Stratified Commutativity in Verification Algorithms for Concurrent Programs

AZADEH FARZAN, University of Toronto, Canada
DOMINIK KLUMPP, University of Freiburg, Germany
ANDREAS PODELSKI, University of Freiburg, Germany

The importance of exploiting *commutativity relations* in verification algorithms for concurrent programs is well-known. They can help simplify the proof and improve the time and space efficiency. This paper studies commutativity relations as a first-class object in the setting of verification algorithms for concurrent programs. A first contribution is a general framework for *abstract commutativity relations*. We introduce a general soundness condition for commutativity relations, and present a method to automatically derive sound abstract commutativity relations from a given proof. The method can be used in a verification algorithm based on abstraction refinement to compute a new commutativity relation in each iteration of the abstraction refinement loop. A second result is a general proof rule that allows one to combine multiple commutativity relations, with incomparable power, in a *stratified* way that preserves soundness and allows one to profit from the full power of the combined relations. We present an algorithm for the stratified proof rule that performs an optimal combination (in a sense made formal), enabling usage of stratified commutativity in algorithmic verification. We empirically evaluate the impact of abstract commutativity and stratified combination of commutativity relations on verification algorithms for concurrent programs.

CCS Concepts: • **Theory of computation → Program verification**; **Abstraction**; *Concurrency*.

Additional Key Words and Phrases: Commutativity, Partial Order Reduction

ACM Reference Format:
Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2023. Stratified Commutativity in Verification Algorithms for Concurrent Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 49 (January 2023), 28 pages. https://doi.org/10.1145/3571242

1 INTRODUCTION

In the verification of concurrent programs, *commutativity reasoning* has been known to be greatly beneficial since the seminal work by [Lipton 1975]. Commutativity can be employed to simplify programs [Elmas et al. 2009; Kragl and Qadeer 2018], resulting in programs that have simpler proofs [Farzan et al. 2022; Farzan and Vandikas 2019, 2020], and (in the setting of fully automated verification) it can decrease verification time and space complexity [Cassez and Ziegler 2015; Farzan et al. 2022; Wachter et al. 2013]. The underlying idea is that for certain pairs of statements, the order in which they are executed does not matter: Such statements *commute*. Two *interleavings* (sequences of statements from different threads) that only differ in the ordering between commuting statements can be considered *equivalent*. Consequently, it suffices to prove the correctness of one interleaving in order to deduce the correctness of the interleaving's entire equivalence class.

Authors' addresses: Azadeh Farzan, University of Toronto, Toronto, Canada, azadeh@cs.toronto.edu; Dominik Klumpp, University of Freiburg, Freiburg im Breisgau, Germany, klumpp@informatik.uni-freiburg.de; Andreas Podelski, University of Freiburg, Freiburg im Breisgau, Germany, podelski@informatik.uni-freiburg.de.

2475-1421/2023/1-ART49
https://doi.org/10.1145/3571242

A subset of program interleavings that covers all equivalence classes is called a *reduction*, and such a reduction soundly represents all program behaviors. It hence suffices to prove correctness of a reduction to conclude correctness of the program, and because they contain only a subset of interleavings, reductions can have simpler proofs and more compact representations.

Commutativity reasoning in this sense crucially relies on the precise notion of commutativity between two statements: What does it mean for two statements to commute? There exists a wide spectrum of (fundamentally different) possible answers to this question. As an example, *concrete commutativity* can be defined based on the concrete semantics of statements (as a transition relation between states): Two statements commute if executing them in either order defines the same transition relation.

Let us consider an example program that shows where concrete commutativity can be useful to construct a reduction, and under what circumstances it might be insufficient. Figure 1 shows a concurrent program with three threads. Threads T_1 and T_2 each compute a sum over an array A and store the result in variables x and y, respectively. After every iteration of the respective loops, information about the progress of the computation is printed. Thread T_3 injects noise into the computation by increasing the value of x and decreasing the value of y. The specification we want to prove for this program is that, if initially variables x and y are initialized to 0, then after all threads terminate, it holds that $x \geq y$.

Thread T_1:

```
i := 0;
while (i < N) {
  x := x + A[i];
  i := i + 1;
  print(i);
}
```

Thread T_2:

```
j := 0;
while (j < N) {
  y := y + A[j];
  j := j + 1;
  print(j);
}
```

Thread T_3:

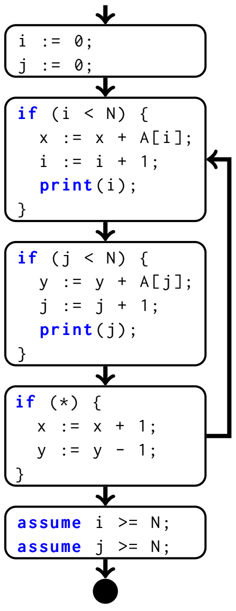
```
while (*) {
  x := x + 1;
  y := y - 1;
}
```

Procedure `print`:

```
print(val) {
  atomic {
    buf[ptr] := val;
    ptr := ptr + 1;
  }
}
```

Fig. 1. Example.

Verifying that the example program satisfies this specification presents a challenge to verification algorithms, which have to find the loop invariants automatically. When all interleavings are considered, the proof requires complex assertions: Among others, the verification algorithm must find an invariant of the form $x \geq \sum_{k=0}^{i} A[k] \wedge y \leq \sum_{k=0}^{j} A[k]$. Complex assertions like these are out of reach for verification algorithms.



```
i := 0;
j := 0;
```

```
if (i < N) {
  x := x + A[i];
  i := i + 1;
  print(i);
}
```

```
if (j < N) {
  y := y + A[j];
  j := j + 1;
  print(j);
}
```

```
if (*) {
  x := x + 1;
  y := y - 1;
}
```

```
assume i >= N;
assume j >= N;
```

Fig. 2. Reduction

Commutativity (specifically, *concrete commutativity*) allows us to reduce the program: The statements of thread T_3 commute (concretely) against all statements of threads T_1 and T_2. We can thus focus on a reduction where we first consider all interleavings of threads T_1 and T_2, and only after both these threads have terminated, we allow T_3 to execute. This reduction is the sequential composition of the parallel execution of T_1 and T_2 with the thread T_3, i.e., $(T_1 \parallel T_2);T_3$. Any interleaving of the concurrent program is equivalent (up to concrete commutativity) to an interleaving in this reduction. Now, it is possible to prove the postcondition $x \geq y$ for threads T_1 and T_2, and then show that the assertion $x \geq y$ is preserved by the subsequent (sequential) execution of thread T_3.

However, to account for all interleavings of $T_1 \parallel T_2$, we still need a complex loop invariant. Since the statements `print(i)` of thread T_1 and `print(j)` of thread T_2 do not commute concretely, no significant further reduction is possible. Consequently, verification algorithms still will fail to verify this reduction, which takes advantage of concrete commutativity.

This is unsatisfying. If the statements `print(i)` and `print(j)` were allowed to commute, we could find a reduction with a simple proof. Figure 2 illustrates a subset of the interleavings of our example program, namely the interleavings where the loops in each thread are executed in lockstep (each loop performs one iteration in turn). Under the assumption that the statements `print(i)` and `print(j)` commute, this subset of interleavings is a reduction of the concurrent program in Figure 1. And this reduction has a simple proof: The entire program in Figure 2 can be proven correct using (conjunctions of) the assertions $i = 0$, $i = j$, $i = j + 1$, $x \geq y$, $x \geq y + A[j]$, $i \leq N$, $i > N$, $j \leq N$ and $j > N$. All of these assertions are simple enough to be discovered by a verification algorithm.

Intuitively, it should be sound to allow the statements `print(i)` and `print(j)` to commute: Their relative ordering is irrelevant to the correctness property. In fact, an even stronger condition holds: The relative ordering of these statements is irrelevant for the set of assertions given above (i.e., the set of assertions in the proof for the reduction in Figure 2).

This paper builds on the observation that the success of the verification algorithm depends crucially on the particular choice of the relation used to define commutativity. Once we depart from concrete commutativity, there is a large spectrum for notions of *abstract* commutativity. The crucial issue here is the soundness of abstract commutativity: The correctness of the interleavings in the reduction must still imply the correctness of all interleavings of the program. In this paper, we base definitions of abstract commutativity (as well as the notion of its soundness) on a given proof for a reduction. We observe an apparent circular dependency (abstract commutativity is based on a proof for a reduction which refers to abstract commutativity) which is vaguely reminiscent of rely-guarantee reasoning: The proof relies on the reduction defined by the abstract commutativity relation, and in turn the proof guarantees soundness of the abstract commutativity relation.

The idea of abstract commutativity has been used in the literature [Elmas et al. 2009; Kragl and Qadeer 2018]. Our shared view is that abstraction can help increase commutativity. However, in [Elmas et al. 2009; Kragl and Qadeer 2018], the increased commutativity is chiefly used for the construction of larger atomic blocks to exploit local reasoning. Our thesis is that abstract commutativity can be significantly useful in proving properties of concurrent programs that concern only a relatively small slice of the program: for example, properties local to a thread, or safety of memory accesses, or lightweight properties like race detection. The smaller the slice, the higher the potential for abstract commutativity, namely for many statements outside the relevant slice. This in turn results in a reduction with a substantially smaller representation to be used by the verification algorithm. In these cases, abstract commutativity significantly improves the time and space efficiency of algorithmic verification. In contrast to (semi-)interactive settings such as [Elmas et al. 2009; Kragl and Qadeer 2018], where soundness of the abstract commutativity is left to the user, for a verification algorithm it must hold by construction.

As a first contribution, the paper proposes a framework that allows us to investigate the commutativity relation as a parameter to a verification algorithm. We define general soundness conditions for commutativity relations in this framework to ensure correctness of the verification is preserved. We introduce several *abstract* commutativity relations that can be *constructed automatically* to admit algorithmic verification, and we investigate their soundness. We show empirically that even coarse abstractions have significant benefit for the verification algorithm.

Concrete commutativity and abstract commutativity are incomparable in power; i.e., one does not subsume the other. They are both useful, and they come with different strengths. Coming back to our example, we note that the two statements `print(i)` and `print(j)` commute under a suitable abstract commutativity relation, because the property of interest is blind to the order in which data is printed (formally, to the final value of the variable `buf`). Hence, if we could combine abstract commutativity with the concrete commutativity discussed before, we could conclude that Figure 2 soundly represents all program interleavings. Thus, the combination of abstract and concrete commutativity would lead to the simple proof we outlined.

The example thus highlights a challenge for verification algorithms: Neither the concrete nor the abstract commutativity relation alone may be sufficient for the purpose of constructing a simple proof. Particularly, neither concrete commutativity in isolation nor the abstract commutativity presented in this paper in isolation allows us to reduce the example program to the reduction illustrated in Figure 2 and thus have a simple proof. The question is whether one can leverage the power of both the concrete and the abstract commutativity relation, to construct a simple proof, in a verification *algorithm*: Unlike in an interactive setting, the algorithm cannot rely on the user to intervene and decide where to apply concrete or abstract commutativity. We show that this problem is challenging by demonstrating that several straightforward approaches (such as taking the union of the commutativity relations) are either unsound or unsuitable for verification algorithms.

As second contribution, we introduce the concept of *stratified reduction*, as a way to take advantage of multiple incomparable notions of commutativity. We introduce a stratification-based proof rule that combines commutativity relations on a declarative level, and show soundness of this proof rule. Our proof rule is general in nature, and can potentially be applied in many settings, not only in algorithmic verification.

Intuitively, a stratified reduction amounts to applying the two commutativity relations in strata: To transform a given interleaving of the threads to one in the reduction, one can swap statements according to the concrete commutativity relation arbitrarily often, but only until one starts swapping statements according to the abstract commutativity relation (again, arbitrarily often). Formally, a stratified reduction defines a subset of interleavings such that, for every interleaving, there exists a *stratified* sequence of swaps (with the choice of the point of switch from the stratum of concrete swaps to the stratum of abstract swaps) that transforms the interleaving into an interleaving in the reduction.

In a sense, the stratified reduction is a more powerful combination of concrete and abstract commutativity than previously found in the literature [Elmas et al. 2009; Kragl and Qadeer 2018]: There, each statement is *either* considered under concrete commutativity, *or* it is abstracted and then always considered under abstract commutativity. By contrast, a stratified sequence of swaps allows a single statement to be swapped with other statements under concrete as well as under abstract commutativity. The only restriction is that swaps up to concrete commutativity must precede swaps up to abstract commutativity; arbitrarily interleaved swaps (as possible with the union of commutativity relations) lead to unsoundness.

We investigate the concept of a stratified reduction for the pair of two commutativity relations, where the first is the concrete commutativity relation and second is the abstract commutativity relation. We establish that the same principle applies to the general case of a stratified reduction for a tuple of n commutativity relations (where the sequence of commutativity relations in the tuple is again ordered, with the first one being more abstract than the second etc.).

In the context of algorithmic verification, the question arises whether there exists an algorithm that constructs a stratified reduction. Formally, given a candidate proof, and given a sound abstract commutativity relation wrt. the proof, the algorithm has to compute an effective representation of the corresponding subset of interleavings. Effectiveness here refers to the check of the validity of a candidate proof (not for all interleavings but) for the corresponding subset of interleavings.

As the third main contribution of the paper, we present such an effective construction for stratified reduction, for which we prove that it is optimal (in a sense that is made formal): Intuitively, it squeezes the last bit of theoretically feasible advantage from the combination. The key insight from this construction is that the optimal selection of "switch points" between strata (i.e., between different commutativity relations) for an infinite number of traces in an infinite number of equivalence classes can be finitely represented. We also present a schematic algorithm that allows for the exploration of other selections of switch points which may sacrifice theoretical optimality for practical performance, while still guaranteeing soundness.

We have implemented a verification algorithm that can be parametrized in the reduction for the concrete commutativity relation, or the sequence of reductions for the sequence of abstract commutativity relations (one for each of the candidate proofs constructed during the execution of the verification algorithm), or the stratified reduction (as discussed above). The experimental results indicate the potential practical value of our main contributions.

Contributions. To summarize, our conceptual contribution is to single out the importance of the commutativity relation as a parameter in verification algorithms for concurrent programs. Previous work [Farzan et al. 2022; Farzan and Vandikas 2019, 2020] has shown the potential in a systematic investigation of reductions through different choices of representative interleavings. However, these works (like many others) always assume a fixed commutativity relation underlying the reduction. The commutativity relation is never investigated *in its own right*, though the paramount importance of its role is obvious – perhaps because it has not been considered that there exists a wide (and systematically describable) space of choices. This can be put in stark contrast to, e.g., static analysis or, formally, abstract interpretation, where the central parameter (the abstract domain) is investigated in its own right.

Our first technical contribution is a general soundness condition for commutativity relations that allows them to be soundly used in verification algorithms based on abstraction refinement. We present a method to derive sound commutativity relations as well as two instances, and empirically show the benefit of abstract commutativity.

Our second technical contribution is a new proof rule based on *stratified commutativity* that combines multiple incomparably powerful commutativity relations in a way that preserves soundness and allows one to profit from the full power of the combined relations.

Finally, our third technical contribution is an algorithmic realization of stratified commutativity, allowing its usage in algorithmic verification. We prove that our algorithm is theoretically optimally (in a sense made formal). We show the practical potential of this combination approach in an empirical evaluation.

2 BACKGROUND: CONCURRENT PROGRAMS AND COMMUTATIVITY

Let us fix the basic notions concerning concurrent programs and commutativity.

Programs. We assume that **Stmt** denotes the set of all atomically executed program statements, e.g. assignments like `x:=y+1`, nondeterministic updates `x:=*` and conditions / assume statements like `x>=10` and `x==y+1` (we use different colors to indicate that statements belong to different threads). An assume statement like `x>=10` blocks unless the condition is fulfilled. A trace τ is a sequence of statements; i.e., $\tau \in \mathbf{Stmt}^*$. A program P is a set of traces; i.e., $P \subseteq \mathbf{Stmt}^*$. Intuitively, we identify a program P with the set of traces that correspond to paths in the program's control flow graph (a path may not correspond to any actual execution; in this case, the corresponding trace τ is *infeasible*).

In this work we are concerned with *concurrent* programs P. That is, P is the set of traces that correspond to all interleavings of traces taken from the control flow graphs of individual threads. We require that the set of traces representing P forms a regular language. We address programs with a *bounded* number of threads. Our model is general enough to accommodate concurrent programs that dynamically fork and join threads (as long as the number of threads can be bounded). Throughout the paper, we assume that the program P is fixed.

We sometimes use a deterministic finite automaton (DFA) to represent the program P. A state q of the DFA is then called a *location*, and encodes the program counter of all running threads. Our approach does not require the DFA to be explicitly constructed (it is exponentially large in the number of threads); it suffices that the DFA's transition function can be computed. It will always be clear from the context whether P refers to a DFA or a set of traces.

Each statement $st \in \mathbf{Stmt}$ is associated with a *semantics* $[\![st]\!]$, a binary relation between states of the program P (i.e., variable assignments). We extend the semantics function from statements $st \in \mathbf{Stmt}$ to traces $\tau \in \mathbf{Stmt}^*$ through relational composition in the usual way, i.e., $[\![\tau\, st]\!] = [\![\tau]\!] \circ [\![st]\!]$ and $[\![\varepsilon]\!] = \mathrm{id}$. We use the standard notation for (valid) Hoare triples $\{\varphi\}\; st\; \{\psi\}$ for a statement st and for assertions φ and ψ (similarly $\{\varphi\}\; \tau\; \{\psi\}$ for a trace τ) .

Correctness. Throughout the paper, we assume that the specification of correctness for the program P is given as a fixed precondition-postcondition pair $(\varphi_{\text{pre}}, \varphi_{\text{post}})$ of assertions. We use **correct** for the set of all traces that are correct; i.e., $\mathbf{correct} = \{\tau \in \mathbf{Stmt}^* \mid \{\varphi_{\text{pre}}\}\; \tau\; \{\varphi_{\text{post}}\}\}$. Then the program P is *correct* if all program traces $\tau \in P$ are correct, i.e., if $P \subseteq \mathbf{correct}$.

The setting is general enough to accommodate correctness specified by an *assert statement* in the program. In this case, we define P as a set of *error traces* and specify correctness by the precondition-postcondition pair $(\top, \bot)$. Correctness then states the *infeasibility* of each error trace. Intuitively, an error trace corresponds to a path leading to an auxiliary *error location*; an error trace always ends with the assume statement `!e` which uses the negation of the expression e in the assert statement.

We distinguish the notion of a *proof* and the notion of a *proof for the program P*. A *proof* Π is simply a set of valid Hoare triples over statements. We often identify Π with the set of traces which can be proven correct by combinations of the Hoare triples in Π. It will be clear from context whether Π refers to a set of Hoare triples or to a set of traces.

The proof Π is a *proof for the program P* if all traces of P can be proven correct by Π, i.e., if $P \subseteq \Pi$ (which is decidable, via DFAs). Not every proof Π is a proof for the program P. The existence of a proof for P implies that P is correct (by transitivity of inclusion).

Commutativity and Closure. A *commutativity relation I* is a symmetric binary relation over statements (in the literature, I is often referred to as *independence* relation). Intuitively, membership of a pair (st_1, st_2) in a commutativity relation I captures that in some sense (which sense precisely depends on I), the ordering in which st_1 and st_2 are executed "does not matter". Formally, this is captured by the *Mazurkiewicz equivalence* $\sim_I$ induced by I, i.e., an equivalence relation over traces defined as the least congruence $\sim_I$ such that $(a, b) \in I$ implies $ab \sim_I ba$ (the congruence refers to the monoid $\mathbf{Stmt}^*$; i.e., if $ab \sim_I ba$ then $u\,ab\,v \sim_I u\,ba\,v$ for all sequences u and v in $\mathbf{Stmt}^*$).

We denote by $cl_I(L)$ the *closure* of a set of traces L under the equivalence relation $\sim_I$, i.e., the set of all traces that are equivalent to some trace in L.

3 ABSTRACT COMMUTATIVITY

A suitable notion of commutativity between statements is a key ingredient in many verification algorithms for concurrent programs. In this section, we discuss several possible ways to define notions of commutativity. In particular, we investigate a wide spectrum of *commutativity relations* that allow for proof simplification and efficiency gains in proof checking. In contrast to previous work, our approach is *parametrized* in a commutativity relation (or in fact, multiple commutativity relations), and clearly identifies the properties such relations must satisfy to guarantee soundness.

The first commutativity relation we define is based on the concrete semantics of statements. If executing the statements in either order has the exact same effect (the same semantics), then we declare the statements commutative. Formally, this relation is defined by

$$I_C := \{ (st_1, st_2) \in \textbf{Stmt} \times \textbf{Stmt} \mid [\![st_1 st_2]\!] = [\![st_2 st_1]\!] \}$$

A sufficient condition for two statements to commute under the *concrete commutativity* I_C is that neither statement writes to a variable read or written by the other statement.

The concrete commutativity relation I_C can be soundly used in the verification of any program and for any safety property: Correctness of a trace implies that all traces equivalent to I_C are also correct. Consequently, it suffices to prove correctness of one representative trace in each equivalence class to conclude correctness of the entire program.

As explained in the introduction, concrete commutativity does not take into account the program being verified, nor the property being proven. We go beyond this simple notion of commutativity and, given a program, a property to be proven, and a candidate proof Π for this program and property, we define commutativity relations that take Π into account and are *safe wrt.* Π. Under such commutativity relations, statements may commute that do not commute concretely. We call such commutativity relations *abstract* commutativity relations, to reflect that they are not tightly bound to the statements' concrete semantics but rather to the abstraction inherent in the proof Π.

Definition 3.1 (Proof-Specific Safety). *Given a proof* Π*, a commutativity relation* I *is* safe *wrt.* Π *if the closure of* Π *under the Mazurkiewicz-equivalence* $\sim_I$ *contains only correct traces.*

$$cl_I(\Pi) \subseteq \textbf{correct}$$

Safety of a commutativity relation I wrt. a proof Π means that whenever a trace τ has a proof in Π and τ is equivalent to τ' up to I ($\tau \sim_I \tau'$), then τ' satisfies the given correctness property as well – even though τ' may not have a proof in Π. Our observation above that concrete commutativity can be soundly used for any program and property can now be restated: I_C is safe wrt. any proof Π. The fact that safety is in general relative to the proof is crucial in this work, it is precisely this restriction that allows us to let statements commute that do not commute concretely.

In a verification algorithm based on abstraction refinement, we begin with an empty proof ($\Pi = \emptyset$). Every commutativity relation is safe wrt. this proof. The algorithm then successively constructs a proof that covers more and more correct traces. For each proof, we have a new notion of safety. The following proof rule allows us to conclude correctness of the verified program and terminate the verification.

Proposition 3.2 (Commutativity Proof Rule). *If the commutativity relation* I *is safe wrt. the proof* Π*, and all traces of a program* P *are in the closure of* Π*, then the program* P *is correct.*

$$P \subseteq cl_I(\Pi) \implies P \textit{ is correct}$$

The proof rule states that, for any safe commutativity relation I, it is sufficient if each program trace is equivalent (up to I) to a trace proven by Π in order to conclude that the program is correct; just like for concrete commutativity above.

Next, we present a general scheme to derive a safe commutativity relation from a proof Π, based on statement abstractions. In Section 3.2 we instantiate the scheme to a particular statement abstraction that we use for examples throughout the paper and in the evaluation. We conclude Section 3 with the presentation of another safe commutativity relation in Section 3.3 which does not fall into the scheme of Section 3.1.

3.1 Commutativity of Abstractions

One approach to derive a commutativity relation is via *statement abstraction*: Consider two statements that do not commute concretely, i.e., the order in which they are executed affects the behaviour. Intuitively, this difference in behaviour is often irrelevant to the correctness of the program. Statement abstraction allows us to formally capture this intuition, and eliminate such dependencies between statements. In particular, we here consider functions $\alpha : \mathbf{Stmt} \to \mathbf{Stmt}$ that map program statements to other program statements. We call α a *statement abstraction* if it is conservative, i.e., the semantics of the abstracted statement allows all behaviours that are part of the original semantics.

$$[\![st]\!] \subseteq [\![\alpha(st)]\!] \text{ for all } st \in \mathbf{Stmt}$$

As an intuition, a nondeterministic assignment `x:=*` is a conservative abstraction of a (deterministic) assignment `x:=y`; and the statement `x>=0` is a conservative abstraction of the statement `x>=10`. On the other hand, `x>=10` would not be a conservative abstraction of the statement `x:=10`. Given a statement abstraction α, we define a commutativity relation I_C^α, building on the notion of concrete commutativity:

Definition 3.3 (Commutativity induced by Statement Abstraction). *Let α be a statement abstraction. We define I_C^α as the set of all pairs of statements (st_1, st_2) where $(\alpha(st_1), \alpha(st_2)) \in I_C$.*

To determine if two statements *commute under* α (or simply, *commute abstractly*), we check if their respective abstractions (as given by α) commute concretely. Because α changes the semantics of statements, these abstractions may commute when the original statements do not. Naturally, we require certain conditions on the abstraction α to guarantee that the induced commutativity is safe.

Definition 3.4 (Proof-Preserving Abstraction). *Given a proof Π, we say that a statement abstraction α* preserves Π *(α is* proof-preserving*) if for each Hoare triple $\{\varphi\}\ a\ \{\psi\}$ in Π, the Hoare triple $\{\varphi\}\ \alpha(a)\ \{\psi\}$ is also valid.*

Intuitively, proof preservation states that we must make sure to not over-approximate the behaviour of statements so far as to invalidate the proof. This is particularly crucial in an automated setting: In an interactive setting (such as [Elmas et al. 2009]), we could leave it to the user to choose a statement abstraction, and accordingly make the user responsible for choosing the abstraction such that the abstracted program still satisfies the specification they wish to verify. In the automated setting, it instead falls to the approach itself to ensure soundness. We here use the proof as a "guard rail" against abstracting too far. Proof-preservation is sufficient to ensure safety of the induced commutativity relation:

Theorem 3.5 (Proof-Preservation implies Safety). *If the statement abstraction α preserves the proof Π, then the induced commutativity relation I_C^α is safe wrt. to Π.*

Proof. Given traces τ_1, τ_2 with $\tau_1 \in \Pi$ and $\tau_1 \sim_{I_C^\alpha} \tau_2$, we must prove that $\tau_2 \in \mathbf{correct}$. By proof preservation, the statement-wise abstracted trace $\alpha(\tau_1)$ can be proven correct using the assertions from Π (just like τ_1). We have $\alpha(\tau_1) \sim_{I_C} \alpha(\tau_2)$, and because $\sim_{I_C}$ preserves correctness, this implies $\alpha(\tau_2) \in \mathbf{correct}$. But because α is conservative, $[\![\tau_2]\!] \subseteq [\![\alpha(\tau_2)]\!]$, and so $\tau_2 \in \mathbf{correct}$. □

To give a general perspective: The abstraction refinement loop of a verification algorithm generates various proofs. Given a scheme to construct a statement abstraction α from the current proof, such that α is (by construction) proof-preserving, the algorithm may then use the proof rule above (Proposition 3.2) with the commutativity induced by α to determine if the verification can terminate and conclude that the analyzed program is correct.

3.2 Case Study: Projection to the Proof

In this section, we present a scheme to construct statement abstractions from a given proof, and we investigate the commutativity induced (in the sense of Section 3.1) by the constructed statement abstractions. This commutativity relation is used in examples throughout the paper, and forms the basis for our evaluation.

Given a proof Π, we define a statement abstraction that can be automatically constructed through *projection to the proof* Π. Intuitively, the idea behind projection to the proof is this: Suppose two statements, say `x:=17` and `x:=y`, do not commute under the concrete commutativity relation I_C. In a sense, we can say that a *reason for non-commutativity* is the variable x: Depending on the order in which the statements are executed, the final value of x changes. However, if we observe that the proof "does not care" about the value of x, then we might reasonably expect the two statements to commute under a proof-specific commutativity relation. Formally, if none of the proof assertions contains x as a free variable, the idea is to abstract the statements `x:=17` and `x:=y` such that the abstracted statements commute.

Consider another example: The statements `x:=17` and `y:=2*x` do not commute under the concrete commutativity I_C. The variable x is again involved in causing this non-commutativity, but in a different sense than before: Depending on the ordering of the two statements, the data flow through x changes, affecting the final value of y. Once again, we can imagine a setting in which the proof does not constrain the value of x at any point (but the value of y is constrained, perhaps through an assertion $y \neq 1$). We expect the two statements to commute wrt. to such a proof.

The statement abstraction defined through projection to the proof allows the statements in both of these examples to commute. We achieve this by replacing every occurrence of x (the variable not mentioned in the proof) in a statement by a non-deterministically chosen value.

For the formal presentation, it is convenient to represent statements through logical formulae. As is standard, we translate statements to *transition formulae*, first-order formulae over logical variables x and x' for every program variable x. For instance, the statement `x:=x+1` is expressed through the transition formula $x' = x + 1 \land y' = y$; and the compound statement `x==0;x:=y` is expressed as $x = 0 \land x' = y \land y' = y$ (assuming x and y are the only program variables). The statement abstraction by projection to a proof is then defined as follows:

Definition 3.6 (Projection to a Proof). *Given a proof* Π, *let* Y *be the set of all free variables occurring in assertions in* Π, *along with their primed version* ($Y = fvars(\Pi) \cup \{v' \mid v \in fvars(\Pi)\}$). *We define the* proof-projection α_Π to Π *of a transition formula* tf *as the projection of* tf *to* Y, *i.e., the existential quantification of all other free variables.*

$$\alpha_\Pi(tf) := \exists_{-Y} . tf$$

The abstraction existentially quantifies all (input or output) variables in the transition formula whose corresponding program variable does not appear in the proof Π. Intuitively, the existential quantification of an input variable x means that the abstracted transition formula reads a nondeterministic value rather than the value of x. The quantification of an output variable x' means that the abstracted transition formula modifies the value of program variable x to a nondeterministically chosen value.

Example 3.7. Consider the statements in the explanation above. We assume that the only Hoare triple used by Π is the triple $\{\top\}$ `y:=2*x` $\{y \neq 1\}$. The statement `x:=17` is abstracted to the nondeterministic assignment `x:=*`. The statement `x:=y` is similarly abstracted to `x:=*`. Finally, `y:=2*x` is abstracted to the statement `x:=* ; y:=* ; y%2==0`, which nondeterministically assigns both x and y but preserves the constraint that the new value of y is even. Below, we give the representation of both the original and the abstracted statements as transition formulae.

$$\begin{array}{llll}
\texttt{x:=17} \equiv x' = 17 \land y' = y & \alpha_\Pi(\texttt{x:=17}) & \equiv \exists x, x' \,.\, x' = 17 \land y' = y & \equiv\ y' = y \\
\texttt{x:=y} \equiv x' = y \land y' = y & \alpha_\Pi(\texttt{x:=y}) & \equiv \exists x, x' \,.\, x' = y \land y' = y & \equiv\ y' = y \\
\texttt{y:=2*x} \equiv x' = x \land y' = 2x & \alpha_\Pi(\texttt{y:=2*x}) & \equiv \exists x, x' \,.\, x' = x \land y' = 2x & \equiv\ 2 \mid y'
\end{array}$$

The first two statements are abstracted to the same statement, and trivially commute under this abstraction (either ordering of the abstracted statements has the same semantics). The abstracted second and third statements also commute: The second statement preserves the value of y (while nondeterministically updating x), and the third statement updates y to a nondeterministically chosen even value (and also nondeterministically updates x).

Closer examination of the abstracted statements also shows that they are conservative abstractions of the original statements, and they preserve the Hoare triple used by the proof. In fact, this holds for all statements and all proofs Π:

Proposition 3.8. *Let Π be a proof. Then the commutativity relation $I_C^{\alpha_\Pi}$ induced by proof-projection to Π is safe wrt. Π.*

Proof. We first show that proof-projection α_Π is a conservative statement abstraction, and that it preserves the proof Π (Definition 3.4). Then we apply Theorem 3.5 to conclude the result. □

Consequently, one might think it is a good idea to employ the commutativity induced by proof-projection as a drop-in replacement for the concrete commutativity relation: We can guarantee that the commutativity relation is always safe, and we have seen that in some cases it allows more statements to commute (Example 3.7). However, there exist statements that commute concretely but do not commute under proof-projection.

Example 3.9. Consider the statements A[i]:=3 and A[i+1]:=4. These statements commute concretely, because $i \neq i + 1$ always holds and hence disjoint portions of the array A are updated. However, if the proof does not make reference to the variable i (although it does reference A), the statements are abstracted to A[*]:=3 and A[*]:=4, respectively: statements that update the array at nondeterministically chosen positions. Hence one update could overwrite the change made by the other. Thus the order in which the abstracted statements are executed affects the semantics, and they do not commute. In transition formula notation:

$$\begin{array}{rl}
\texttt{A[i]:=3} & \equiv A' = A[i \triangleleft 3] \land i' = i \\
\texttt{A[i+1]:=4} & \equiv A' = A[i+1 \triangleleft 4] \land i' = i \\
\alpha_\Pi(\texttt{A[i]:=3}) & \equiv \exists i, i' \,.\, A' = A[i \triangleleft 3] \land i' = i \qquad \equiv \exists v \,.\, A' = A[v \triangleleft 3] \\
\alpha_\Pi(\texttt{A[i+1]:=4}) & \equiv \exists i, i' \,.\, A' = A[i+1 \triangleleft 4] \land i' = i \equiv \exists v \,.\, A' = A[v \triangleleft 4]
\end{array}$$

As witnessed by Example 3.7 and Example 3.9, concrete commutativity and commutativity induced by statement abstraction (such as proof-projection) are incomparable: In general, neither relation subsumes the other. Section 5 introduces our approach to obtain the benefits from both. Section 7 shows how commutativity induced by proof-projection behaves in a practical evaluation.

3.3 Case Study: Proof-Stuttering Commutativity

While (proof-preserving) statement abstractions represent a general recipe for abstract commutativity relations, there is a wider space of commutativity relations that are safe wrt. a proof. To showcase this, we present such a commutativity relation *not* based on statement abstractions in this section. The underlying idea is as follows: When verifying a program, certain statements may turn out to be completely irrelevant to the proof of correctness. Specifically, we may observe that such statements do not contribute anything to the proof (they do not modify or constrain variables in a way useful to establish correctness), nor do they disturb the proof (they do not invalidate proof assertions on the program variables established by other statements). Such statements can be executed at any point in the program and any number of times (or even omitted), without affecting the proof of correctness. These statements may implement non-functional aspects of the program (say, logging), or the verified specification may not express full functional correctness.

Definition 3.10 (Stuttering modulo Proof). *Given a proof* Π, *we call a statement* st stuttering modulo Π *if the following two conditions hold:*

- *If* $\{\varphi\}\ st\ \{\psi\}$ *is a Hoare triple used by the proof* Π, *then it holds that* $\varphi \models \psi$.
- *If the assertion* φ *is used by the proof* Π, *then the Hoare triple* $\{\varphi\}\ st\ \{\varphi\}$ *is valid.*

Let $\mathbf{Stutter}_\Pi$ *denote the set of all such statements.*

Example 3.11. Consider a proof Π that uses only the three Hoare triples $\{\top\}$ `x:=y` $\{x \geq y\}$, $\{x \geq y\}$ `x:=x+1` $\{x \geq y\}$ and $\{x \geq y\}$ `x<y` $\{\bot\}$. The statement `x:=x+1` is stuttering modulo Π.

To capture the idea that the ordering of stutter statements relative to other statements does not matter, we define the following commutativity relation:

Definition 3.12 (Stutter Commutativity). *Given a proof* Π, *the* stutter commutativity *relation* I_S *is the set of all pairs of statements, where at least one of the two statements is stuttering modulo* Π.

$$I_S = \{\ (st_1, st_2) \in \mathbf{Stmt} \times \mathbf{Stmt} \mid st_1 \in \mathbf{Stutter}_\Pi \vee st_2 \in \mathbf{Stutter}_\Pi\ \}$$

Example 3.13 (continued from Example 3.11). Up to the stutter commutativity relation I_S, the statement `x:=x+1` commutes against both the statement `x:=y` and against the statement `x<y`.

Though the stutter commutativity relation I_S is not defined via a statement abstraction (following the scheme introduced in Section 3.1), we can show that it is indeed safe (Definition 3.1):

Proposition 3.14 (Stutter Commutativity Safety). *Given a proof* Π, *the stutter commutativity relation* I_S *is safe wrt.* Π.

Proof. Given traces τ_1, τ_2 with $\tau_1 \in \Pi$ and $\tau_1 \sim_{I_S} \tau_2$, we must prove that $\tau_2 \in$ **correct**. Let Π^+ be the Hoare triples from Π enriched with all valid Hoare triples of the form $\{\varphi\}\ st\ \{\varphi\}$ for all assertions φ used by Π and all statements st. We know that τ_1 can be proven correct using the Hoare triples in Π^+. By induction over the number of swaps, we show that the same holds for all traces equivalent to τ_1: The assertions before and after a stuttering statement are the same, and a swap simply means one assertion is no longer repeated, and another is instead repeated once more. □

Stutter commutativity is in general incomparable with concrete commutativity as well as commutativity induced by projection to the proof. For instance, the statements `i:=3` and `j:=3` commute concretely, but under a proof using the Hoare triples $\{\top\}$ `i:=3` $\{i = 3\}$ and $\{i = 3\}$ `j:=3` $\{i = j\}$, neither of the statements is stuttering.

Stutter commutativity can help in efficiently checking a proof, intuitively because it allows the proof check to soundly prune parts of the state space. However, stutter commutativity is not very useful for simplifying proofs: All traces equivalent to a representative proven correct by a proof Π can be proven correct using the same assertions (see the proof sketch for Proposition 3.14). Only the number of required Hoare triples may be lower.

4 THE CHALLENGE OF COMBINING COMMUTATIVITY RELATIONS

We introduced two instances of safe commutativity relations. Since both turned out to be incomparable with concrete commutativity, the question arises: How can we gain the full benefit of both concrete and abstract commutativity? It turns out that such a combination is challenging: Several reasonable approaches do not provide satisfying solutions. The subsequent sections then present our combination approach and show that it does not suffer from the same limitations.

For the remainder of this section, we assume a fixed proof Π, and let $I^\flat$ and $I^\sharp$ be commutativity relations that are safe wrt. Π. Intuitively, we have in mind the case where $I^\flat = I_C$ is the concrete commutativity, and $I^\sharp$ is an abstract commutativity relation (such as presented in Section 3.2 or Section 3.3). Ideally we want a combination of these commutativity relations such that:

- **(G1)** The combination allows us to *soundly* conclude program correctness. The approaches presented in this section combine the relations $I^\sharp$ and $I^\flat$ into a single commutativity relation, hence we can use *safety wrt. the proof* Π to judge the combined commutativity relation.
- **(G2)** The combination allows us to *gain the full benefits* of both relations. As a minimum, we require that, if our proof rule (Proposition 3.2) instantiated with either commutativity relation $I^\sharp$ or $I^\flat$ alone allows us to conclude that the program is correct, then the combination should also allow us to conclude correctness.
- **(G3)** The combination is suitable for *automated verification*. That is, it does not require user input to determine which commutativity relation is applied in each context.

The most straightforward idea to combine the two commutativity relations $I^\flat$ and $I^\sharp$ would be to take their union. Unfortunately, the union of two safe commutativity relations can be unsafe:

Observation 4.1. *Even if the commutativity relations $I^\sharp$ and $I^\flat$ are safe wrt. Π, the commutativity relation $I^\sharp \cup I^\flat$ obtained by union is not necessarily safe wrt. Π.*

Example 4.2. Consider a program with three threads, in which thread 1 only executes the statement `x:=1`, thread 2 executes `x==1;z:=1`, and thread 3 executes `x==2;z:=2`. We want to prove the postcondition $z = 2$, and the current proof Π is given by the Hoare triples $\{\top\}$ `x:=1` $\{\top\}$, $\{\top\}$ `x==1;z:=1` $\{\top\}$, and $\{\top\}$ `x==2;z:=2` $\{z = 2\}$.

The statements `x==1;z:=1` and `x==2;z:=2` commute concretely: Executing them in either order is infeasible, because the guard of the statements are contradictory. On the other hand, the statements `x:=1` and `x==2;z:=2` commute under proof-projection: Since x does not appear in the proof, the guard x==2 is abstracted away. Consequently, if $J = I_C \cup I_C^{\alpha_\Pi}$, we have

$$\texttt{x:=1}\ \texttt{x==1;z:=1}\ \texttt{x==2;z:=2} \sim_J \texttt{x:=1}\ \texttt{x==2;z:=2}\ \texttt{x==1;z:=1}$$
$$\sim_J \texttt{x==2;z:=2}\ \texttt{x:=1}\ \texttt{x==1;z:=1}$$

But the trace `x==2;z:=2` `x:=1` `x==1;z:=1` violates the postcondition $z = 2$. In other words, a trace proven correct by the proof is equivalent to an incorrect trace, i.e., J is not safe wrt. Π.

The soundness problem can be partially circumvented under some conditions. For instance, if we commit to applying the commutativity relations only to disjoint sets of statements (say, in different parts of the program), a form of sound combination is possible:

Definition 4.3 (Mixed Commutativity Relation). *Let M, N be disjoint sets of statements. The* mixed commutativity relation *$I^\flat\ {}_M{\bowtie}_N\ I^\sharp$ is the union of $I^\flat$ (restricted to M) and $I^\sharp$ (restricted to N).*

$$I^\flat\ {}_M{\bowtie}_N\ I^\sharp := (I^\flat \cap M^2) \cup (I^\sharp \cap N^2)$$

The mixed commutativity relation is safe wrt. the proof Π **(G1)**, but under a stronger premise that implies safety of both $I^\sharp$ and $I^\flat$.

Proposition 4.4 (Safety of Mixed Commutativity). *Let M, N be disjoint sets of statements. Assume that the commutativity relations $I^\sharp$ and $I^\flat$ satisfy the inclusion $cl_{I^\flat}(cl_{I^\sharp}(\Pi)) \subseteq \mathbf{correct}$. Then the mixed commutativity relation $I^\flat \; {}_M{\rhd\!\!\lhd}_N \; I^\sharp$ is safe wrt. the proof Π.*

Proof. If two traces are equivalent up to $I^\flat \; {}_M{\rhd\!\!\lhd}_N \; I^\sharp$, we can (by disjointness of M and N) first make all swaps up to $I^\sharp$, yielding a trace in $cl_{I^\sharp}(\Pi)$. We then make all swaps up to $I^\flat$, which by assumption yields a correct trace. □

We study the additional premise in this result further in Section 5, and connect it to a formal notion of being "more abstract". For now we simply note that it is always fulfilled if $I^\sharp$ is safe wrt. to Π, and $I^\flat = I_C$ is the concrete commutativity.

Note that the restriction to disjoint sets of statements is quite severe: Traces with different interleaving between statements from M and N are never considered equivalent up to mixed commutativity. Consequently, mixed commutativity does not allow us to fully benefit from both commutativity relations **(G2)**. For the case where our commutativity relations are induced by statement abstractions, the restriction to disjoint sets of statements is in fact unnecessary:

Proposition 4.5 (Selective Abstraction). *Let α, β be statement abstractions that preserve the proof Π, and let M, N be disjoint sets of statements. Then the commutativity relation induced by the* selective abstraction *$\alpha \; {}_M{\bowtie}_N \; \beta$ is safe wrt. Π and subsumes the mixed commutativity $I_C^\alpha \; {}_M{\rhd\!\!\lhd}_N \; I_C^\beta$.*

$$I_C^{\alpha \, {}_M{\bowtie}_N \, \beta} \supseteq I_C^\alpha \; {}_M{\rhd\!\!\lhd}_N \; I_C^\beta$$

Here, $(\alpha \; {}_M{\bowtie}_N \; \beta)(\mathit{st}) = \alpha(\mathit{st})$ for $\mathit{st} \in M$, $(\alpha \; {}_M{\bowtie}_N \; \beta)(\mathit{st}) = \beta(\mathit{st})$ for $\mathit{st} \in N$, and $(\alpha \; {}_M{\bowtie}_N \; \beta)(\mathit{st}) = \mathit{st}$ otherwise.

In particular, note that the commutativity relation $I_C^{\alpha \, {}_M{\bowtie}_N \, \beta}$ induced by selective abstraction may allow commutativity of statements in M with statements in N, so the inclusion can be strict. Furthermore, we do not need the additional premise of Proposition 4.4. Hence, any commutativity that can be achieved by mixing commutativity relations induced by statement abstractions can also be achieved with a single statement abstraction.

However, commutativity induced by selective abstraction suffers from some of the same drawbacks as mixed commutativity: Firstly, both combinations rely on the choice of suitable sets of statements M and N. This is challenging particularly in an automated setting, where we can not delegate responsibility for an optimal choice to a user **(G3)**. Secondly, neither combination in general subsumes the commutativity afforded by the two inputs individually **(G2)**, for any choice of M, N. Next, we present an alternative approach for combining commutativity relations that does not suffer from these limitations, and achieves our three goals **(G1)**, **(G2)** and **(G3)**.

5 THE STRATIFIED COMMUTATIVITY PROOF RULE

Section 4 presents several approaches to combine two commutativity relations, and discusses their limitations. A key takeaway is this: Composing two commutativity relations into a single relation requires tradeoffs to ensure safety. Making the best choice for this tradeoff presents an obstacle in automated verification setting. Hence we explore the idea of maintaining two *separate* commutativity relations, and present a new proof rule that incorporates both relations.

We observed before (in Example 4.2) that the union of two safe commutativity relations, say the concrete commutativity relation I_C and a commutativity relation I_C^α induced by a proof-preserving statement abstraction, can be unsafe. The reason for this lies in the fact that taking the union allows us to interleave swaps of statements up to concrete and abstract commutativity arbitrarily. We show that imposing a definite order in which the two commutativity relations are applied leads to a sound proof rule.

Let us first explain the intuition behind this emphasis on the order in which the commutativity relations are applied, before arriving at the formal definition (Definition 5.1). To this end, we examine Example 4.2 more closely. Given a trace proven correct by the proof Π, we first swapped the statements `x==1;z:=1` and `x==2;z:=2`, which commute concretely. However, note that these statements do not commute under projection to the proof (the guards are abstracted away, and the updates to z do not commute). We then subsequently swapped two statements that commute under projection to the proof, but not concretely. Note that the order in which these two swaps were performed was crucial; the statements swapped in the second step are not even adjacent in the first step. By safety of the concrete commutativity I_C, we know that the trace obtained after the first swap, `x:=1` `x==2;z:=2` `x==1;z:=1`, is correct. However, this trace is not proven by Π. Consequently, even though the abstract commutativity $I_C^{\alpha_\Pi}$ induced by projection to the proof is safe wrt. Π, we can not guarantee that the trace obtained after the second swap is still correct – in fact, it is incorrect.

Had we first applied a swap up to the commutativity $I_C^{\alpha_\Pi}$ induced by proof-projection, safety of $I_C^{\alpha_\Pi}$ wrt. Π would have guaranteed that the resulting trace is correct. The concrete commutativity I_C *always* preserves correctness, hence any subsequent swaps up to I_C would still yield correct traces. This suggests that we should obey a certain order in which we swap statements up to the two commutativity relations: Starting from a trace proven by Π, we first swap statements up to $I_C^{\alpha_\Pi}$, and only afterwards we swap statements up to I_C. In the following definition of a *stratified proof*, the idea of applying our commutativity relations in this precise order is our guiding principle. Note that we fix here (and for the remainder of this section) the "more concrete" commutativity relation $I^\flat$ to be precisely the concrete commutativity I_C, while still allowing any safe abstract commutativity $I^\sharp$ (since this combination is our main application). Further below, we discuss in more detail under which conditions two (or more) commutativity relations can be combined in this way.

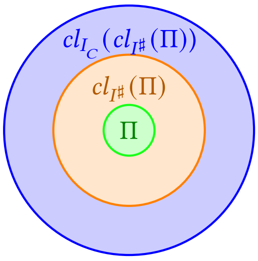


Fig. 3. Commutativity Strata

Definition 5.1 (Stratified Proof). *Let Π be a proof. The corresponding* stratified proof *is the set of traces derived by* (1) *taking the traces proven by* Π, (2) *adding all traces equivalent up to the abstract commutativity* $I^\sharp$*, and* (3) *adding to the set resulting from* (2) *all traces equivalent to the concrete commutativity* I_C*. Formally, we denote this set by* $cl_{I_C}(cl_{I^\sharp}(\Pi))$.

Figure 3 illustrates the strata of commutativity applied around the proof. The following proof rule allows us to conclude correctness from a stratified proof.

Theorem 5.2 (Stratified Proof Rule). *Let Π be a proof. If all program traces of a program P are covered by the stratified proof corresponding to* Π*, then P is correct.*

$$P \subseteq cl_{I_C}(cl_{I^\sharp}(\Pi)) \implies P \text{ is correct}$$

The order in which the two commutativity relations are applied – first abstract commutativity, then concrete commutativity – is absolutely crucial for the soundness of this proof rule. Notably, simply reordering the strata (i.e., inverting the order in which the two closure operations are applied) in Theorem 5.2 results in unsoundness:

Observation 5.3. *There exist a program P and a proof* Π *such that the inclusion* $P \subseteq cl_{I^\sharp}(cl_{I_C}(\Pi))$ *holds, but P is incorrect. Example 4.2 shows such a case.*

The idea is that once we swap any two statements up to the concrete commutativity, we only know that the resulting trace is still correct. However, we lose any connection between the resulting trace and the proof Π: The trace may not be proven correct by Π, and intuitively the implicit abstraction inherent in the proof Π may be too liberal to show correctness of the trace. Formally, safety of $I^\sharp$ wrt. Π only guarantees correctness of traces equivalent (up to $I^\sharp$ alone) to a trace proven by Π. It is precisely this limitation that enables $I^\sharp$ to declare statements as commutative that do not commute concretely; and it is precisely this limitation that requires us to impose an order on the commutativity relations. Fortunately, it turns out that this seeming limitation is still quite permissive and yields a powerful proof rule. In fact, it is straightforward to show (using only the monotonicity and extensivity of closure) that the stratified proof rule generalizes the proof rule based on a single commutativity relation: If the premise of Proposition 3.2 holds for either commutativity relation, then the premise of Theorem 5.2 holds as well.

$$P \subseteq cl_{I_C}(\Pi) \implies P \subseteq cl_{I_C}(cl_{I^\sharp}(\Pi)) \qquad\qquad P \subseteq cl_{I^\sharp}(\Pi) \implies P \subseteq cl_{I_C}(cl_{I^\sharp}(\Pi))$$

We have seen already in the introduction that the generalization can be strict. Furthermore, it follows directly from the proof of Proposition 4.4 that the stratified proof rule subsumes the mixing of commutativity relations with disjoint domains:

$$P \subseteq cl_{I_C\, {}_M\!\!\bowtie_N I^\sharp}(\Pi) \implies P \subseteq cl_{I_C}(cl_{I^\sharp}(\Pi))$$

Similarly, there always exist adversarial choices of M and N such that the *selective abstraction* is less powerful than the stratified proof rule (though in general they are incomparable). The stratified proof rule does not require us to select sets M and N and is thus well suited for automated verification.

Beyond Two Commutativity Relations. Our approach generalizes from two to any number of commutativity relations. This requires a closer look at the conditions under which commutativity relations can be combined in a stratified manner. The following definition formulates a general condition for soundness:

Definition 5.4 (Stratifiability). *Given a proof* Π*, we say that the sequence of commutativity relations* $(I_1, I_2, \dots, I_n)$ *is* stratifiable *if iteratively applying the corresponding closures to the proof* Π *yields a set that contains only correct traces, i.e., if*

$$cl_{I_n}(cl_{I_{n-1}}(\dots cl_{I_1}(\Pi)\dots)) \subseteq \mathbf{correct}.$$

If a sequence of commutativity relations $(I_1, I_2, \dots, I_n)$ is stratifiable, then each commutativity relation I_i in the sequence is safe wrt. the proof Π (this can be shown using the monotonicity of closure). The presentation above is based on a special case of stratifiability:

Observation 5.5. *Given a proof* Π *and an abstract commutativity relation* $I^\sharp$ *that is safe wrt.* Π*, the pair* $(I^\sharp, I_C)$ *formed by the abstract commutativity relation* $I^\sharp$ *and the concrete commutativity relation* I_C *is stratifiable.*

For other stratifiable sequences of (possibly more than two) commutativity relations, we again turn to statement abstractions. In Section 4, given the two commutativity relations $I^\sharp$ and $I^\flat$, we fix the terminology to call $I^\sharp$ "more abstract" and $I^\flat$ "more concrete". In this section, given the sequence of commutativity relations $I_1, \dots, I_n$, we generalize the terminology and call I_i "more abstract" than I_j if $i < j$. The terminology stems from the following statement:

Proposition 5.6 (Stratifiability on Statement Abstractions). *Let* $\alpha_1, \alpha_2, \dots, \alpha_n$ *be a sequence of proof-preserving statement abstractions and let* $I_C^{\alpha_i}$ *be the commutativity relation induced by the statement abstraction* α_i*. If each statement abstraction* α_i *is more abstract than* α_{i+1}*, i.e.,* $[\![\alpha_i(st)]\!] \supseteq [\![\alpha_{i+1}(st)]\!]$ *for all statements* st*, then the sequence of induced commutativity relations* $(I_C^{\alpha_1}, I_C^{\alpha_2}, \dots, I_C^{\alpha_n})$ *is stratifiable.*

Proof. Let $\mathbf{correct}_\alpha = \{\tau \mid \alpha(\tau) \in \mathbf{correct}\}$. From $\Pi \subseteq \mathbf{correct}_{\alpha_1}$, the fact that $\mathbf{correct}_\alpha$ is closed under I_α, and $\mathbf{correct}_{\alpha_i} \subseteq \mathbf{correct}_{\alpha_{i+1}}$, it inductively follows that the iterated closure $cl_{I_{\alpha_n}}(\ldots cl_{I_{\alpha_1}}(\Pi)\ldots)$ is a subset of $\mathbf{correct}_{\alpha_n}$, which in turn is a subset of **correct**. □

As another example (not covered by the above proposition), the sequence $(I_S, I_C^{\alpha_\Pi}, I_C)$ formed by the stutter commutativity relation I_S (see Section 3.3), the commutativity induced by proof-projection $I_C^{\alpha_\Pi}$ (see Section 3.2), and the concrete commutativity relation I_C is stratifiable.

The proof rule based on stratified commutativity, stated in Theorem 5.2 specifically for the stratifiable pair $(I^\sharp, I_C)$, generalizes to every stratifiable sequence of commutativity relations:

Definition 5.7 (n-Stratified Proof). *Let Π be a proof, and let $(I_1, \ldots, I_n)$ be a stratifiable sequence of commutativity relations. The n-stratified proof corresponding to Π is the set of traces $cl_{I_n}(\ldots cl_{I_1}(\Pi)\ldots)$ which is obtained by iteratively applying the corresponding closures to the proof Π.*

Theorem 5.8 (n-Stratified Proof Rule). *Let Π be a proof, and let $(I_1, \ldots, I_n)$ be a stratifiable sequence of commutativity relations. If all program traces of a program P are covered by the n-stratified proof corresponding to Π, then P is correct.*

$$P \subseteq cl_{I_n}(\ldots cl_{I_1}(\Pi)\ldots) \implies P \text{ is correct}$$

6 EFFECTIVE COMMUTATIVITY STRATIFICATION

While the stratified proof rule (Theorem 5.2) and its generalization to n relations (Theorem 5.8) are theoretically powerful, they are not directly amenable to algorithmic verification: A verification algorithm that constructs candidate proofs Π in an abstraction refinement loop must be able to automatically check whether a given proof Π is sufficient to conclude correctness of the analyzed program. We call this step the *proof check*, and it amounts to deciding the premise of a proof rule. However, already the premise of our single-relation proof rule (Proposition 3.2) presents a challenge: Even though the program P and the proof Π can be represented as regular languages, the closure $cl_I(\Pi)$ induced by a commutativity relation I may not be regular. In fact, the premise of Proposition 3.2 is known to be undecidable [Ochmanski 1995]. The premise of our new stratified proof rule Theorem 5.2 is similarly undecidable (for instance, it collapses to the undecidable single-relation case if $I_C \subseteq I^\sharp$).

In the following, we study a strengthening of the stratified commutativity proof rule's premise to a decidable condition. For clarity of presentation, we again discuss this first for the case of some "abstract" commutativity relation $I^\sharp$ (that is safe wrt. the proof Π) and the concrete commutativity I_C. Further below, we generalize to an arbitrary stratifiable sequence of commutativity relations.

6.1 Towards Decidable Proof Checking

It is a well-known result [Ochmanski 1995] that the single-relation case can be made decidable by strengthening the premise of the proof rule (Proposition 3.2) through the introduction of certain *normal forms* for equivalence classes, which are defined via lexicographic orders. The premise of Proposition 3.2 states that for every trace τ of the program P, there exists a trace τ' that is equivalent (up to a commutativity relation I), such that τ' is proven correct by the proof Π.

$$\forall \tau \in P \,.\, \exists \tau' \in \Pi \,.\, \tau' \sim_I \tau$$

Given a lexicographic total order $\preceq$ over traces, this condition can be strengthened by requiring that the trace τ' is less than the trace τ up to this order:

$$\forall \tau \in P \,.\, \exists \tau' \in \Pi \,.\, \tau' \sim_I \tau \land \tau' \preceq \tau \tag{1}$$

It follows that in fact, the $\preceq$-minimal trace in the equivalence class of τ must be proven correct by Π. We call this minimal trace the *representative* for τ. To conclude that condition (1) holds, it then suffices to show that the set of all such representatives for traces of P is subsumed by the proof Π.

We generalize a definition from recent work [Farzan et al. 2022] and define the set of representatives (called a *reduction*) wrt. a single commutativity relation as follows:

Definition 6.1 (Generalized Reduction). *Let I be a commutativity relation, and let L be a set of traces (not necessarily closed under I). The reduction of L wrt. I is defined as the set of minimal representatives present in L, for each equivalence class of a trace in L.*

$$red^{I}_{\preceq}(L) := \{\, \min_{\preceq}([\tau]_I \cap L) \mid \tau \in L \,\}$$

Our generalization allows the reduction operator $red^{I}_{\preceq}(\cdot)$ to be applied to sets of traces L that are not closed under I: We first intersect each equivalence class $[\tau]_I$ with the language L, before taking the minimum of the resulting set. This intersection ensures that $red^{I}_{\preceq}(L)$ is always a subset of L. For closed sets L, our definition exactly coincides with the definition of [Farzan et al. 2022].

The ability to define the reduction of a non-closed language is of relevance below. However, for now let us take the reduction $red^{I}_{\preceq}(P)$ of the set of program traces P. As intuitively argued above, the set $red^{I}_{\preceq}(P)$ is subsumed by Π if and only if condition (1) holds. It is a known result [Ochmanski 1995] that the reduction $red^{I}_{\preceq}(P)$ is a regular language (taking advantage of the fact that the program P is closed under I). Hence condition (1) amounts to an inclusion between regular languages, which can be effectively decided.

We apply the same reasoning to find an analogous, decidable sufficient condition for the stratified proof rule, Theorem 5.2. The premise of the proof rule allows us, for every trace τ of the program P, to first find an I_C-equivalent trace τ'', and then find a trace τ' that is $I^\sharp$-equivalent to τ'', such that τ' is proven correct by the proof Π. In other words, it is equivalent to the following:

$$\forall \tau \in P \,.\, \exists \tau' \in \Pi, \tau'' \in \mathbf{Stmt}^* \,.\, \tau' \sim_{I^\sharp} \tau'' \sim_{I_C} \tau$$

We strengthen this condition by ordering the traces τ, τ'' and τ' with a total lexicographic order:

$$\forall \tau \in P \,.\, \exists \tau' \in \Pi, \tau'' \in \mathbf{Stmt}^* \,.\, \tau' \sim_{I^\sharp} \tau'' \sim_{I_C} \tau \wedge \tau' \preceq \tau'' \preceq \tau \tag{2}$$

Finally, in order to find an algorithm to decide condition (2), we again frame the problem as a language inclusion, using the reduction operator defined above.

Proposition 6.2. *Condition (2) holds if and only if $red^{I^\sharp}_{\preceq}(red^{I_C}_{\preceq}(P) \cup \Pi) \subseteq \Pi$ holds.*

We arrive at a new proof rule. Below, in Section 6.2 and Section 6.3, we present an algorithm that effectively decides the premise of this proof rule.

Theorem 6.3 (Decidable Proof Rule). *Let Π be a proof, and let $\preceq$ be a lexicographic order on traces. If the inclusion below holds, then the program P is correct.*

$$red^{I^\sharp}_{\preceq}(red^{I_C}_{\preceq}(P) \cup \Pi) \subseteq \Pi \tag{3}$$

Proof. Suppose the inclusion (3) holds. From Proposition 6.2 and the reasoning above, it follows that $P \subseteq cl_{I_C}(cl_{I^\sharp}(\Pi))$ holds. The correctness of P then follows by Theorem 5.2. $\square$

Let us examine the premise of Theorem 6.3 more closely. We first take the reduction of the program up to the concrete commutativity relation I_C. Then we take the union with the proof Π, and finally we take the reduction of the resulting set of traces up to the abstract commutativity relation $I^\sharp$. The reduction operators up to the two commutativity relations are here applied to the program P in exactly the reverse order to the application of the corresponding closure operators to Π in Theorem 5.2.

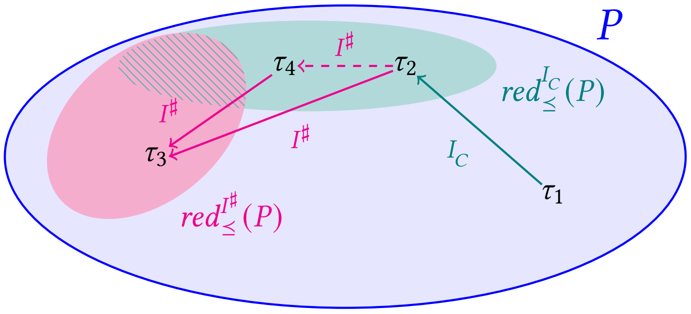


Fig. 4. Reduction up to two commutativity relations $I^\sharp$ and I_C

The addition of the traces proved correct by the proof Π before applying the second reduction is perhaps counterintuitive. To understand this, examine the illustration in Figure 4: Given a trace τ_1 in the program, the first reduction (shown in green) contains an I_C-equivalent representative τ_2 for the trace τ_1. Next, we apply the second reduction up to the abstract commutativity $I^\sharp$. However, the input language for this reduction operator is not closed under $I^\sharp$, hence the minimal trace τ_3 in the $I^\sharp$-equivalence class of τ_2 may not be available as representative for τ_2. If we did not add the traces proved correct by Π, Definition 6.1 would choose the minimal *available* trace τ_4 as the representative for τ_2. However, if τ_3 is already proven correct, adding it back before we apply the second reduction allows us to choose τ_3 as the representative for both τ_2 and τ_4. The following proposition states that indeed, adding back the traces of Π yields a weaker criterion than merely applying both reductions: I.e., the proof check based on Theorem 6.3 succeeds more often than a proof check based on the inclusion $red_{\preceq}^{I^\sharp}(red_{\preceq}^{I_C}(P)) \subseteq \Pi$. Furthermore, Theorem 6.3 is still a more general proof rule than the decidable variants of the single-relation proof rule.

Proposition 6.4. *Each of the inclusions below defines a stronger condition than the inclusion* (3).

$$red_{\preceq}^{I^\sharp}(red_{\preceq}^{I_C}(P)) \subseteq \Pi \qquad red_{\preceq}^{I^\sharp}(P) \subseteq \Pi \qquad red_{\preceq}^{I_C}(P) \subseteq \Pi$$

Let us reflect on the role of the proof Π in our proof rule. Suppose for the moment that we instantiate the abstract commutativity $I^\sharp$ with the commutativity induced by projection to the proof α_Π (Section 3.2). Now, the proof Π appears three times in the premise of our proof rule:

$$red_{\preceq}^{I_C^{\alpha_\Pi}}(red_{\preceq}^{I_C}(P) \cup \Pi) \subseteq \Pi$$

Our verification hence amounts to searching a proof Π that is a post-fixpoint of the function

$$\lambda X \, . \, red_{\preceq}^{I_C^{\alpha_X}}(red_{\preceq}^{I_C}(P) \cup X)$$

Even if instead of the commutativity $I_C^{\alpha_\Pi}$ induced by projection to the proof α_Π, one were to use some other abstract commutativity relation $I^\sharp$, this relation $I^\sharp$ would still have to (implicitly) depend on the proof Π, in order to ensure that $I^\sharp$ is safe wrt. the proof Π. Consequently, and in contrast to previous proof rules, a "good proof" (i.e., a proof for which the verification succeeds) must not only cover a large set of traces, it must also allow powerful abstract commutativity. What characterizes such a proof is an open question, and likely highly dependent on the strategy to derive commutativity from the proof.

Before we move on to present an algorithm capable of deciding the premise of Theorem 6.3, let us complete this discussion by again considering the generalization to any stratifiable sequence of commutativity relation $(I_1, \ldots, I_n)$. Again, the results for two relations generalize directly:

THEOREM 6.5 (DECIDABLE n-STRATIFIED PROOF RULE). *Let Π be a proof, and let $\preceq$ be a lexicographic order on traces. If the inclusion below holds, then the program P is correct.*

$$red_{\preceq}^{I_1}(\dots red_{\preceq}^{I_{n-1}}(red_{\preceq}^{I_n}(P) \cup \Pi) \dots \cup \Pi) \subseteq \Pi \tag{4}$$

Analogous to Proposition 6.4, the above proof rules generalizes the corresponding rule for any of the commutativity relations in isolation:

PROPOSITION 6.6. *Each inclusion $red_{\preceq}^{I_i}(P) \subseteq \Pi$ defines a stronger condition than inclusion (4).*

Accordingly, the proof rules based on the respective single reductions are weaker than the proof rule based on stratified reduction. In fact, we have an infinite hierarchy of stratified reductions: $(n + 1)$-stratified reductions can be more powerful than n-stratified reductions, for any n.

6.2 A Schematic Construction of Stratified Reductions

In the previous section, we have introduced a new proof rule (Theorem 6.5, and the special case of Theorem 6.3) for concurrent programs based on the idea of *strata* of commutativity. In this section, we show that the premise of this new proof rule is decidable, i.e., we present an algorithm that decides the inclusion (4). Our algorithm takes the following parameters as inputs:

(1) *the program P:* As explained in Section 2, we think of the program P as a deterministic finite automaton whose alphabet is made up of program statements.
(2) *a proof Π:* As defined in Section 2, a proof is a set of Hoare triples. In this section, we identify the proof Π with a finite automaton whose states are given by assertions, with the precondition φ_{pre} the initial and the postcondition φ_{post} the (only) accepting state, and each transition from a state φ to a state ψ labeled by a statement $\mathfrak{st}$ corresponds to a Hoare triple $\{\varphi\}\ \mathfrak{st}\ \{\psi\}$ in Π. We assume wlog. that the automaton Π is deterministic and total.
(3) *a lexicographic order $\preceq$ on traces:* The notion of the order can be generalized from "normal" lexicographic orders to the larger class of *positional lexicographic orders* [Farzan et al. 2022]: We assume that $\preceq$ is given by an underlying total order on statements $<_q$ for each location q of the program P.
(4) *a stratifiable sequence of commutativity relations $(I_1, \dots, I_n)$:* We assume that the program P, viewed as a set of program traces, is closed under all these commutativity relations. This assumption is justified, as we can always restrict the relations such that this is the case. Intuitively, this amounts to only considering commutativity between statements of different threads.

Given these parameters, the algorithm to decide the inclusion (4) constructs a finite automaton $\mathfrak{R}_{opt}$. The language recognized by this automaton $\mathfrak{R}_{opt}$ is *equivalent* to the language

$$L := red_{\preceq}^{I_1}(\dots red_{\preceq}^{I_{n-1}}(red_{\preceq}^{I_n}(P) \cup \Pi) \dots \cup \Pi) \tag{5}$$

modulo the set Π of traces covered by the proof: $\mathfrak{R}_{opt}$ is a subset of Π if and only if L is a subset of Π (i.e., inclusion (4) holds). Thereby we decide the inclusion (4) by deciding the inclusion $\mathfrak{R}_{opt} \subseteq \Pi$, an inclusion between deterministic finite automata.

There is an inherent nondeterminism in the idea of stratified commutativity: To get from a trace τ of the program P to its representative trace τ' in the proof, we must first swap statements up to concrete commutativity, and then at some point nondeterministically decide to now swap statements up to abstract commutativity instead – or, in the general case, decide to switch from swaps according to I_i to swaps according to I_{i+1}. In this section, we present a schematic construction of a reduction automaton $\mathfrak{R}_f$ that is parametrized in a strategy f (called a *budget function*) that resolves this nondeterminism. Identifying the budget function as a parameter allows us to compare different strategies, and compare the quality of the resulting reductions. Each strategy corresponds to a sufficient condition for the inclusion (4). Section 6.3 then presents a deterministic strategy (a budget function *opt*) that *optimally* resolves the nondeterminism, and furthermore allows us to *precisely* decide the inclusion (4), rather than some strictly stronger condition.

The construction of stratified reductions takes inspiration from the work on *sleep sets* [Godefroid 1996] in the *partial order reduction* literature. The constructed automaton assigns to each state a set of letters (the *sleep set*). Intuitively, outgoing transitions labeled by a letter in a state's sleep set can be safely pruned. Our construction additionally stores for each letter in the sleep set which commutativity relations justify the letter's presence in the sleep set: In case we have only the two relations $I^\sharp$ and I_C, this is either just the abstract commutativity relation $I^\sharp$ in isolation, or the abstract and concrete commutativity relations ($I^\sharp$ and I_C) in combination. In the general setting of a stratifiable sequence $(I_1, \ldots, I_n)$, a letter's presence in the sleep set can be justified with any prefix $(I_1, \ldots, I_k)$, for $k \leq n$, of the sequence. Hence, our construction essentially replaces sleep sets by *sleep maps*, partial functions from letters into the set $\{1, \ldots, n\}$. If a statement a is in the sleep map m, we call $m(a)$ the *price* of a.

Storing the price of each letter in the sleep map is crucial to avoid arbitrary interleavings of swaps up to the different commutativity relations and thus precisely implement the idea of stratified commutativity. Once we use a swap based on the commutativity relation I_{i+1} to justify pruning a transition, we must not subsequently justify pruning a transition (at a successor state) labeled by the same letter based on a swap allowed only by the commutativity I_i.

Our construction resolves the nondeterminism inherent in the idea of stratified commutativity through a concept called *budget functions*. Intuitively, in cases where a transition can be pruned based on the "more concrete" commutativity relation I_{i+1}, a budget function decides if the transition should be pruned (limiting future applications of the "more abstract" commutativity relation I_i) or if the transition should be kept. This connects with the discussion following Theorem 6.3: "Adding back" traces to the inner (concrete) reduction (or here, not removing them in the first place), allows more freedom in choosing the representatives for the outer (abstract) reduction. Formally, budget functions are defined as follows:

Definition 6.7 (Budget Function). *Let the program be given as DFA $P = (Q, \Sigma, \delta, q_{\text{init}}, F)$. A* budget function f *for P takes a location $q \in Q$ of the program P, a sleep map $m : \Sigma \rightharpoonup \{1, \ldots, n\}$, an upper bound $k \in \{1, \ldots, n\}$ and a letter $a \in \Sigma$, and returns a "budget" $f(q, m, k, a) \in \{1, \ldots, n\}$ such that $f(q, m, k, a) \leq k$.*

The idea behind a budget function is this: If the price $m(a)$ for a letter a in the current sleep map m is less or equal to the budget $f(q, m, k, a)$ for the current state q and upper bound k, we may prune transitions labeled with this letter a. If the price $m(a)$ exceeds the budget (or is undefined), transitions labeled a must not be pruned. Formally, the construction is defined as follows:

Definition 6.8 (Stratified Reduction). *Let the program be given as DFA $P = (Q, \Sigma, \delta, q_{\text{init}}, F)$ with $\Sigma \subseteq$* **Stmt***, and let f be a budget function. The* reduction automaton *for f is defined as the DFA*

$$\mathfrak{R}_f := (\hat{Q}, \Sigma, \hat{\delta}_f, \hat{q}_{\text{init}}, \hat{F})$$

where

- *each state $\langle q, m, k \rangle$ consists of a location q, a sleep map m, and a budget k,*
$$\hat{Q} = Q \times (\Sigma \rightharpoonup \{1, \ldots, n\}) \times \{1, \ldots, n\}$$
- *the alphabet Σ is the set of statements occurring in the program P,*
- *the transition function $\hat{\delta}_f$ is defined below,*

- *the initial state consists of the initial location, the empty sleep map, and the maximum budget,*

$$\hat{q}_{\text{init}} = \langle q_{\text{init}}, \emptyset, n \rangle$$

- *and a state $\langle q, m, k \rangle$ is accepting if q is accepting.*

$$\hat{F} = F \times (\Sigma \rightharpoonup \{1, \ldots, n\}) \times \{1, \ldots, n\}$$

The transition function is given as follows:

$$\hat{\delta}_f(\langle q, m, k \rangle, a) = \begin{cases} \textit{undefined} & \textbf{\textit{if}}\ \delta(q,a)\ \textit{undefined, or}\ m(a)\ \textit{defined and}\ m(a) \le k' \\ \langle q', m', k' \rangle & \textbf{\textit{else}} \end{cases}$$

where $q' = \delta(q, a)$, $k' = f(q, m, k, a)$ and $m' = \mathit{constrain}_{k'}(\mathit{transfer}_a(\mathit{cost}_f))$ with

$$\mathit{cost}_f(b) := \begin{cases} \min\big(m(b), f(q, m, k, b)\big) & \textbf{\textit{if}}\ b <_q a\ \textit{and}\ m(b)\ \textit{defined} \\ f(q, m, k, b) & \textbf{\textit{else if}}\ b <_q a \\ m(b) & \textbf{\textit{else if}}\ m(b)\ \textit{defined} \\ \textit{undefined} & \textbf{\textit{else}} \end{cases}$$

$$\mathit{transfer}_a(\hat{m})(b) := \begin{cases} \min\{i \mid \hat{m}(b) \le i \wedge (b, a) \in I_i\} & \textbf{\textit{if}}\ \hat{m}(b)\ \textit{defined} \\ \textit{undefined} & \textbf{\textit{else}} \end{cases}$$

$$\mathit{constrain}_{k'}(\hat{m})(b) := \begin{cases} \hat{m}(b) & \textbf{\textit{if}}\ \hat{m}(b)\ \textit{defined and}\ \hat{m}(b) \le k' \\ \textit{undefined} & \textbf{\textit{else}} \end{cases}$$

A state $\langle q, m, k \rangle$ of the reduction automaton stores the sleep map m as well as the *budget* k. This budget is determined by the budget function f (see the definition of the transition function $\hat{\delta}_f$), and represents the maximum price that can be "paid" (the maximum commutativity relation that can be used to prune transitions) in the state and all (transitively) reachable successor states.

In case a transition is not pruned, the computation of the new sleep map proceeds in multiple steps: First, the sleep map cost_f contains statements that are either already in the sleep map m, or are ordered before the current statement a. This is analogous to classical sleep sets. The price of a statement b in cost_f is determined by the minimum of the price in m and the budget assigned to b. This means that if the budget is lower than the price in m (and thus, the transition labeled b was not pruned), the price for b decreases: This has precisely the effect that we can once again prune transitions labeled b according to the "more abstract" relation, because we "added back" traces beginning with b. This is illustrated in Figure 5.

In the next step, i.e., for the sleep map $\mathit{transfer}_a(\mathit{cost}_f)$, we must possibly increase the price of each letter in the sleep map to a level i such that a and b commute under relation I_i. Implicitly we are here reasoning about swapping statements a and b in a trace. In the classical (single-relation) case, this corresponds simply to checking if a and b commute under the given relation. In our construction, the cases where the price strictly increases correspond precisely to the points where the construction switches from "more abstract" commutativity to "more concrete" commutativity.

Finally, through $\mathit{constrain}_{k'}$, we evict all statements from the sleep map whose price exceeds our assigned budget k'. This ensures that we indeed obey the upper limit imposed by the budget, i.e., if the budget is k', we do not use the more concrete commutativity relations I_k with $k > k'$ to prune transitions from the successor state.

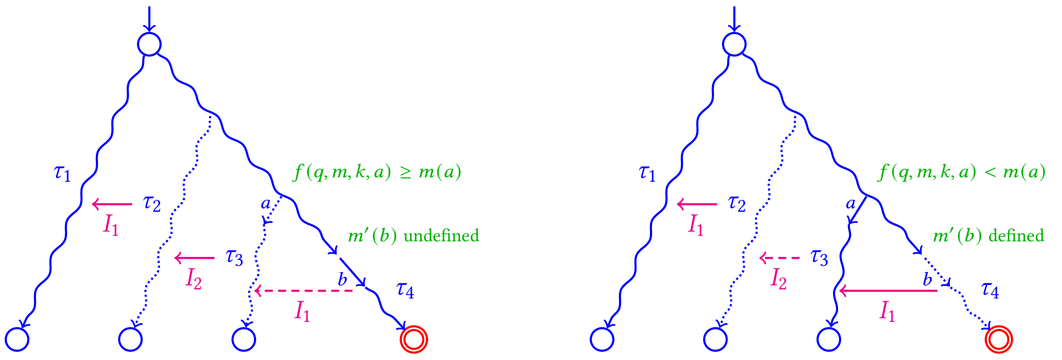


Fig. 5. The reduction automaton on the left accepts the traces $\{\tau_1, \tau_4\}$. The rejected trace τ_2 is I_1-equivalent to τ_1, and the rejected trace τ_3 is I_2-equivalent to τ_2. Though τ_4 is I_1-equivalent to τ_3, it must be included in the reduction because we already used I_2. But τ_4 is not proven by Π (indicated by the red state), so the reduction is not covered by the proof. On the right, an alternative budget function prevents us from pruning the transition labeled a, so τ_3 is included in the reduction. In turn, we can later prune the transition labeled b (because b is in the sleep map), and the resulting reduction $\{\tau_1, \tau_3\}$ is covered by the proof.

All instances of this schematic construction (for different choices of f) correspond to some sufficient condition for inclusion (4):

Theorem 6.9 (Reduction). *For all budget functions f, if the language recognized by the reduction automaton $\mathfrak{R}_f$ is covered by the proof Π (i.e., $\mathfrak{R}_f \subseteq \Pi$), then inclusion (4) holds.*

Hence, deciding the automata inclusion $\mathfrak{R}_f \subseteq \Pi$ allows us to soundly conclude program correctness (using Theorem 6.5). However, depending on the budget function f, the inclusion $\mathfrak{R}_f \subseteq \Pi$ might represent an unnecessarily strong sufficient condition. For instance, if f returns 1 for all inputs, then the computed reduction is exactly the reduction up to I_1, i.e., $\mathfrak{R}_f = red_{\preceq}^{I_1}(P)$. In this case, we do not truly benefit from applying stratified commutativity. There is a wide spectrum of budget functions that may lead to less or more useful reductions. In the next section, we propose a particular budget function that allows us to benefit from each commutativity relation.

6.3 Resolving Nondeterminism with Optimism

Our proposed budget function is based on two observations:

(1) Assigning a higher budget for a letter allows us to prune more transitions in the successor state (and all transitively reachable states). This can be crucial to exclude an unproven trace from the reduction.
(2) Assigning a lower budget (preventing usage of "more concrete" commutativity) for a letter a yields a more powerful sleep map for other successor states (i.e., a sleep map more likely to contain a at a lower price). This allows us to exclude more traces starting from such successor states.

Hence, the best budget function would be one that assigns as low a budget as possible, as long as it excludes all unproven traces from the reduction. In order to define this budget function precisely, we modify the input given to the automaton construction: Rather than applying it directly to the program automaton P, we first construct a product automaton $P \otimes \Pi$ of the program and the proof. Here, the proof acts as a kind of monitor: It never blocks (its transition function is total), and acceptance in the product is purely determined by the location of P. Thus, the product automaton recognizes precisely the language of P; only the structure is modified. States of this automaton now have the form $\langle q, \varphi\rangle$ for a location q of P and an assertion φ from Π. We define the budget function:

Definition 6.10 (Optimistic Budget). *The* optimistic budget function *opt for $P \otimes \Pi$ assigns the lowest budget such that the successor state can not reach an accepting location of P without also reaching the assertion $\perp$ in Π. If no such budget exists, we default to the highest possible budget. Formally, let*

$$M = \left\{ i \mid i < k \wedge \neg\exists q_f \in F, \varphi \neq \perp, m', k' . \langle \delta_{P\otimes\Pi}(\langle q, \varphi\rangle, a), m_i, i\rangle \rightarrow^*_{\mathfrak{R}_{opt}} \langle\langle q_f, \varphi\rangle, m', k'\rangle \right\}$$

where $m_i = constrain_i(transfer_a(cost_{opt}))$. If $M \neq \emptyset$, we define $opt(\langle q, \varphi\rangle, m, k, a) := \min(M)$, otherwise $opt(\langle q, \varphi\rangle, m, k, a) := k$.

The definition of the optimistic budget is recursive: *opt* appears in the definition of m_i and in the reachability relation $\rightarrow^*_{\mathfrak{R}_{opt}}$. In the definition of m_i, *opt* will only be applied to letters $b < a$. And the reachability relation $\rightarrow^*_{\mathfrak{R}_{opt}}$ is here only applied to states with a budget strictly lower than k. Hence the function is well-defined. Also note that in cases where $m(a)$ is defined, we can further restrict $i < m(a)$ without changing the reduction automaton, as the transition function $\hat{\delta}_{opt}$ (specifically $cost_{opt}$) will ignore higher values anyway.

The optimistic budget function can be implemented by always first ("optimistically") choosing the lowest budget 1. We then recursively check if the successor state under this budget finds an unproven trace. If not, then we stick to the chosen budget. If on the other hand, we find an unproven trace, and the chosen budget was strictly less than the state's maximum budget, we increment the budget, and check again if it is now sufficient. For efficiency, our implementation avoids repeatedly exploring the same parts of the automaton.

We show that among all budget functions, the optimistic budget *opt* is optimal, in the sense that it gives us the weakest possible inclusion:

Proposition 6.11 (Optimality). *Let f be a budget function. If the inclusion $\mathfrak{R}_f \subseteq \Pi$ holds, then the inclusion $\mathfrak{R}_{opt} \subseteq \Pi$ also holds.*

Moreover, the automaton $\mathfrak{R}_{opt}$ for the optimistic budget function *opt* is equivalent modulo Π to the language $L := red^{I_1}_{\preceq}(\ldots red^{I_{n-1}}_{\preceq}(red^{I_n}_{\preceq}(P) \cup \Pi) \ldots \cup \Pi)$, allowing us to decide inclusion (4).

Theorem 6.12 (Correctness). *The automaton $\mathfrak{R}_{opt}$ accepts a subset of the proven traces, i.e., $\mathfrak{R}_{opt} \subseteq \Pi$ holds, if and only if inclusion (4) holds.*

Proof. We prove the inclusions $\mathfrak{R}_{opt} \subseteq L \cup \Pi$ and $L \subseteq \mathfrak{R}_{opt} \cup \Pi$. The result then follows. For details, we refer to the extended version of our paper [Farzan et al. 2023]. □

7 EXPERIMENTAL RESULTS

We implemented a prototype of the verification approach based on abstract commutativity and stratified reduction in order to demonstrate its practical applicability. To this end, we based our implementation on the open-source software model checker Ultimate GemCutter [Klumpp et al. 2022], which verifies programs using an approach based on reduction and concrete commutativity [Farzan et al. 2022]. Our implementation augments GemCutter with support for abstract commutativity and stratified reductions. Specifically, we implemented abstract commutativity based on projection to the proof (Section 3.2). We evaluate the implementation empirically on a standard set of benchmarks. In our evaluation, we are interested in answering the following questions:

Q1 *What is the impact of using abstract commutativity as compared to concrete commutativity?* Does abstract commutativity allow us to successfully analyse more programs? Does it simplify proofs or decrease verification time? And in which scenarios does it perform best?

Q2 *Is the stratified commutativity approach practically feasible?* Can the overhead involved in computing stratified reductions be made manageable?

Q3 *What advantage can be gained from stratified commutativity?* Compared to applying concrete or abstract commutativity in isolation, is there further potential for proof simplification?

To investigate these questions, we analysed benchmarks from three benchmark sets:

(1) First, we verified the 732 programs in the *ConcurrencySafety* category of the Software Verification Competition (SV-COMP'22) [Beyer 2022]. Each of these programs is written in C and comes with a reachability specification.

(2) Second, we verified the same programs against a more light-weight property, namely *memory safety*: The program only dereferences valid pointers to allocated memory, and does not invoke `free()` on an invalid pointer. Violations of this property would constitute undefined behaviour in C.

(3) Third, we verified 50 custom benchmark programs. Some of these programs showcase the limitations of concrete commutativity, others are challenging for abstract commutativity.

The evaluation was performed using the BenchExec benchmarking tool [Beyer et al. 2019] on a Debian 10.10 machine with a AMD Ryzen Threadripper 3970X 32-Core Processor. Each verification run was given a timeout of 15 min and a memory limit of 8 GB.

Discussion: Checking Abstract Commutativity. Our experiments showed a severe practical limitation for the commutativity induced by projection to the proof: If SMT solvers are used to check commutativity of the abstracted statements, timeouts frequently occur due to the quantification introduced by the projection to the proof. In many cases, unsuccessful commutativity checks consume a large part of the given time. To make abstract commutativity as given by projection to the proof practical, we instead check a sufficient condition. Recall from Section 3 that a sufficient condition for two statements to commute concretely is that neither statement writes to a variable read or written by the other statement. This criterion can be relaxed further to account for nondeterministic writes: We allow both statements to write to some variable, as long as both statements *nondeterministically assign* the variable (without any restriction of the new value). Let $\mathit{read}(st)$ denote the set of program variables read by a statement st, $\mathit{write}(\mathit{st})$ the set of variables possibly modified by st, and $\mathit{havoc}(\mathit{st})$ the set of variables nondeterministically assigned (without restrictions) by st. The following condition implies that st_1 and st_2 commute concretely:

$$\mathit{read}(\mathit{st}_1) \cap \mathit{write}(\mathit{st}_2) = \mathit{write}(\mathit{st}_1) \cap \mathit{read}(\mathit{st}_2) = \emptyset \wedge \mathit{write}(\mathit{st}_1) \cap \mathit{write}(\mathit{st}_2) \subseteq \mathit{havoc}(\mathit{st}_1) \cap \mathit{havoc}(\mathit{st}_2)$$

Applying this criterion to the abstractions $\alpha_\Pi(\mathit{st}_1)$ and $\alpha_\Pi(\mathit{st}_2)$ given by projection to a proof Π (in place of st_1 and st_2) yields the following efficiently checkable sufficient condition for commutativity under projection to the proof:

$$\begin{aligned}\mathit{read}(\alpha_\Pi(\mathit{st}_1)) \cap \mathit{write}(\alpha_\Pi(\mathit{st}_2)) &= \mathit{write}(\alpha_\Pi(\mathit{st}_1)) \cap \mathit{read}(\alpha_\Pi(\mathit{st}_2)) = \emptyset \\ &\wedge \mathit{write}(\alpha_\Pi(\mathit{st}_1)) \cap \mathit{write}(\alpha_\Pi(\mathit{st}_2)) \subseteq \mathit{havoc}(\alpha_\Pi(\mathit{st}_1)) \cap \mathit{havoc}(\alpha_\Pi(\mathit{st}_2))\end{aligned}$$

This is not a departure from the concept of abstract commutativity: It merely represents one possible way (an alternative to using SMT solvers) to effectively check commutativity of the abstracted statements, a problem which – for complex statements involving arrays and nonlinear arithmetic – is in any case undecidable in general. Soundness of our implementation follows from the fact that commutativity under projection to a proof Π is safe wrt. Π (which implies safety of any subrelation) and the theoretical developments presented in this paper.

Table 1. Number (#) of successful benchmarks, CPU time, memory (mem), and number of refinement rounds.

	Concrete				Abstract				Stratified			
	#	time (s)	mem (GB)	rounds	#	time (s)	mem (GB)	rounds	#	time (s)	mem (GB)	rounds
SV-COMP Benchmarks: Original Specification												
successful	432	13 465	493	3 937	429	12 806	576	4 036	434	14 595	697	4 027
- correct	145	5 081	131	1 419	142	4 216	150	1 397	147	5 977	178	1 469
- incorrect	287	8 384	362	2 518	287	8 545	426	2 639	287	8 617	519	2 558
SV-COMP Benchmarks: Memory Safety												
successful	486	19 589	443	6 107	504	17 056	514	8 156	505	17 590	578	8 251
- correct	485	19 523	442	6 025	503	16 970	512	8 074	504	17 527	577	8 169
- incorrect	1	65	1	82	1	85	1	82	1	64	1	82
Custom Benchmarks												
successful	24	1 197	30	229	18	861	22	131	30	1 426	48	283
- correct	23	1 192	30	228	17	854	22	130	29	1 421	48	282
- incorrect	1	5	<1	1	1	7	<1	1	1	5	<1	1

Results. Table 1 shows evaluation data for the verification using concrete commutativity in isolation (the classical GemCutter setup), with abstract commutativity in isolation, and with stratified commutativity. We observe that abstract commutativity is particularly powerful when we check the lightweight memory safety property: It allows us to successfully verify 18 additional programs, compared to concrete commutativity. This corresponds to our intuition (explained in the introduction) that such properties allow for a lot of abstract commutativity. For the more complex reachability properties specified in the SV-COMP benchmarks, concrete commutativity is superior, though only slightly. The similar results are likely due to the fact that the proof involves most program variables and does not permit for much abstraction (at least, through projection to the proof). The slight disadvantage for abstract commutativity is to be expected, considering that our implementation of abstract commutativity is limited to the syntactic criterion above. For the custom benchmarks, stratified commutativity is able to verify the largest number of programs, fully benefiting from the need for both abstract and concrete commutativity. For both SV-COMP benchmark sets, stratified commutativity successfully analyzes approximately as many benchmarks as the better of the two commutativity relations. Over all three benchmarks sets, the verification using stratified commutativity successfully analyzes the largest number of programs.

Implementation of Stratified Commutativity. Despite its theoretical advantage, the optimal algorithm for stratified commutativity has significant overhead in in time and space required to compute a stratified reduction. However, in Section 6, we defined a *class* of algorithms for stratified reductions. While the instance based on the optimistic budget function (Section 6.3) is theoretically optimal in terms of the reduction, another algorithm that strikes a tradeoff between the quality of the reduction and the time required for the computation may fare better in practice. Particularly, we ran three instances of the schematic algorithm: the optimal instance in Section 6.3, as well as two randomized variants of the reduction algorithm that, if the optimistic budget is 2, randomly decide (with a bias of 10 % resp. 90 %) whether they assign a budget of 1 or 2. The data presented in Table 1 refers to a portfolio aggregation of these three algorithms, and differs significantly from the data for the individual algorithms, as shown in Table 2.

Over all benchmarks, the optimal algorithm was the fastest of the three stratified reductions in 310 cases (including 18 unique benchmarks only it could solve within the time limit), the algorithm that is biased towards abstract commutativity ("bias 10 %") was the fastest in 704 cases (including 7 unique benchmarks), and the algorithm that is biased towards concrete commutativity ("bias 90 %") was the fastest in 407 cases (including 5 unique benchmarks). This opens up an avenue for future work on improving the practical benefit of stratified reductions: Further investigation of suitable

Table 2. Number (#) of successful benchmarks, CPU time, memory (mem), and number of refinement rounds for 3 different budget functions.

	Optimal				Bias 10 %				Bias 90 %			
	#	time (s)	mem (GB)	rounds	#	time (s)	mem (GB)	rounds	#	time (s)	mem (GB)	rounds
SV-COMP Benchmarks: Original Specification												
successful	426	15 900	785	3 821	426	13 230	648	3 958	420	14 471	743	3 703
- correct	144	5 324	194	1 366	139	4 123	142	1 319	139	4 501	162	1 332
- incorrect	282	10 576	591	2 455	287	9 107	506	2 639	281	9 970	581	2 371
SV-COMP Benchmarks: Memory Safety												
successful	495	20 016	544	7 341	498	15 585	528	7 437	498	17 506	544	7 812
- correct	494	19 928	543	7 259	497	15 522	527	7 355	497	17 442	543	7 730
- incorrect	1	88	1	82	1	64	1	82	1	64	1	82
Custom Benchmarks												
successful	26	1 304	37	249	19	1 120	35	145	19	1 195	35	157
- correct	25	1 298	37	248	18	1 113	34	144	18	1 190	34	156
- incorrect	1	6	<1	1	1	7	<1	1	1	5	<1	1

budget functions, or adaptation of the budget function across refinement rounds, could strike a compromise between theoretical optimality and practical efficiency.

8 RELATED WORK

There has been a variety of previous work on using commutativity for the analysis of concurrent programs. Many approaches in this field do not treat commutativity relations as first-class objects, but rather fix one underlying definition of commutativity (or "independence") for the approach [Abdulla et al. 2014; Flanagan and Godefroid 2005; Kahlon et al. 2009]. While [Godefroid 1996] considers the possibility of different commutativity relations, they impose a soundness notion ("valid dependency relation") that effectively limits these relations to capture the spirit of what we call concrete commutativity. Different commutativity relations then correspond to efficiently checkable sufficient conditions for concrete commutativity. Much of this work is concerned with finite-state systems or executions of bounded length, whereas our work focuses on the proof of correctness for infinite-state programs.

Commutativity reasoning has been integrated in abstraction-refinement based verification of concurrent programs before [Cassez and Ziegler 2015; Chu and Jaffar 2014; Farzan et al. 2022; Farzan and Vandikas 2019, 2020; Wachter et al. 2013]. In all these works, commutativity is mostly equated with concrete commutativity (or with efficiently checkable sufficient conditions).

[Wachter et al. 2013], [Farzan and Vandikas 2020] and [Farzan et al. 2022] consider forms of *conditional* commutativity, where additional knowledge about the program variables' current values allows more statements to commute than according to our definition of concrete commutativity. [Wachter et al. 2013] derive this additional information from a separate program analysis, whereas [Farzan and Vandikas 2020] and [Farzan et al. 2022] take the information from the proof constructed by the verification itself ("proof-sensitive commutativity"). In some cases the effects can be similar, however the theories underlying conditional commutativity in this style and our notion of *abstract* commutativity are quite different: While abstract commutativity is based on allowing *more* behaviour (while preserving safety wrt. the proof), conditional commutativity instead *restricts* possible behaviours (to behaviours that can actually appear in the given context). In the case of [Farzan et al. 2022], the fact that information is taken from the proof is rather incidental (any other reliable source, such as a separate program analysis, would do), whereas the soundness of our abstract commutativity is necessarily deeply tied to the proof constructed by the very same verification in which the commutativity is applied. Conditional commutativity and abstract commutativity are two orthogonal techniques, and in fact both can be applied in combination.

[Chu and Jaffar 2014] define "property-directed commutativity", a notion of commutativity specific to the given program and property. This is similar in spirit to our idea of defining a commutativity relation specific to a given proof (since the proof is obviously specific to given program and property). However, their notion of commutativity is fixed *a priori*, i.e., before the execution of the algorithm. More specifically, they first fix a set of programs (through patterns that limit the operations that can be performed on certain variables) and a set of properties (e.g. properties stating that a variable is bounded by a constant). They then define a commutativity relation that is sound for the fixed set of programs and properties. By contrast, the verification algorithm in our approach computes abstract commutativity relations from each new proof during the iteration of the abstraction refinement loop: This means that the verification algorithm continuously adjusts the notion of commutativity as it progresses (extracting more and more information needed for the proof of the property).

[Elmas et al. 2009] present a verification calculus in which abstraction and commutativity-based reduction are the two key rules. Essentially, the given program is alternatingly reduced and further abstracted. Each abstraction step may allow more statements to commute, which enables further reduction. [Kragl and Qadeer 2018] extend this work by providing a notation to describe the sequence of intermediate programs in a single *layered* program. This line of research can be distinguished from our work as follows: *(1)* Abstractions are not derived by an algorithm but they are chosen by the user, *(2)* it is left up to the responsibility of the user that the abstractions are precise enough so that the abstracted program does not violate the verified property, and *(3)* it is left up to the ingenuity of the user to select abstractions that are helpful to commutativity. These are precisely the three aspects that must be addressed in the setting of verification algorithm where abstractions must be computed automatically, and that our approach captures through: *(1)* projection to the proof or proof-stuttering commutativity, *(2)* proof-preserving abstractions and safety of a commutativity relation wrt. a given proof, and *(3)* stratified commutativity.

[Elmas et al. 2009] and [Kragl and Qadeer 2018] are based on the slightly more general setting of *semi-commutativity* (or "left" and "right movers"). Previous work [Farzan et al. 2022; Farzan and Vandikas 2019] has shown semi-commutativity to be equally applicable to applicable for reductions induced by lexicographic preference orders (as in this paper). Both our proof rule (Section 4) and our algorithm (Section 6) generalize to semi-commutativity in a straightforward way. However, the result that our algorithm captures the proof rule precisely (and not just soundly), Theorem 6.12, depends on the assumption that the commutativity relation is symmetric.

Finally, certain specialized commutativity notions can be found in the literature, for instance based on a kind of *observational equivalence* [Koskinen and Bansal 2021]. Many such commutativity relations can be formulated in our framework for abstract commutativity, and may have a strong potential for verifying programs in many domains. Generally, there is a wide space of possibilities to automatically derive more sophisticated commutativity relations and to use them in our approach.

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. https://doi.org/10.1145/2535838.2535845

Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 375–402. https://doi.org/10.1007/978-3-030-99527-0_20

Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2019. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y

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, Vol. 9450)*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer, 233–248. https://doi.org/10.1007/978-3-662-48899-7_17

Duc-Hiep Chu and Joxan Jaffar. 2014. A Framework to Synergize Partial Order Reduction with State Interpolation. In *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8855)*, Eran Yahav (Ed.). Springer, 171–187. https://doi.org/10.1007/978-3-319-13338-6_14

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. https://doi.org/10.1145/1480881.1480885

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound sequentialization for concurrent program verification. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 506–521. https://doi.org/10.1145/3519939.3523727

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2023. *Appendix to: Stratified Commutativity in Verification Algorithms for Concurrent Programs.* Technical Report. Uploaded as supplementary material to this paper in the ACM Digital Library.

Azadeh Farzan and Anthony Vandikas. 2019. 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, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 200–218. https://doi.org/10.1007/978-3-030-25540-4_11

Azadeh Farzan and Anthony Vandikas. 2020. Reductions for safety proofs. *Proc. ACM Program. Lang.* 4, POPL (2020), 13:1–13:28. https://doi.org/10.1145/3371081

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 110–121. https://doi.org/10.1145/1040305.1040315

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. https://doi.org/10.1007/3-540-60761-7

Vineet Kahlon, Chao Wang, and Aarti Gupta. 2009. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 398–413. https://doi.org/10.1007/978-3-642-02658-4_31

Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. 2022. Ultimate GemCutter and the Axes of Generalization - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 479–483. https://doi.org/10.1007/978-3-030-99527-0_35

Eric Koskinen and Kshitij Bansal. 2021. Decomposing Data Structure Commutativity Proofs with mn-Differencing. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 81–103. https://doi.org/10.1007/978-3-030-67067-2_5

Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 79–102. https://doi.org/10.1007/978-3-319-96145-3_5

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721. https://doi.org/10.1145/361227.361234

Edward Ochmanski. 1995. Recognizable Trace Languages. In *The Book of Traces*, Volker Diekert and Grzegorz Rozenberg (Eds.). World Scientific, 167–204. https://doi.org/10.1142/9789814261456_0006

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. http://ieeexplore.ieee.org/document/6679412/

Received 2022-07-07; accepted 2022-11-07