program trace counterexamples contribute towards the discovery of a program proof, and proofs of infeasibility of the rest would contribute towards discovery of invariants that expand the set of sound contextual commutativity relations. In our tool, we use Craig interpolation to produce proofs of infeasibility of these counterexamples. In general, since program traces are the simplest forms of sequential programs (loop and branch free), any automated program prover (that can handle proving them) may be used.

7.1 Termination, Soundness and Completeness

Let us assume that the program is correct, and more specifically, there exists a proof Π^* that subsumes one of its contextual reductions in $\text{CRed}^*(P)$. Ideally, we would like to claim that the refinement algorithm of Figure 8 succeeds under these conditions.

Note that the convergence of the algorithm depends on two factors: (1) the counterexamples used by the algorithm belong to $\mathcal{L}(\Pi^*)$ and (2) the proofs discovered by the backend solver for these counterexamples use assertions from 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. This problem is orthogonal to the contribution of this paper.

The former poses a new problem specific to our methodology: if the ideal proof Π^* is the target of the refinement loop, since the set of traces proved correct in $\mathcal{L}(\Pi^*)$ is incomparable to the set of program traces P, then one cannot just use any program trace as an *appropriate* counterexample.

A key observation from LTA's can be used to solve this problem. For an LTA M and regular language L, when there exists no $R \in \mathcal{L}(M)$ such that $R \subseteq L$, then there exists a finite set of counterexamples C such that for all $R \in \mathcal{L}(M)$, there exists some $\tau \in C$ such that $\tau \in R$ and $\tau \notin L$ [Farzan and Vandikas 2019a]. This is very significant, since it means that when we check a proof candidate $\mathcal{L}(\Pi)$, and it does not cover any program reduction, then there are finitely many counterexamples that together *cover* all reductions. This means that if all these counterexamples are proved infeasible, and Π is updated with these proofs, then the proof has made meaningful progress for every single reduction.

But what about invalid counterexamples from the set C_f (of Figure 8) which our algorithm simply ignores? Note that these may appear again in subsequent rounds. Fortunately, this behaviour turns out to be unproblematic: strong progress essentially relies on finding new counterexamples for each *correct* reduction. Whether we find new (or even any) counterexamples for incorrect reductions is of no importance.

Theorem 7.1 (Strong Progress). *Assume there exists a reduction $P^* \in \mathrm{CRed}^*(P)$ (or alternatively $\mathrm{SRed}^*(P)$) and a set of assertions Π^*, such that $P^* \subseteq \mathcal{L}(\Pi^*)$. If the algorithm of Figure 8 uses assertions from Π^* to prove the infeasibility of those counterexamples which belong to $\mathcal{L}(\Pi^*)$, then it 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 extra condition on proofs of traces from Π^* rules out diverging behaviour that could occur due to the wrong choice of assertions by the backend prover. A wrong choice of assertions can cause divergence in any standard software model checking algorithm (even for sequential integer programs with simple proofs) that relies on discovery of loop invariants through interpolation. The assumption that there exists a proof for a reduction (in the fixed set $\mathrm{CRed}^*(P)$) of the program ensures that the algorithm is searching for a proof that does exist. Note that, in general, a proof may exist for a reduction of the program which is not in $\mathrm{CRed}^*(P)$. That is, the algorithm is not complete with respect to all reductions, but only reductions in $\mathrm{CRed}^*(P)$. Since checking the premises of SafeRed for all semantic reductions is undecidable as discussed in Section 3, a complete algorithm does not exist. The soundness of the algorithm is a straightforward consequence of the soundness of reductions stated in Sections 4 and 5.

8 AN EFFICIENT ALGORITHM FOR PROOF CHECKING

The proof checking algorithm described in Section 7 boils down to an emptiness check on the intersection (i.e. product) of the LTAs representing all program reductions and the proof language. The size of the reduction LTA is exponential on the input program size, both in terms of the number of states and the number of transitions per state. This can make proof checking prohibitively expensive, even though the emptiness test is performed in linear time. In this section, we propose a new algorithm for proof checking that has a provably better worst-case time complexity and works better in practice. First, we show how to drastically reduce the number of transitions considered during proof checking. This also allows us to easily employ antichain-based optimizations (in the style of [Farzan and Vandikas 2019a; Wulf et al. 2006]) to better deal with the exponential state space, which in turn allows us to further reduce the number of transitions considered and arrive at an asymptotically better algorithm than the one given in [Farzan and Vandikas 2019a].

8.1 Proof-Driven Maximal Independence Relations

Our construction of the reduction LTA enumerates a class of contextual independence relations and a class of reductions that are induced by them. The key observation of this section is that a specific proof candidate instigates some additional structure in the state space of all contextual independence relations that would allow us to *soundly and completely* choose one *maximal* relation instead of exploring them all.

For the purpose of checking a proof candidate Π, it suffices to only consider those independence relations that are correct according to Π, i.e. all independence relations $\mathcal{I} : \Sigma^* \to \mathcal{P}(\Sigma \times \Sigma)$ such that $\text{sound}(\mathcal{I}) \subseteq \mathcal{L}(\Pi)$. The proof checking algorithm never certifies a reduction based on an unproven independence relation. In fact, even among independence relations that are correct according to Π, it suffices to consider only a single *maximal independence relation* $\mathcal{I}_\Pi \subseteq \mathcal{I}_P$ induced by the proof Π, defined as

$$\mathcal{I}_\Pi(\sigma) = \{(a, b) \in \mathcal{I}_P(\sigma) \mid \sigma \cdot \text{indep}_{a,b} \in \mathcal{L}(\Pi)\}.$$

This independence relation declares a pair of statements independent precisely when it is sound to do so according to Π. By maximal, we mean that any independence relation $\mathcal{I}$ that is sound according to Π is subsumed by $\mathcal{I}_\Pi$.

Proposition 8.1. *For any independence relation $\mathcal{I} \subseteq \mathcal{I}_P$, if* $\text{sound}(\mathcal{I}) \subseteq \mathcal{L}(\Pi)$ *then $\mathcal{I} \subseteq \mathcal{I}_\Pi$.*

$\mathcal{I}_\Pi$ can be shown to be regular by modifying the automaton recognizing $\mathcal{L}(\Pi)$. By Theorem 5.2, $\text{CRed}_{\mathcal{I}_\Pi}(P)$ is representable by an LTA. The following theorem states that proof checking against $\mathcal{I}_\Pi$ is just as good as proof checking against all independence relations.

Theorem 8.2. *There exists some $P' \in \text{CRed}^*(P)$ satisfying $P' \subseteq \mathcal{L}(\Pi)$ iff there exists some $P'' \in \text{CRed}_{\mathcal{I}_\Pi}(P)$ satisfying $P'' \subseteq \mathcal{L}(\Pi)$.*

Therefore, we can replace the LTA representing $\text{CRed}^*(P)$ with the LTA representing $\text{CRed}_{\mathcal{I}_\Pi}(P)$ in our proof checking algorithm. While this reduces the number of transitions considered exponentially, it also increases the state space of the reduction LTA since $\text{CRed}_{\mathcal{I}_\Pi}(P)$ must simulate the automaton witnessing regularity of $\mathcal{I}_\Pi$. This turns out to be of no consequence: the proof automaton already contains the state space of $\mathcal{I}_\Pi$, so the state space of the product automaton remains unchanged. Thus we obtain a product automaton with exponentially fewer transitions and an identical state space, resulting in an exponentially faster proof checking algorithm.

8.2 Optimizing Emptiness Test Through Antichains

While LTAs can be checked for emptiness in linear time, the size of the checked automaton is exponential in $|\Sigma|$ (in the worst case), since the reduction LTA (as described in Section 4.1) must maintain sleep sets. In [Farzan and Vandikas 2019a], this problem is alleviated using antichain methods [Wulf et al. 2006] for the case of symmetric and non-contextual independence relations. The idea is that emptiness checking reduces to constructing a set of *inactive* states from which no language is accepted. The set of inactive states is shown to be downwards-closed with respect to a particular subsumption relation, which allows the set to be represented compactly by its maximal elements (i.e. antichains).

It turns out that with the contextual (and semi-) independence relations, we can also adapt these methods as an optimization for proof checking. This comes as a positive byproduct of the use of the maximal independence relation $\mathcal{I}_\Pi$ that we described in Section 8.1. The key observation is that we can pretend that we are in the non-contextual setting of [Farzan and Vandikas 2019a], but recover all the relevant information about $\mathcal{I}_\Pi$ from the state of the product automaton that contains information about both program and proof automaton states. Therefore, we can recover the required information about $\mathcal{I}_\Pi$ and know what transitions are independent at each state of the product automaton. The technical details about the adaptation are included in the extended version of this paper [Farzan and Vandikas 2019b].

Antichain methods do not generally improve the worst-case computational complexity of an algorithm, since the size of the largest antichain on sets of a finite alphabet is still exponential in the size of the alphabet. However, antichains are never larger than the sets they represent, and often exponentially smaller, making antichain-based algorithms very efficient in practice.

8.3 Time Complexity of Proof Checking

The final source of complexity that we would like to eliminate is a factorial component that arises because our reduction automaton considers all possible exploration strategies of the program (through enumerations of all order functions). In particular, each state in the reduction automaton has a transition for all $|\Sigma|!$ linear orderings over Σ. In conjunction with the optimization of Section 8.2, this translates to an iterated antichain meet over all linear orders, which has exponential-of-factorial complexity. Fortunately, we can reduce this factor to only exponential in the size of Σ.

As mentioned previously, LTA emptiness is calculated via a fixed point computation. The complexity occurs within the function over which the fixed point is computed. Therefore, we shall focus on the complexity of calculating this function. This function, which we call $F^{\max}$, has type

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

where Q_P and Q_Π are respectively the state spaces of the DFAs accepting the program language P and the proof language $\mathcal{L}(\Pi)$. Since the input of $F^{\max}$ is itself a function, we define the size of a function $X : Q_P \times Q_\Pi \to \mathcal{P}(\mathcal{P}(\Sigma))$ to be the maximum size of all its outputs, i.e.

$$|X| = \max\{|X(q_P, q_\Pi)| \mid q_P \in Q_P, q_\Pi \in Q_\Pi\}.$$

Theorem 8.3. *The algorithm as of Section 8.2 computes $F^{\max}(X)$ in $O((|\Sigma||X|)^{2|\Sigma|!})$ time.*

There is a key observation that allows us to improve these results. The following lemma captures this by effectively permitting the elimination of the set of linear orders from our calculations:

Lemma 8.4. *Let $A \subseteq \Sigma \times \mathcal{P}(\Sigma)$ be a relation satisfying $\forall (a, S) \in A.\, a \in S$. Then*

$$(\forall O \in \mathcal{L}in(\Sigma).\, \exists (a, S) \in A.\, O(a) \subseteq S) \iff (\exists \emptyset \subset A' \subseteq A.\, \forall (a, S) \in A'.\, \overline{\mathrm{Dom}(A')} \subseteq S)$$

where $\mathrm{Dom}(A')$ *is the domain of* A'.

The equality implied by the lemma is used to improve the fixed point calculation of $F^{\max}$, based on the LTA built using the construction of Theorem 5.2 instantiated by the sound independence relation of Proposition 8.1. The left-hand side of the equality appears in the fixed point computation, and the right hand side lets us drop the enumeration of all linear orders from it. This leads us to the following result of reduced overall complexity for the dominant computation part of proof checking:

Theorem 8.5. *$F^{\max}(X)$ can be computed in $O(2^{|\Sigma|}|\Sigma||X|)$ time.*

9 EXPERIMENTAL RESULTS

There are two facts that make an experimental evaluation of the technique worthwhile: (1) Our reduction sets are necessarily *incomplete*. There may exist a general semantic reduction of the program (in the sense of Definition 3.1) with a simple proof, but this reduction may not belong to the set of S-reductions or C-reductions defined in this paper. Therefore, an experimental evaluation to see how well the incomplete reductions fare in practice is essential. (2) The worst case time complexity of our algorithm is exponential, and therefore, it is important to know if an implementation of this algorithm can handle realistic examples.

9.1 Implementation

We have implemented our approach in a tool called Siever written in Haskell. Siever accepts a program written in a simple imperative language. The input language supports integers, booleans, arrays, uninterpreted functions, deterministic and non-deterministic branches and loops, parallel composition, assume statements, and assignment statements. The desired safety property is encoded in the input program itself in the form of *assume* statements, and Siever attempts to prove the program safe.

Siever implements all of the optimizations of Section 8. Note that since the algorithm as of Section 8 computes a fixpoint of a function over the product state space of the program and proof DFAs, no tree automata are ever explicitly constructed during an execution of the tool.

SMT Solvers. Siever supports a variety of background solvers. Only a few solvers support interpolation, but Siever can use different solvers for interpolation and proof generalization. For interpolation, Siever supports Z3, MathSAT, and SMTInterpol. For proof generalization, Siever additionally supports Yices and CVC4. The main reason for multiple solver support is the general fragility of the interpolation tools. For example, MathSAT does well on some of our arithmetic benchmarks, but bugs out easily with the array benchmarks, while SMTInterpol does better with array interpolants. On the other hand, MathSAT performs better when it works.

Counterexamples. The set of counterexamples that provide the convergence guarantee of Theorem 7.1 are often too large to be practically useful. It turns out that the algorithm converges in the strong majority of the cases if one selects only one counterexample from this set to move forward. The algorithm may take a few more refinement rounds to converge this way, but each round executes much faster and the overall time for verification ends up being substantially lower.

The choice of counterexample can have a substantial impact on the total verification time. One can imagine many heuristics for this selection. We use two specifically for the evaluation in this section: one that picks a (mostly) sequentialized trace from all available traces (S), and another one which picks a mostly interleaved (I) counterexample; that is, it uses the counterexample that is going through the steps of different threads in a round-robin manner.

Recall the example in Figure 1. For verification with contextual (semi) reductions, the time under the (I) counterexample selection criterion is three times slower than the one under (S), since the *good* reduction is the sequential reduction. Big gaps like this one (in either direction) are observed in most benchmarks.

9.2 Evaluation

The target programs for our approach are those where a proof for the entire program is out of the reach of current automated verification tools due to the expressivity of the language of required interpolants. For this reason, Siever cannot be compared against existing tools, as the premise is that they should fail on the majority of these benchmarks.

Given the same proof, checking it against an infinite set of reductions, in contrast to a single program in classic verification, is bound to be (theoretically) more expensive. Therefore, when not needed, reductions can cause a potentially large overhead on verification time. The exception is the cases where they are not strictly needed (from the theoretical point of view) but using them leads to a much smaller proof (in terms of the total number of assertions). In these scenarios, the smaller proof can offset the overhead of proof checking against reductions and lead to a better overall verification time.

Benchmarks. We have a diverse set of benchmarks in Table 1 which includes programs that require the reductions presented in this paper to be verified by an automated prover. In other words, they are theoretically beyond the reach of the automated provers. The reason for this is that a Floyd-Hoare style proof for the entire program (i.e. unreduced) will require a rich language of assertions that reason about (1) unbounded message buffers, (2) non-linear constraints, or (3) quantified facts about arrays, or a combination of these features. The benchmarks are arranged in Table 1 based on the complexity of these required assertions. Siever manages to prove these benchmarks correct by discovering a reduction for which the base theories of Linear Integer Arithmetic (LIA), Uninterpreted Functions (UF), and theory of arrays (unquantified) suffice.

Our second set of benchmarks, reported in Table 2, includes programs for which the S/C-reductions presented in this paper are not strictly required. They either require no reductions at all or the sleep-set reductions (from [Farzan and Vandikas 2019a]) are sufficient for proving them correct automatically. We use this second set of benchmarks to highlight the fact that the rich set of reductions presented in this paper can be of practical importance even if not theoretically required.

Unbounded buffers are modelled in these benchmarks using uninterpreted functions. More precisely, a buffer is modelled using a triple $\langle f, i_0, i_1 \rangle \in (\mathbb{Z} \to \mathbb{Z}) \times \mathbb{Z} \times \mathbb{Z}$ where $f(i)$ denotes the ith element in the buffer, i_0 points to the first element in the buffer, and i_1 points to the last.

Benchmark	Reduction Style		
	S + C	C	S
Unbounded Buffers			
channel-sum	1.8	**0.8**	TO
horseshoe	**45.2**	45.5	TO
prod-cons	7.5	**3.4**	TO
prod-cons-3	138.9	**26.8**	TO
prod-cons-eq	**3.4**	4.3	TO
queue-add-2	6.2	**5.9**	TO
send-receive	5.7	**4.9**	TO
send-receive-alt	1.1	**0.5**	TO
simple-queue	0.3	**0.03**	TO
queue-add-3	**214.4**	TO	TO
Nonlinear			
Figure 3	TO	**0.3**	TO
mult-4	TO	**25.5**	TO
mult-equiv	197.9	TO	**194**
counter-fun	**0.8**	TO	TO
Arrays			
simple-array-sum	**52.4**	97.4	TO
three-array-min	**39.1**	74.2	TO
three-array-sum	**28.1**	58.4	TO
three-array-max	TO	**34.6**	TO
Unbounded Buffers + Nonlinear			
buffer-mult	**131.5**	212.4	TO
buffer-series	**62.9**	216.7	TO
buffer-series-array	**97.9**	292.2	TO
queue-add-2-nl	**14.4**	15.7	TO
queue-add-3-nl	344	**314**	TO
Unbounded Buffers + Arrays			
dec-subseq-array	**4.6**	7.3	TO
inc-subseq-array	**4.5**	6.7	TO

Table 1. Experimental Results. Times are in seconds. Best times are in boldface. TO indicate a timeout (set at 20mins).

Results. We ran Siever on the benchmarks on a Dell Optiplex 3050 with an Intel(R) Core(TM) i7-7700 CPU (4 cores, 2 threads per core) and 32GB of RAM, running 64-bit Ubuntu 18.04. The results are reported in Tables 1 and 2. Siever has an option to turn semi-commutativity on and off, and we used it to measure the impact of it alone, and also when added to contextual commutativity relations. Note that C-reductions (of Definition 5.8) are by default defined based on contextual semi-commutative relations, and therefore, they correspond to the "S + C" option in Table 1. The "C" column corresponds to C-reductions without semi-commutativity.

The "None" column in the table corresponds to our implementation of [Heizmann et al. 2009] which does not perform reductions or any optimizations specific to handling concurrent programs. Therefore, it can be considered as a baseline algorithm. Our benchmarks are not very large programs. It is unlikely that a proof is not found by this baseline algorithm due to known intractability issues of concurrent program verification (i.e. state-space explosion). There are two reasons for failure: (1) the proof for the program is beyond capabilities of state-of-the-art SMT solvers (for interpolation and verification-condition checking), and (2) the algorithm falls into the well-known divergent behaviour of automated verification where sufficiently strong loop invariants are not produced.

All benchmarks in Table 1 fall in category (1). From Table 2, the benchmarks under Arrays also fall in category (1). Without S-reductions, C-reductions, or sleep-set reductions of [Farzan and Vandikas 2019a], there would be a need for universal quantification over array elements. The rest of benchmarks in Table 2, for which the "None" algorithm timeouts, fall under category (2). In these instances, Siever succeeds with reductions because it gets lucky with the counterexamples of the reduction-based method and sidesteps the divergence issues. The example in Figure 1 is one of these examples.

If we modify the precondition to remove the assertion `{ M = N }` and change the postcondition to `{ y = N - M }`, then interpolation-based proofs do not diverge. This is listed in Table 2 as "Figure 1 (alt)". The results clearly demonstrate that C-reductions are very powerful in producing proofs in the majority of cases. There are cases that S-reductions alone make proving a program possible and there are cases that semi-commutativity substantially boosts contextual reductions. Theoretically, adding the option of semi-commutation specifications should only make the tool perform better. However, a change in the independence relation could result in a change in the counterexamples used in the refinement rounds, and in three of the benchmarks (from both tables), this change seems to be adversarial for the algorithm; in these cases, the contextual reductions without semi-commutativity manage to produce a proof, but adding in semi-commutativity causes timeouts.

Other than a few exceptions, the counterexample selection strategy (I) described before produces the fastest time over the benchmarks. With the solvers the results are mixed. The best times are split (near half-half) between MathSAT and SMTInterpol.

Benchmark	Reduction Style				
	S + C	C	S	Weaver	None
Arrays + Nonlinear					
dot-product-array	62.2	105	**50.8**	54.6	TO
Arrays					
max-array-hom	1644	906	**756**	1483	TO
max-array	232	306	**36.2**	446	TO
min-array-hom	830	1366	**578**	**578**	TO
min-array	151.5	180	**51.9**	56.9	TO
sum-array-hom	116	167	**115**	123	TO
sum-array	64	94.5	**46.5**	50.9	TO
parray-copy	311	407	**218**	232	TO
mts-array	TO	TO	**1176**	1190	TO
sorted	TO	TO	**819**	TO	TO
Unbounded Buffers					
commit-1	3.2	5.3	**1.7**	1.7	1.9
commit-2	15.9	38.4	**6.4**	14.4	31.1
two-queue	8.1	**2.6**	15.1	15.2	57.7
Standard Language of Assertions					
Figure 1	0.21	**0.2**	TO	TO	TO
Figure 1 (alt)	1.4	2.1	**1.2**	3.0	3.2
counter-determinism	**5.8**	10.5	TO	TO	TO
difference-det	25.9	25.1	**12.5**	TO	TO
nonblocking-cntr	1.3	**1.1**	TO	TO	TO
nonblocking-cntr-alt	**3.7**	3.8	19.3	28.1	TO
min-le-max	3.2	2.7	0.2	**0.1**	0.4
threaded-sum-2	**3.1**	3.4	9.5	10	3.4
threaded-sum-3	139	90.7	**84.2**	TO	TO

Table 2. Experimental results on benchmarks that do not require contextual reductions. Times are in seconds. Best time for each benchmark appears in boldface. TO indicate a timeout (set at 30mins). Weaver is the tool from [Farzan and Vandikas 2019a]. None is a variation of [Heizmann et al. 2009] without any reductions.

On average 13 refinement rounds are required to verify the benchmarks, with 43 being the maximum number. The average proof size is about 93 assertions and the largest proof includes 340 assertions. Of the benchmarks that take more than 5 seconds to verify, the majority are three-threaded programs. For these, on average, 81% of the time is spent in proof construction, 11% in proof checking, and 8% in interpolation. For the six four-threaded benchmarks, the averages are different: 38% of the time is spent in proof construction, 54% in proof checking, and 8% in interpolation. It is expected that as the number of threads increases, the cost of proof checking should dominate the total verification time since it increases exponentially with the number of threads.

Siever and all of our benchmarks are available at https://github.com/weaver-verifier/weaver.

The Optimized Proof Checking Algorithm

In Section 8, we proposed a novel way of devising a faster proof checking algorithm. We evaluate this algorithm separately by comparing the times taken for proof checking a complete proof for each benchmark in the standard algorithm versus the optimized algorithm. The optimized algorithm performed 6.7x faster than the standard one in the best case, and 1.7x faster on average. In the worst case, the optimized algorithm's performance was the same as the standard algorithm (in exactly one case). Note that this is an isolated evaluation of the algorithms for poofs that check. In the refinement loop, most proof checking tests fail, and using the optimized version produces better overall speedups than these reported numbers. Since the two algorithms may produce different counterexamples, one cannot compare the overall time of the entire verification process between the two proof checking algorithm choices. There is no guarantee that the speedups (or slowdowns) observed are not due to better (or worse) luck with the selection of counterexamples.

10 RELATED WORK

The contributions of this paper relate to several topics, including automated concurrent program verification, relational verification, and program reductions. Each topic has a vast literature of related work. Here, we only explore connections to the most relevant work. Specifically, a large body of related work using reduction for the purpose of bug finding (in contrast to producing proofs) is not discussed since the focus of this paper is on sound reductions for verification.

Reductions for Concurrent Program Verification

Lipton's reduction [Lipton 1975] has inspired several approaches to concurrent program verification [Elmas et al. 2009; Hawblitzel et al. 2015; Kragl et al. 2018], which fundamentally opt for inferring large atomic blocks of code (using various different techniques) to leverage mostly sequential reasoning for concurrent program verification. QED [Elmas et al. 2009] and CIVL [Hawblitzel et al. 2015] frameworks both use refinement-oriented approaches to proving concurrent programs correct. These *semi-automatic* systems use a combination of ideas to simplify proofs of concurrent programs. Specifically, yield predicates (location invariants) are similar to the contexts for commutativity in this paper. CIVL [Hawblitzel et al. 2015] takes advantage of classic movers wherever applicable, so as not to have to rely too heavily on yield predicates. QED [Elmas et al. 2009] performs small rewrites in the concurrent program that have to be justified by potentially expensive reduction and invariant reasoning. Both systems are more broadly applicable since they deal with functions and subroutines which are not part of our program model.

In a different direction, program reductions (beyond atomicity specifications) have been used to simplify concurrent and distributed program proofs by eliminating the need to reason about unbounded message buffers. In [Genest et al. 2007], the theory of Mazurkiewicz traces is used to define a category of distributed systems, modelled as automata communicating through channels, which are *existentially bounded*. *Natural proofs* [Desai et al. 2014] and *pretend synchrony*[von Gleissenthall et al. 2019] (among many more) use the same fundamental idea to simplify reasoning about distributed systems. For the programs which are targets of these approaches, large atomic blocks are not a reduction of choice since the aim of the reduction (i.e. program simplification) is to simplify the program from an asynchronous to almost synchronous.

Natural proofs [Desai et al. 2014] work for unbounded domains but boundedly many processes. *Pretend synchrony* [von Gleissenthall et al. 2019] provides an extension that works with unboundedly many processes by rewriting the program into an equivalent synchronous one. To make this possible, however, assumptions are made about loops (that there is no loop state) and round non-interference (no carried state between rounds). These are reasonable assumptions for distributed protocols but do not apply to concurrent message-passing programs. We also assume boundedly many processes, simply to be able to use finite state automata. Limited notions of context appear in some domain-specific reduction techniques. For example, in natural proofs [Desai et al. 2014], it matters whether a buffer is empty or not. Contextual reductions, however, are more general than context specific to buffers.

We emphasize that all these techniques are incomplete, even for the particular domains for which they were designed. Contextual reductions are also incomplete. As already noted in [Farzan and Vandikas 2019a], the problem of finding a reduction is as difficult as proving safety.

Partial Order Reduction

Partial-order reduction (POR) [Abdulla et al. 2014, 2017; Godefroid 1996] is a class of techniques that reduces the state space of search (for violation of a safety property) by removing redundant paths. POR techniques are concerned with finding a single (preferably minimal) reduction of (mostly) finite-state systems, and their primary application is in reachability/unreachability queries. We use the underlying ideas in POR in a non-standard way. The design of the LTAs that recognize S-reductions and C-reductions are informed by them.

Context has been incorporated into POR algorithms before. In [Godefroid and Pirottin 1993; Katz and Peled 1992], conditional dependence is used as a weakening of the independence relation to increase the potential for reduction. Conditional dependence adds a third component to the dependence relation, which is a (single) state. These techniques are exclusively applicable to finite-state systems. One can view one of the contributions of this paper as providing a way of lifting these ideas to infinite-state programs. In [Wang et al. 2008], the notion of *guarded dependence* is introduced which extends the state to a predicate (i.e. a set of states). This is then used to perform POR in the context of bounded symbolic model checking of finite state systems.

A language-theoretic notion of context has been previously studied in the context of models of concurrency [Sassone et al. 1993]. Our language-theoretic definition can be viewed as a weakening of that notion of Generalized Mazurkiewicz Trace Languages, which have additional *consistency* and *coherence* conditions on relation $\mathcal{I}$.

Partial order reduction has been combined with automated verification methods to tackle the large state space of multithreaded programs [Cassez and Ziegler 2015; Wachter et al. 2013; Wang et al. 2009]. In [Wachter et al. 2013], POR is combined with the classic IMPACT algorithm to lift it to concurrent programs. In [Cassez and Ziegler 2015], POR is applied to the concurrent control-flow automaton of the program to construct a reduced one, which is then used for proof construction/checking in a classic refinement algorithm in the style of [Heizmann et al. 2009]. Contexts do not play (a significant) role in pruning mechanisms of either of the approaches presented in [Cassez and Ziegler 2015; Wachter et al. 2013].

Relational and Hypersafety Verification

Program reductions have been used in relational and hypersafety verification [Barthe et al. 2011; Goguen and Meseguer 1982; Pnueli et al. 1998; Sabelfeld and Myers 2003; Sousa and Dillig 2016; Sousa et al. 2014] where reductions are applied to product programs to obtain simple proofs of relational/hyper- properties. The important observation is that since the copies of the program in such product programs are completely disjoint, the statements fully commute for the purpose of constructing a reduction. The contributions of this paper become significant when one does not have such a trivial commutativity relation. For example, if the goal is to prove a relational/hyper property of a concurrent program, where beyond the top-level product, commutativity specifications within a copy become relevant. We have examples of this (for instance proving the determinism of a concurrent program) among our benchmarks in Section 9. Neither of the approaches cited can handle concurrent programs.

The work in [Farzan and Vandikas 2019a] is the closest to ours in terms of methodology and in the fact that it handles concurrent programs. There, in the same style of refinement loop, the space of *non-contextual* reductions based on a *symmetric* dependence relation are explored for the purpose of verification of hypersafety properties of sequential and concurrent programs. Programs proved correct in this paper that require reasoning about semi-commutativity and contextual commutativity are theoretically beyond the scope of the algorithm presented in [Farzan and Vandikas 2019a].

11 CONCLUSION AND FUTURE WORK

The notion of *context* for program reductions had not received much attention before this paper. C-reductions provide a solution to incorporate contextual reductions in the automated program verification tool. The preliminary experimental results (Section 9) are promising. Nontrivial examples, that previously could not be proved automatically, can be verified using Siever . There is, however, much more work left ahead to explore the full potential of program reductions for automated program verification.

First, in Section 8, we presented algorithmic optimizations that ensure no additional complexity is incurred for contextual reductions over the simpler reductions of [Farzan and Vandikas 2019a]. The proof checking algorithm however still has a high complexity. This may be acceptable as a worst-case complexity for proof checking, but the construction of Section 8 implicitly requires a construction of the program control flow automaton, whose size is exponential on the number of threads. Consider the case of a very simple parallel program where all threads are *disjoint* and the postcondition refers only to the variables in a single thread. Siever can handle proving this program for a small number of threads, but as the number of threads grows, Siever 's verification time grows exponentially with it. It will be interesting to explore alternative ways of defining the reduction LTAs and/or the exploration algorithms to find solutions with better average case complexity when the number of threads grows but the verification task remains simple. For example, exploiting symmetry for replicated code is one possible avenue of investigation.

Second, it would be interesting to see how the idea of *abstraction* used by QED [Elmas et al. 2009] and CIVL [Hawblitzel et al. 2015] can be incorporated in the framework put forward by this paper to gain more powerful reductions. Briefly, a non-commuting statement can be abstracted in way a that the new program still satisfies the property of interest and the new abstract statement commutes against more statements than the old concrete one. These abstractions are suggested manually in [Elmas et al. 2009; Hawblitzel et al. 2015] and it will be interesting to investigate if the same insight can be inferred automatically.

Another limitation of the current approach is that it works for a fixed number of threads. It would be interesting to explore if *predicate automata* [Farzan et al. 2015] or *nominal automata* [Bojanczyk et al. 2014] can be used to formulate reductions for parameterized concurrent programs.

ACKNOWLEDGMENTS

This material is based upon work supported by National Science and Engineering Research Council of Canada (NSERC) Discovery Grant and Discovery Accelerated Supplement Award.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 373–384.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM* 64, 4 (2017), 25:1–25:49.

Franz Baader and Stephan Tobies. 2001. The Inverse Method Implements the Automata Approach for Modal Satisfiability. In *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings (Lecture Notes in Computer Science)*, Rajeev Goré, Alexander Leitsch, and Tobias Nipkow (Eds.), Vol. 2083. Springer, 92–106.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings (Lecture Notes in Computer Science)*, Michael J. Butler and Wolfram Schulte (Eds.), Vol. 6664. Springer, 200–214.

Mikolaj Bojanczyk, Bartek Klin, and Slawomir Lasota. 2014. Automata theory in nominal sets. *Logical Methods in Computer Science* 10, 3 (2014).

Franck Cassez and Frowin Ziegler. 2015. Verification of Concurrent Programs Using Trace Abstraction Refinement. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.), Vol. 9450. Springer, 233–248.

Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 709–725.

Volker Diekert and Yves Métivier. 1997. Partial Commutation and Traces. In *Handbook of Formal Languages, Volume 3: Beyond Words.*, Grzegorz Rozenberg and Arto Salomaa (Eds.). Springer, 457–533.

Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces.* World Scientific.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 2–15.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2013. Inductive data flow graphs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 129–142.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2015. Proof Spaces for Unbounded Parallelism. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 407–420.

Azadeh Farzan and Anthony Vandikas. 2019a. Automated Hypersafety Verification. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.), Vol. 11561. Springer, 200–218.

Azadeh Farzan and Anthony Vandikas. 2019b. Reductions for Safety Proofs (Extended Version). *CoRR* abs/1910.14619 (2019). arXiv:1910.14619 http://arxiv.org/abs/1910.14619

Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2005. Exploiting Purity for Atomicity. *IEEE Trans. Software Eng.* 31, 4 (2005), 275–291.

Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 338–349.

Blaise Genest, Dietrich Kuske, and Anca Muscholl. 2007. On Communicating Automata with Bounded Channels. *Fundam. Inform.* 80, 1-3 (2007), 147–167.

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science, Vol. 1032. Springer.

Patrice Godefroid and Didier Pirottin. 1993. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings (Lecture Notes in Computer Science)*, Costas Courcoubetis (Ed.), Vol. 697. Springer, 438–449.

Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982.* IEEE Computer Society, 11–20.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 449–465.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science)*, Jens Palsberg and Zhendong Su (Eds.), Vol. 5673. Springer, 69–85.

Shmuel Katz and Doron A. Peled. 1992. Defining Conditional Independence Using Collapses. *Theor. Comput. Sci.* 101, 2 (1992), 337–359.

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2018. Synchronizing the Asynchronous. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs)*, Sven Schewe and Lijun Zhang (Eds.), Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 21:1–21:17.

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science)*, Bernhard Steffen (Ed.), Vol. 1384. Springer, 151–166.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. 1993. Deterministic Behavioural Models for Concurrency. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings (Lecture Notes in Computer Science)*, Andrzej M. Borzyszkowski and Stefan Sokolowski (Eds.), Vol. 711. Springer, 682–692.

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 57–69.

Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. 2014. Consolidation of queries with user-defined functions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 554–564.

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL* 3, POPL (2019), 59:1–59:30.

Björn Wachter, Daniel Kroening, and Joël Ouaknine. 2013. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 210–217.

Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. 2009. Symbolic pruning of concurrent program executions. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, Hans van Vliet and Valérie Issarny (Eds.). ACM, 23–32.

Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 382–396.

Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. 2006. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science)*, Thomas Ball and Robert B. Jones (Eds.), Vol. 4144. Springer, 17–30.