Sound Sequentialization for Concurrent Program Verification

Azadeh Farzan
University of Toronto
Canada
azadeh@cs.toronto.edu

Dominik Klumpp
University of Freiburg
Germany
klumpp@informatik.uni-freiburg.de

Andreas Podelski
University of Freiburg
Germany
podelski@informatik.uni-freiburg.de

Abstract

We present a systematic investigation and experimental evaluation of a large space of algorithms for the verification of concurrent programs. The algorithms are based on sequentialization. In the analysis of concurrent programs, the general idea of sequentialization is to select a subset of interleavings, represent this subset as a sequential program, and apply a generic analysis for sequential programs. For the purpose of verification, the sequentialization has to be sound (meaning that the proof for the sequential program entails the correctness of the concurrent program). We use the concept of a preference order to define which interleavings the sequentialization is to select ("the most preferred ones"). A verification algorithm based on sound sequentialization that is parametrized in a preference order allows us to directly evaluate the impact of the selection of the subset of interleavings on the performance of the algorithm. Our experiments indicate the practical potential of sound sequentialization for concurrent program verification.

CCS Concepts: • **Theory of computation** → **Program verification**; Regular languages; **Automated reasoning**; *Verification by model checking*.

Keywords: Concurrency, Sequentialization, Partial Order Reduction

ACM Reference Format:
Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound Sequentialization for Concurrent Program Verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523727

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523727

1 Introduction

Sequentialization à la KISS [32] has established itself as a promising approach in the analysis of concurrent programs. The issue that sequentialization addresses is known as the explosion problem: In the *interleaving model* where nondeterministic thread switches are possible at every control location, the space of interleavings grows exponentially in the number of threads.[1] There is a wide body of research [20, 21, 26, 27, 33, 34] on approaches based on sequentialization, with successful methods that can find subtle, previously undetected bugs, e.g., in bluetooth drivers. The underlying idea is to select a subset of interleavings, represent this subset by a (nondeterministic) sequential program, and apply a generic analysis for sequential programs. This paper investigates the idea of sequentialization for the algorithmic verification of concurrent programs. Algorithmic verification here broadly refers to algorithms to automatically find and check a program proof in the form of Floyd/Hoare-style assertions, typically with the aid of an SMT solver.

Intuitively, many program interleavings are equivalent because many pairs of statements in different threads *commute* (they are *independent*); i.e., they can be ordered either way without changing the result of the execution of the interleaving. In this case, the analysis of one interleaving already covers its entire equivalence class.

To capture the general idea of sequentialization in the context of verification, we use the concept of a *sound reduction*. We refer to a subset of the set of all interleavings of the concurrent program as a *reduction* [15]. The reduction is *sound* if it contains at least one representative for each equivalence class of interleavings. Thus, a sound sequentialization of a program amounts to the choice of a sound reduction. We can now express verification based on sequentialization in terms of the following two steps: the construction of a sound reduction and the verification of the reduction. In the context of verification (as opposed to falsification), the sequentialization must be sound for the outcome of the verification to be sound.

In the literature, special cases of sound sequentialization have been studied. In [11, 24, 28], programs are sequentialized by combining actions within a thread in atomic blocks,

[1] Terminology: An *interleaving* is a sequence of statements along a path in the product of the control flow graphs of the individual threads of the concurrent program.

i.e., we obtain the reduction by dropping interleavings that do not correspond to sequential compositions of atomic blocks. In [9, 35], programs are sequentialized by placing context switches after communication between threads, i.e., the reduction is obtained by dropping interleavings that do not conform to this policy.

We carry out a systematic investigation and experimental evaluation of a large space of algorithms that perform sound sequentializations. We base the investigation on the concept of a *preference order*, an order defined over the set of program interleavings. For each equivalence class of interleavings, to chose the representative to be included in the reduction, one takes the smallest interleaving according the the preference order (i.e. *the most preferred*) among all interleavings in the class. By definition, the reduction defined by a preference order is sound. If the preference order is a total order, the reduction is *minimal*: i.e., the reduction includes precisely one representative for each equivalence class and will no longer be sound if one of its elements is removed.

We use a wide range of preference orders and obtain a large space of sequentializations. A *lexicographic* preference order is defined by a total order on the alphabet of statements (reductions that correspond to lexicographic preference orders have been studied in the context of *trace monoids*; see [30]). In the special case where the total order (on statements) is based on the thread owners of statements, the lexicographic preference order favours interleavings that choose to context switch out of a thread at the latest possible point. This resembles the sequentializations in [11, 24, 28].

We introduce the class of *positional* preference orders by allowing the total order on the alphabet of statements to depend on the context in which we select the next statement in the interleaving. Since the set of program statements is finite, there are only finitely many total orderings (i.e. permutations) of the alphabet of statements. Additionally, there are finitely many control locations in a concurrent program with a bounded number of threads. We exploit these two facts in order to finitely represent a positional preference order, using finite automata to keep track of the context (i.e. the current control location). As a special case, positional preference orders can capture synchronous reductions [9, 35]: We can change the order on the alphabet of statements after every communication between threads.

The technical contribution of the paper is an effective method that, for every choice of a preference order, constructs a *space-efficient* finite representation of a *minimal* sound reduction of a given program. That is, the method is *parametric* in the particular preference order.

The fact that the method is *parametric* in the choice of a preference order allows us to carry out a series of experiments for a range of preference orders, and evaluate the impact of the preference order on the performance of the verification algorithm. Schematically, a verification algorithm goes through *refinement rounds* and successively checks a proof candidate until the check succeeds and a proof has been found. The performance of the algorithm is determined by the two aspects, proof finding and proof checking (the number of refinement rounds needed to find a proof, and the efficiency of the check in each refinement round). Since a reduction is an (in general, infinite) language of interleavings (which are words over the alphabet of thread statements) and can be (finitely) represented by a deterministic finite automaton (DFA), it is natural to phrase the verification algorithm in the general terms of *trace abstraction refinement* [19]: Here, the input program is already represented by a DFA (i.e. the control flow graph, with entry and exit location as initial resp. final state). That is, the verification algorithm takes as input a finite representation of a reduction in *exactly* the same way as a sequential program.

The fact that the reduction is *minimal* reflects the ideal situation where the burden for the proof cannot be made (relatively) any smaller; the proof needs to cover only the interleavings in the reduction but every interleaving in a minimal reduction requires a proof. We represent a (candidate) proof by a set of assertions and a corresponding set of Hoare triples with statements from the individual threads. We say that the proof *covers* an interleaving if there exists a consecutive sequence of Hoare triples in the given set that follows the statements in the interleaving. Different choices of minimal reductions, as induced by different preference orders, may lead to substantially different proofs, in terms of the number and type of assertions in the complete proof, the number of refinement rounds to find it, and whether a proof can be found through an automated process at all [15, 16].

The fact that the method constructs a *space-efficient* finite representation of a (minimal) reduction (for every choice of a preference order) is relevant because the finite representation is an input to the check of a candidate proof; its size is the dominating factor for the efficiency of the check. Checking a candidate proof amounts to checking that it covers each interleaving in the reduction, which is decidable (it can be reduced to the inclusion between two DFA). Section 7 discusses the proof check and shows that we can tightly integrate the construction of the reduction DFA *on the fly* with the proof check. The choice of preference order has an impact on the cost of the proof check because it depends on the preference order whether a space-efficient finite representation for the reduction exists. We show the efficiency of our construction through a theoretical argument for a special case in which a linear-size finite representation exists.

The overall contribution of the paper is a verification algorithm that is parametrized by the preference order. The performance of every (automated) verification algorithm is determined by the two aspects of proof finding and proof checking, and in ours, the choice of the preference order has an impact on both aspects. The key insight from our experiments is that the impact of the choice of the preference order on the proof finding aspect plays a larger role in the

overall performance than its impact on the proof checking aspect. This is in line with our intuition that efficiency in proof checking is not of much use if it never succeeds.

Roadmap. We demonstrate key features of our approach on a motivating example in Section 2. After a background section on concurrent programs (Section 3), we study preference orders that define (minimal) reductions for which finite representations exist (Section 4). Section 5 gives an algorithmic construction of such finite representations, based on the concept of *sleep sets*. While this construction is effective, it is not geared towards space-efficiency. Section 6 presents an approach that is geared towards space-efficiency, based on the concept of *persistent sets*. The final outcome of this section is a construction of *space-efficient* representations for *minimal* reductions (based on an integration of sleep sets and persistent sets). Section 7 presents an *on the fly* algorithm that integrates the construction from Section 6 into the proof check for (the finite representation of) a reduction. Section 7 furthermore investigates how we can benefit from this integration by making the construction proof-sensitive. We present our experimental evaluation in Section 8. We conclude with a comparison to related work (Section 9).

Proofs for our theoretical results can be found in the appendix [13]. The artifact accompanying the paper [14] contains the detailed evaluation results and information on how to reproduce the results.

2 Motivating Example

For our motivating example, we take a corrected version of the bluetooth device driver code, a classical example in sequentialization literature [32].[2] The driver consists of a `User` method and a `Stop` method (Figure 1(a,b)). The setup for our example has a fixed number of threads running `User` and one thread running `Stop`, which tries to shut down the driver. This can only be done when the device is not in use, that is, there is no user thread currently using the driver. Initially, the variable *pendingIo* has value 1, and the boolean variables *stoppingFlag*, *stoppingEvent* and *stopped* are *false*.

Correctness is encoded by an `assert` statement present in only one of the user threads (due to the symmetry of the program), which captures the fact that, if the device is in use by this thread, then the driver has not been stopped. We generalized the example by letting user threads enter and exit the device in a loop for an arbitrary number of times.

Verifying a Reduction. Observe that the *enter* actions of two different user threads *commute*, i.e. the order in which they are executed does not affect the reachable program states. The same is true for two *exit* actions of two different user threads. Therefore, the following traces (i.e. sequences of program statements) are all equivalent up to arbitrary commutation of these actions, and the safety of either implies the safety of the other two:

$\tau_1 = enter_3;\ enter_1;\ assert;\ enter_2;\ exit_3;\ close;\ exit_1;\ exit_2$

$\tau_2 = enter_2;\ enter_1;\ assert;\ enter_3;\ exit_2;\ close;\ exit_1;\ exit_3$

$\tau_3 = enter_1;\ enter_2;\ enter_3;\ assert;\ exit_1;\ exit_2;\ exit_3;\ close$

The *sleep set* algorithm of Section 5 constructs a *minimal* reduction of the bluetooth program for the purpose of verification. The verification then needs one refinement round less than without reduction, but incurs some overhead in time and space. Conversely, *persistent sets* as in Section 6 improve on space and time per refinement round, but require the same number of refinement rounds.

In this example, the (automatically generated) proof of correctness without reductions uses the assertions $\top$, $\bot$, $\neg stopped$, $stoppingFlag$ as well as assertions $pendingIo \geq C \wedge \neg stoppingEvent$ for constants $C \in \{1, \ldots, n\}$, where n is the number of user threads. Every program trace that contains a failing *assert* can be given a Floyd/Hoare annotation using these assertions, such that the first assertion is $\top$ and the last is $\bot$. This proves that such traces do not correspond to feasible program executions, and the program is correct. The central thesis of this paper is that the most gain in using reductions for program verification stems from the simplification of the proof. The discussed proof effectively counts how many threads have entered but not exited the device. Therefore, the number of assertions grows linearly with the number of threads. The commutativity discussed yields a reduction that excludes infinitely many behaviours, but does not admit a significantly simpler proof.

Simpler Proofs with Conditional Commutativity. Consider the *enter* action of one thread against the *exit* action of another. They commute *under the condition* that $pendingIo > 1$. A refinement of our algorithm (see Section 7) takes advantage of this type of conditional commutativity information. With this additional commutativity, the equivalence classes of program behaviours become larger, and the reduction ends up with fewer interleavings in it. Most importantly, this reduction admits a much simpler proof than the one discussed before.

We have $pendingIo > 1$ in any context where at least one user thread has entered but not yet exited, and the `Stop` thread has not yet executed `Close`. In such contexts, all actions of all remaining user threads commute, and all but one interleaving can be soundly excluded. For the proof of this reduction, there is no need to *count* the threads anymore; every user thread, bar one, enters and exits without interruption, and the counter never exceeds 2. The assertions $\top$, $\bot$, $\neg stopped$, $stoppingFlag$ as well as $pendingIo \geq 1 \wedge \neg stoppingEvent$ and $pendingIo \geq 2 \wedge \neg stoppingEvent$ suffice to prove correctness; i.e. every trace of the reduction that contains a failing *assert* can be annotated with these assertions. By a commutativity argument, it follows that *no*

[2] The version from [32] is the original one (where a bug was detected); a corrected version is discussed, e.g., in [7, 12, 31].

```
// User method:
while (*) {
  atomic { // Enter
    assume !stoppingFlag;
    pendingIo := pendingIo+1;
  }

  // do work
  assert !stopped;

  atomic { // Exit
    pendingIo := pendingIo-1;
    if (pendingIo == 0) {
      stoppingEvent := true;
    }
  }
}
```

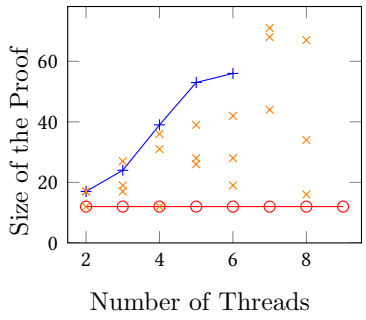
(a)

```
// Stop method:
stoppingFlag := true;

atomic { // Close
  pendingIo := pendingIo-1;
  if (pendingIo == 0) {
    stoppingEvent := true;
  }
}

assume stoppingEvent;
stopped := true;
```

(b)



(c)

Figure 1. Bluetooth driver code and verification results.

program trace containing a failing *assert* can be executed, and therefore the program is correct.

With this reduction, the number of assertions is constant in the number of threads. Our tool verifies instances of the bluetooth example using a constant number of assertions (i.e. 12) and a constant number of refinement rounds (i.e. 3). The key observation is that conditional commutativity significantly simplifies the proof in this example.

Different Reductions. Beyond the reduction discussed above (which we refer to as *sequential composition)*, there are many alternative choices of reductions. For example, one can opt for a preference order that interleaves user threads in a *lockstep* manner. In general, there are infinitely many choices of (computable, minimal) reductions and different reductions admit different proofs. Figure 1(c) demonstrates the effect of the choice of reduction: It compares proof sizes for reductions of instances of the bluetooth example which range from 2 to 10 threads. The red circles refers to sequential composition, and the blue "+" refers to lockstep. Other data points ("×") are from three random choices of preference orders, which yield wildly different proof sizes. Unlike in this example, sequential composition does not always yield the smallest proof (see Section 8). Hence, we do not target one specific choice of reduction but rather let our verification algorithm be parametric in the choice of a preference order.

3 Concurrent Programs

A concurrent program P is composed of a fixed number of threads $P = T_1 \parallel \ldots \parallel T_n$. Each thread T_i is given by a control flow graph with initial location ℓ^i_{init} and a distinguished exit location ℓ^i_{exit}. The location ℓ^i_{exit} has no outgoing edges, but all other locations have at least one. The size $|T_i|$ of a thread is the number of control flow locations; the size of P is the sum of the sizes of all threads, i.e., $\mathit{size}(P) := \sum_{i=1}^{n} |T_i|$.

We interpret the control flow graph of a thread as a deterministic finite automaton $(Q_i, \Sigma_i, \delta_i, \ell_{\text{init}}, \{\ell_{\text{exit}}\})$. Program locations are states, ℓ^i_{init} is the initial state, ℓ^i_{exit} is the (only) accepting state, and the alphabet Σ_i consists of the program statements in thread T_i. We distinguish statements of different threads, i.e., $\Sigma_i \cap \Sigma_j = \emptyset$ whenever $i \neq j$.

We identify P with the *interleaving product automaton* $P = (Q_1 \times \ldots \times Q_n, \Sigma, \delta, \langle \ell^1_{\text{init}}, \ldots, \ell^n_{\text{init}} \rangle, \{\langle \ell^1_{\text{exit}}, \ldots, \ell^n_{\text{exit}} \rangle\})$. The alphabet Σ is the set of all statements of all threads, i.e., $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$. For statement a of thread T_i, we define $\delta(\langle \ell_1, \ldots, \ell_n \rangle, a) = \langle \ell_1, \ldots, \ell_{i-1}, \delta_i(\ell_i, a), \ell_{i+1}, \ldots, \ell_n \rangle$, if $\delta_i(\ell_i, a)$ is defined. A trace (sequence of statements) of the program P is a word accepted by this automaton, i.e., an interleaving of statements from different threads such that each thread ends up in its exit location. Since the interleaving product automaton's size grows exponentially in the number of threads, the algorithms we present in the subsequent sections do not fully construct this product and hence (often) avoid paying exponential cost.

We specify correctness of a concurrent program P via a pre/postcondition-pair $(\mathit{pre}, \mathit{post})$, where pre and post are global assertions over the program variables. Though the extension to the more general case where correctness is specified via assert statements opens up a few interesting points, our formal exposition focuses on the pre/postcondition setting for simplicity of presentation. Section 6.1 discusses some of the difficulties, and our implementation analyses programs with asserts.

Finite Automata. We assume familiarity with deterministic finite automata (DFA), given as $A = (Q, \Sigma, \delta, q_{\text{init}}, F)$ with the usual components. δ is a partial function; we say that A is total if δ is. For a state q, $\mathit{enabled}(q)$ is the set of letters a such that $\delta(q, a)$ is defined. $\mathcal{L}_A(q)$ denotes the language of all words accepted from state q, and $\mathcal{L}(A)$ the language recognized by A. For $w \in \Sigma^*$, $\delta^*_+(w)$ denotes the state reached from q_{init} by reading the longest prefix of w for which a run exists. $|A|$ denotes the number of reachable states of A.

When a result does not refer to DFA of a general form, only to interleaving product automata of concurrent programs, we point this out and use the symbol P instead of A.

4 Reductions

Our goal is to compute language-minimal sound reductions, i.e., reductions that contain *exactly* one representative for each equivalence class. There exists a wide space of such reductions, depending on which representatives are chosen. The choice of representatives significantly impacts proof simplicity and proof check efficiency. We characterize this choice in terms of partial orders over words, called *preference orders*. The representative for each class is the minimal element of the class wrt. this preference order. We introduce a class of preference orders, such that the induced reductions can be effectively and efficiently computed. This characterization allows us to precisely describe what the algorithms in sections 5 and 6 compute, and to contrast different reductions through a language-theoretic study.

Formally, Mazurkiewicz equivalence [30] is a relation between words over an alphabet Σ. Given a symmetric *commutativity relation* $\leftrightarrows \subseteq \Sigma \times \Sigma$, words w_1, w_2 are considered equivalent (denoted $w_1 \sim w_2$) if we can get from w_1 to w_2 by repeatedly swapping adjacent commuting letters. Specifically, $\sim$ is the least reflexive-transitive relation such that $uabv \sim ubav$ for all $u, v \in \Sigma^*$ and $a, b \in \Sigma$ with $a \leftrightarrows b$. A language L is closed if it is a (possibly infinite) union of equivalence classes.

Definition 4.1 (Reduction). *We call L' a* reduction *of L iff $L' \subseteq L$ and for every $w \in L$ there exists a Mazurkiewicz-equivalent representative $v \in L'$.*

For now, we leave the commutativity relation $\leftrightarrows$ parametric, as most of our results are independent of the specific relation. In the case where the alphabet Σ is the set of statements in a concurrent program P, we assume that two statements of the same thread never commute. This assumption is sufficient to guarantee that the language $\mathcal{L}(P)$ of a concurrent program is closed. In Section 7, we discuss a suitable choice for commutativity of program statements.

4.1 Lexicographic Preference Orders

As a first step towards a class of preference orders, we fix a total strict order $<$ over alphabet Σ (the set of program statements), and consider the induced *lexicographic preference order* $\preceq$ on words. Given a language and a commutativity relation over Σ, such a preference order induces a reduction:

Definition 4.2 (Lexicographic Reductions). *Let L be a closed language. The reduction of L induced by $\preceq$ is the set of $\preceq$-minimal representatives in L for each equivalence class.*

$$red_{\preceq}(L) := \{ \min_{\preceq}[w] \mid w \in L \}$$

Here, $[w]$ denotes the Mazurkiewicz equivalence class of w. The language $red_{\preceq}(L)$ contains exactly the (unique) minimal representative of each equivalence class wrt. the lexicographic order induced by $<$, and is thus indeed a reduction of L. The input L only influences *which* equivalence classes are represented in the reduction. The choice of *how* they are represented, i.e., the representative for each class, is independent of L. Formally, we have $red_{\preceq}(L) = L \cap red_{\preceq}(\Sigma^*)$. Preference orders separate the choice of representatives from L, allowing to study the effect of the same choice mechanism on different languages.

It is a well-known result [30] that for a closed regular language L, the reduction $red_{\preceq}(L)$ is also regular. The question of the size of a DFA representing this regular language naturally arises, as the efficiency of our proof check directly depends on the size of the automaton representing the reduced program. We investigate how the two parameters – the commutativity relation, and the preference order – influence the reduction language's space complexity. For our first goal of decreasing the number of refinement rounds, more commutativity is always preferable, as it leads to a smaller reduction language. This is not true of the reduction's space complexity: A DFA for the reduction $red_{\preceq}(L)$ of a regular language L may even need more states than a DFA for L itself.[3] For the investigation into the effect of the preference order, recall that we compute reductions of concurrent programs P given as interleaving product $P = T_1 \parallel \ldots \parallel T_n$, and that the alphabet is given as $\Sigma = \Sigma_1 \uplus \ldots \uplus \Sigma_n$. We consider the ideal (in terms of the language) case of *full commutativity*, i.e., all statements of different threads commute:

$$\forall i, j \,.\, i \neq j \implies \forall a \in \Sigma_i, b \in \Sigma_j \,.\, a \leftrightarrows b$$

Even in this case, the complexity of the reduction language can be exponential in $size(P)$. However, picking the right lexicographic preference order can avoid this explosion. In fact, consider a very natural kind of preference order that is based simply on an underlying ordering of threads; formally:

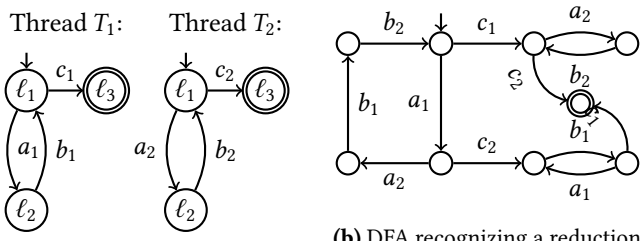
$$i \neq j,\ (\exists a \in \Sigma_i, b \in \Sigma_j \,.\, a < b) \implies \forall a \in \Sigma_i, b \in \Sigma_j \,.\, a < b$$

We call the induced lexicographic preference order *thread-uniform*.

Theorem 4.3. *Let $\preceq$ be a thread-uniform lexicographic preference order. Under full commutativity, the induced lexicographic reduction $red_{\preceq}(\mathcal{L}(P))$ is recognized by a DFA with linearly many states in $size(P)$.*

A linear number of states is as good as we can expect, since the reduction DFA must still contain every program statement. The reduction induced by a thread-uniform preference order is exactly the sequential composition of threads. This corresponds to a fixed-priority scheduling discipline. If we do not have full commutativity, the reduction may include traces that deviate from this scheduling discipline. In

[3] This may surprise readers familiar with partial order reduction literature [2, 17]. However, such works typically assume the input program does not have branches and loops, or only over-approximate the reduction $red_{\preceq}(L)$.



(a) A concurrent program P

(b) DFA recognizing a reduction of P (under full commutativity)

Figure 2. A reduction not induced by any (classic) lexicographic preference order

such cases, the reduction will still contain one representative per equivalence class: For classes that contain an interleaving that adheres to the scheduling discipline, this interleaving will be picked as representative; for classes that do not contain any interleaving that adheres to the scheduling discipline, some other representative is chosen. Only in the case of full commutativity, a direct link between the reduction and the mentioned scheduling discipline can be drawn. Certain other scheduling disciplines however can not be approximated with lexicographic preference orders.

Example 4.4. Consider the program in Figure 2a. Under full commutativity, the DFA in Figure 2b recognizes a reduction that approximates lockstep scheduling. This reduction is not induced by any lexicographic preference order: For the accepted word $a_1a_2b_1b_2a_1a_2b_1b_2c_1c_2$ to be minimal, we would need $a_1 < a_2 < b_1 < b_2 < a_1$, which is a contradiction.

We introduce a more general class of preference orders that admit a wider class of reductions, while still being finitely representable and thus algorithmically treatable.

4.2 Positional Lexicographic Preference Orders

The idea behind *positional* lexicographic preference orders is to vary the underlying order between letters depending on the prefix (of the word) leading to the context for the order. To ensure finite representability, we require that a finite automaton be able to decide the appropriate letter ordering. It may be helpful to think of a *transducer* representing the order; for any prefix word u, the transducer outputs a total order on Σ (a finite set of possibilities). In the following, we assume a DFA A over alphabet Σ with transition function δ, such that $\mathcal{L}(A)$ is closed. Further, let $\preccurlyeq$ be a mapping from states q of A to total strict orders $<_q$ over Σ.

Definition 4.5 (Positional Lexicographic Preference Orders)**.** *The A-positional lexicographic preference order induced by $\preccurlyeq$ is the smallest relation $lex(\preccurlyeq)$ such that*

1. *for all words w, v, we have $(w, wv) \in lex(\preccurlyeq)$*
2. *for all words u, v, w, all letters a, b and a state q such that $q = \delta_+^*(u)$ and $a <_q b$, we have $(uav, ubw) \in lex(\preccurlyeq)$*

Note that the definable orders depend on the *structure* of the DFA A, not the recognized language. Two different DFA that recognize the same language may define different positional lexicographic preference orders. Classic lexicographic orders correspond to the case that the order $<_q$ is the same for all states q. We define the reduction induced by a positional lexicographic preference order analogously to Definition 4.2, and in the following, refer to such reductions as *lexicographic reductions*. Lexicographic reductions form a subclass of the class of *S-reductions* introduced in [16].

Example 4.6. Consider again Figure 2. Let $\preccurlyeq$ be a mapping from states of P to total strict orders over letters, such that $a_1 <_q b_1 <_q c_1 <_q a_2 <_q b_2 <_q c_2$ for $q \in \{\langle \ell_1, \ell_1 \rangle, \langle \ell_2, \ell_2 \rangle\}$, and $a_2 <_{q'} b_2 <_{q'} c_2 <_{q'} a_1 <_{q'} b_1 <_{q'} c_1$ for all other states q'. The corresponding P-positional lexicographic preference order $\preceq := lex(\preccurlyeq)$ satisfies $a_1a_2b_1b_2c_1c_2 \preceq a_1b_1c_1a_2b_2c_2$: After the prefix a_1, P reaches the state $q := \langle \ell_2, \ell_1 \rangle$, and we have $a_2 <_q b_1$. Further, we have $a_1b_1c_1a_2b_2c_2 \preceq a_1b_1a_2b_2c_1c_2$: After prefix a_1b_1 we reach $q' := \langle \ell_2, \ell_2 \rangle$, and $c_1 <_{q'} a_2$. Figure 2b shows a DFA for the induced lexicographic reduction (under full commutativity). The words above are equivalent, and only the minimal word $a_1a_2b_1b_2c_1c_2$ is accepted.

This reflects a general approach for approximating lockstep scheduling as preference order: Statements are ordered by their thread, and if a state q is first (i.e., via a minimal interleaving) reached through an edge labeled by a statement from thread i, we rotate the ordering of threads such that $<_q$ orders statements of thread i after all other statements.

All lexicographic reductions are *language-minimal*, i.e., no proper subset of $red_{lex(\preccurlyeq)}(L)$ is a reduction of L. This means that every equivalence class of L has *exactly* one representative in $red_{lex(\preccurlyeq)}(L)$. Thus our verification approach is never obliged to prove two redundant interleavings. Furthermore, these reductions can be finitely represented.

Theorem 4.7. *The lexicographic reduction $red_{lex(\preccurlyeq)}(\mathcal{L}(A))$ is regular and language-minimal.*

In our setting, the languages representing concurrent programs are always regular and closed. Therefore, Theorem 4.7 permits us to conclude that a *lexicographic reduction* $red_{lex(\preccurlyeq)}(\mathcal{L}(P))$ of program P is always regular and can thus be seamlessly integrated in a trace abstraction refinement loop as described in Section 1. In terms of reduction size and proof check efficiency, this richer class of preference orders can also cause exponential explosion. Despite this, our interest in this class of preference orders is completely justified when attempting to compute a provable program reduction. For certain programs, positional lexicographic preference orders admit simple proofs, while non-positional lexicographic preference orders require complicated proofs out of reach for automated methods [15]. Compared to this, the finite proof checking overhead takes second place. Our evaluation in Section 8 shows the impact of this.

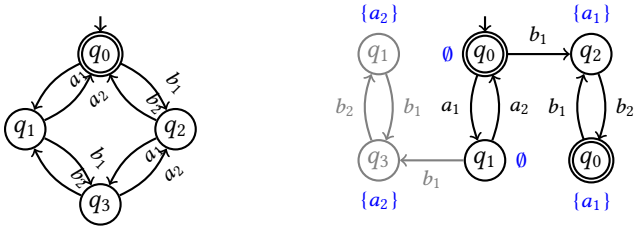


Figure 3. A DFA and the corresponding sleep set automaton.

We have identified a class of language-minimal reductions, and investigated the optimal size of their corresponding finite representations. In the next sections, we turn to effective constructions of automata recognizing such reductions.

5 Finite Representations

We present an effective construction of finite representations for the class of lexicographic reductions introduced in the previous section. Specifically, let us fix a finite automaton $A = (Q, \Sigma, \delta, q_{\text{init}}, F)$ such that $\mathcal{L}(A)$ is closed, and a mapping $\lessdot$ from states of A to total strict orders on Σ. We construct an automaton $\mathfrak{S}_{\lessdot}(A)$ that recognizes precisely the lexicographical reduction $red_{lex(\lessdot)}(\mathcal{L}(A))$. We focus here purely on constructing *some* DFA that recognizes this language-minimal reduction, motivated by the improvements in the number of refinement iterations this allows. Section 6 then focuses on proof check efficiency. Here, we adapt the *sleep set* approach from partial order reduction literature [17].

Definition 5.1 (Sleep Set Automaton)**.** *We define the sleep set automaton* $\mathfrak{S}_{\lessdot}(A) := (Q \times 2^{\Sigma}, \Sigma, \delta_{\mathfrak{S}}, \langle q_{\text{init}}, \emptyset \rangle, F \times 2^{\Sigma})$*, where*

$$\delta_{\mathfrak{S}}(\langle q, S \rangle, a) := \begin{cases} \textit{undefined} & \textbf{\textit{if}}\ a \in S \ \textit{or}\ \delta(q,a)\ \textit{undefined} \\ \langle \delta(q,a), S' \rangle & \textbf{\textit{else}} \end{cases}$$

with $S' = \{\, b \in \textit{enabled}(q) \mid (b \in S \vee b <_q a) \wedge a \leftrightarrows b \,\}$.

The sleep set automaton maintains a set of letters at each state (the eponymous *sleep set*), and prunes outgoing edges labeled with such a letter. The construction guarantees that for any accepting run using a pruned edge, a lexicographically smaller equivalent representative still exists. To achieve a language-minimal reduction, our construction not only prunes edges, but also unrolls loops and branches in the input automaton, by distinguishing multiple occurrences $\langle q, S_1 \rangle$ and $\langle q, S_2 \rangle$ of a state q with different sleep sets $S_1 \neq S_2$.

Example 5.2. Figure 3 shows a DFA and the corresponding sleep set automaton, for the non-positional order $a_1 < a_2 < b_1 < b_2$, and if a_i and b_j commute for all i, j. Sleep sets are annotated in blue. At state $\langle q_2, \{a_1\} \rangle$, the outgoing edge labeled with a_1 has been pruned: All words reaching this state have the form wb_1 for some $w \in \{a_1, a_2\}^*$, and so the extension wb_1a_1 has a lexicographically smaller equivalent representative wa_1b_1. The state q_0 has been duplicated to achieve minimality: For words w leading to state $\langle q_0, \{a_1\} \rangle$, the extension wa_1 has a lexicographically smaller representative, but this is not the case for words leading to state $\langle q_0, \emptyset \rangle$.

Theorem 5.3. *The sleep set automaton* $\mathfrak{S}_{\lessdot}(A)$ *recognizes exactly the lexicographic reduction* $red_{lex(\lessdot)}(\mathcal{L}(A))$ *of* $\mathcal{L}(A)$.

Hence the sleep set automaton gives us a recognizer for a language-minimal reduction. On the other hand, sleep sets only decrease the number of transitions compared to the input automaton, but not the number of states (see for instance theorem 5.4 in [17]). This also holds for our variation; unrolling may even duplicate states. As shown in Figure 3, some of the reachable states (in gray) may be unnecessary: They cannot reach an accepting state. This problem (called *sleep set-blocked executions* [1]) has been discussed extensively in the literature. Our duplication of states makes this issue even more pressing. Section 6 augments our construction in order to deal with this problem.

6 Space-Efficient Representations

While the minimal (in terms of paths) reduction computed by sleep sets is useful during the construction of the proof candidate, there is no state pruning. The size of the reduction DFA (and thus the cost of the proof check in our verification method) is still exponential in the program size. As seen in Example 5.2, this may be caused by useless states, from which no accepting state can be reached. We now discuss a method to eagerly (but soundly) prune edges to such states, so that we never spend the time and memory to construct the full (exponentially large) automaton. In Section 7, we show how this results in an efficient proof check.

6.1 Sound Pruning

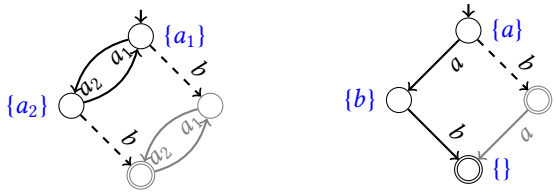
In this section, we discuss sufficient conditions that allow us to soundly prune edges from an automaton state (soundly in the sense that each equivalence class of a word accepted by the automaton is still represented). By pruning edges, and thus implicitly removing states that become unreachable, we obtain a reduction automaton with a smaller number of states. Later, in Section 7.1, we will see an algorithm that computes a set of edges which can soundly be pruned.

Given a DFA A of the form $A = (Q, \Sigma, \delta, q_{\text{init}}, F)$, let π be a mapping that assigns to each state of A a subset of its outgoing edges. The idea is to prune from each state all edges that are *not allowed* by π at that state. We call the resulting automaton $A_{\downarrow\pi}$ the π-reduction of A. Formally,

$$A_{\downarrow\pi} := (Q, \Sigma, \delta_{\downarrow\pi}, q_{\text{init}}, F)$$

where $\delta_{\downarrow\pi}(q, a)$ is undefined if $a \notin \pi(q)$, and equal to $\delta(q, a)$, otherwise. We define two concepts (*weakly persistent* sets and *membranes*) that we use to identify mappings π such that the corresponding π-reduction is sound.

Definition 6.1 (Weakly Persistent Sets)**.** *A subset of outgoing edges* $M \subseteq \textit{enabled}_A(q)$ *is* weakly persistent *in the state*



(a) The *ignoring problem*: The persistent sets (in blue) prune all b-transitions.

(b) Persistent sets prune the dashed transition, leaving no representative for the word b.

Figure 4. Problems solved by weakly persistent membranes.

q if for all words $a_1 \dots a_m$ accepted by a run starting in q, if a letter in the word (at, say, the i-th position) does not commute with one of the letters in M, then there exists an earlier letter in the word (before or at the i-th position) that lies inside of M.

$$a_1 \dots a_m \in \mathcal{L}_A(q),\ b \in M,\ a_i \not\leftrightarrows b \ \Rightarrow\ \exists j \leq i \,.\, a_j \in M$$

Weakly persistent sets are similar to *persistent sets* [17], but less restrictive: We only quantify over accepted words $a_1 \dots a_n$, whereas persistent sets quantify over all runs.

Example 6.2. Consider the automata in Figure 3, and recall that we assume that a_i and b_j commute for all i, j. The set $\{a_2\}$ is weakly persistent in the state q_1: If a word $x_1 x_2 \dots x_m$ is accepted from q_1 and $x_i \not\leftrightarrows a_2$ (hence $x_i \in \{a_1, a_2\}$), then $x_j = a_2$ for some $j \leq i$. Pruning outgoing edges not labeled by a_2 removes the useless states shown in gray in Figure 3.

If we only consider concurrent programs as defined in Section 3 (with special exit locations and a specification given as pre/postcondition-pair), weakly persistent sets suffice to ensure sound pruning. However, this is not the case for general automata, and of particular relevance, it is not the case if we check correctness wrt. assert statements in each thread (a common way to specify correctness for concurrent programs, e.g. in our benchmarks in Section 8). Figure 4 illustrates two cases where weakly persistent sets allow for unsound pruning. To give a more complete picture, we present an additional condition on the chosen set of edges, through the concept of *membranes* (see Figure 5).

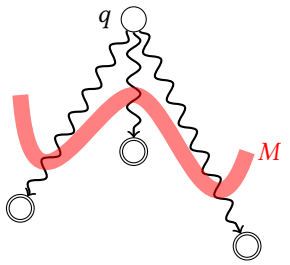


Figure 5. All accepting runs from q must "pass through" (i.e., contain a letter in) the membrane M.

Definition 6.3 (Membrane). *A set $M \subseteq \Sigma$ is a* membrane *for a state q if, whenever a non-empty word is accepted by a run starting in q, the word must contain a letter in M.*

$$a_1 \dots a_n \in \mathcal{L}_A(q),\ n \geq 1 \ \Rightarrow\ \exists i \,.\, a_i \in M$$

We use the examples in Figure 4a and Figure 4b to show how membranes address the two aforementioned problems. The annotated sets are weakly persistent, but not membranes. However, the set $\{b\}$ is a weakly persistent membrane for the initial state in both examples. Pruning with this set still decreases the number of reachable states. As noted before, for a concurrent program, weakly persistent sets alone already allow for sound pruning: For a state of a concurrent program, every weakly persistent set is also a membrane.[4]

We arrive at the soundness result for π-reduction (not specific to concurrent programs).

Theorem 6.4 (Soundness of π-Reduction). *Assume the language of A is closed, and $\pi(q)$ is a weakly persistent membrane for each state q of A. The π-reduced automaton $A_{\downarrow\pi}$ recognizes a reduction of $\mathcal{L}(A)$.*

We investigate the question whether the membrane condition can be relaxed, and show that *locally*, this is not possible.

Proposition 6.5. *The subset of outgoing edges assigned to the state q by the mapping π must be a membrane for q if $\mathcal{L}_{A_{\downarrow\pi}}(q)$ is a reduction of $\mathcal{L}_A(q)$.*

6.2 Space-Efficient and Minimal

In order to achieve our goals of proof simplicity and efficient proof checking, we need a method to compute a sound reduction that is both minimal (in terms of accepted paths) and compact (in terms of states). To this end, we combine the approaches of Section 5 and Section 6.1.

Again, we fix a DFA $A = (Q, \Sigma, \delta, q_{\text{init}}, F)$. Let $lex(\preccurlyeq)$ be an A-positional lexicographical preference order, and let π map states of A to weakly persistent membranes. We apply the π-reduction (see Section 6.1) to the sleep set automaton $\mathfrak{S}_{\preccurlyeq}(A)$ wrt. the preference order $lex(\preccurlyeq)$ (see Section 5) that we have obtained from the automaton A, and obtain the automaton

$$\left(\mathfrak{S}_{\preccurlyeq}(A)\right)_{\downarrow\pi}$$

Algorithmically, we do not construct those parts of the automaton $\mathfrak{S}_{\preccurlyeq}(A)$ made unreachable by the subsequent π-reduction (see Algorithm 2 for details). Care needs to be taken, as an arbitrary combination of a positional lexicographic preference order and a function π does not necessarily yield a sound reduction. A simple sufficient criterion is given by *compatibility*: We say that $lex(\preccurlyeq)$ is *compatible* with π if $<_q$ prefers non-pruned edges (in $\pi(q)$) over pruned edges (not in $\pi(q)$) for every state $q \in Q$.

$$\forall a, b \in \Sigma \,.\, a \in \pi(q) \wedge b \notin \pi(q) \implies a <_q b$$

It remains to clarify the choice of π, i.e., which weakly persistent membranes to use for a reduction of the sleep set automaton $\mathfrak{S}_{\preccurlyeq}(A)$. Fortunately, this is no more complicated

[4] For correctness wrt. assert statements, the edges of any thread with an assert statement must be included to ensure the membrane condition. To avoid including (in the worst case) all threads and being unable to prune, our implementation analyses correctness of the program with respect to asserts in each thread separately, preferring n analyses with (ideally) polynomial proof checking effort over a single analysis with exponential proof checks.

than the choice of weakly persistent membranes for the input automaton A: If M is a weakly persistent membrane for q, then $M \setminus S$ is a weakly persistent membrane for the state $\langle q, S\rangle$ of $\mathfrak{S}_{\preccurlyeq}(A)$. In the remainder of this paper we denote by $\pi_{\mathfrak{S}}$ the function with $\pi_{\mathfrak{S}}(\langle q, S\rangle) = \pi(q) \setminus S$. We arrive at the soundness theorem for the combined reduction:

Theorem 6.6 (Soundness of Combined Reduction)**.** *The automaton* $\left(\mathfrak{S}_{\preccurlyeq}(A)\right)_{\downarrow \pi_{\mathfrak{S}}}$ *recognizes the lexicographic reduction induced by the preference order* $lex(\preccurlyeq)$*.*

$$\mathcal{L}(\left(\mathfrak{S}_{\preccurlyeq}(A)\right)_{\downarrow \pi_{\mathfrak{S}}}) = red_{lex(\preccurlyeq)}(\mathcal{L}(A))$$

The relevance of the two reduction techniques in our context differs from the understood wisdom in model checking (e.g. chapter 8 of [17]): There, persistent sets are the key reduction technique, as the central objective is to reduce the number of states; sleep sets play a secondary role. We contrast this with our setting of concurrent program verification: Sleep sets are of first importance, allowing us to compute *language-minimal* reductions (admitting smaller proofs, which can be found with fewer refinement rounds). Weakly persistent sets play a secondary role in the subsequent step of checking the proof candidate. Put bluntly, if we don't succeed at constructing a good proof candidate, being able to check the proof candidate efficiently won't help us.

7 Proof Checking for Reductions

We have shown how compact recognizers for language-minimal reductions can be computed using techniques from partial order reduction literature. We now apply this result to the proof check of our verification approach. We first present an algorithm to compute weakly persistent sets for a concurrent program, and then integrate this algorithm in the overall proof checking approach.

7.1 Persistent Sets for Concurrent Programs

In Section 6, we have seen that, given a mapping π from states of an automaton to weakly persistent membranes, we can construct small and language-minimal reductions. The question is how one can compute such a mapping π. Here, we investigate the question for the setting where the automaton is an interleaving product automaton P, i.e., where P is defined from the parallel composition of threads, formally $P = T_1 \parallel \ldots \parallel T_n$ (see Section 3). Assume furthermore that $\preccurlyeq$ maps states of P to total strict orders on Σ.

The first step in the design of an algorithm to compute π is to define the problem to be solved. The issue here is to rule out trivial solutions for the mapping π. The first solution is to assign to each state q the whole set of its outgoing edges. This solution is useless as the corresponding π-reduction does not prune any state. The second solution is based on checking reachability in the (unreduced, i.e., exponentially large) state space of the interleaving product automaton. This solution is useless since it defeats the very purpose of

Preprocessing step: *Compute the conflict relation, i.e., the set* $\{(\ell_i, \ell_j) \mid \ell_i \rightsquigarrow \ell_j\}$

Procedure `CompatiblePersistentSet(`q`)`**:**

Input: state $q = \langle \ell_1, \ldots, \ell_n \rangle$ of program P
Output: set M such that M is weakly persistent at q
$active \leftarrow \left\{ i \in \{1, \ldots, n\} \mid enabled_{T_i}(\ell_i) \neq \emptyset \right\}$
$conflicts \leftarrow \{ (i, j) \in active^2 \mid \ell_i \rightsquigarrow \ell_j$
$\quad \vee \exists a \in enabled_{T_j}(\ell_j), b \in enabled_{T_i}(\ell_i) \,.\, a <_q b \}$
$SCCs \leftarrow$ strongly connected components of graph ($active$, $conflicts$)
$E \leftarrow$ topologically maximal SCC in $SCCs$
return $M := \bigcup_{i \in E} enabled_{T_i}(\ell_i)$

Algorithm 1: Computation of Weakly Persistent Sets

persistent sets, which is to avoid the combinatorial explosion in the size of the concurrent program. We narrow down the specification of the algorithm that computes π to three conditions: **(C1)** The corresponding π-reduction is sound; **(C2)** the algorithm has polynomial complexity (in the size of the concurrent program P); and **(C3)** in the case of full commutativity (every statement of a thread commutes with every statement of another thread), the π-reduced automaton is linear in the size of the concurrent program P. The case of a full commutativity relation is used as a test case for reduction algorithms; the question is whether, under the right circumstances, the reduction can be optimal.

Let $q = \langle \ell_1, \ldots, \ell_n \rangle$ be the state for which we aim to compute a weakly persistent membrane. A simple approach to compute persistent sets from the literature [2] is to pick some thread T_i such that the enabled letters of the thread's current location ℓ_i commute with all letters in other threads T_j. The set of enabled letters $enabled_{T_i}(\ell_i)$ then forms a persistent set. We use here a straightforward extension of this idea. Let us first introduce the notion of *conflict*: Location ℓ_i is *in conflict with* location ℓ_j of another thread, denoted $\ell_i \rightsquigarrow \ell_j$, if an outgoing edge a of ℓ_i does not commute with an outgoing edge b of some location ℓ'_j reachable from ℓ_j (within the thread T_j). We pick a subset of threads $E \subseteq \{1, \ldots, n\}$ that have not yet terminated ($enabled_{T_i}(\ell_i) \neq \emptyset$ for $i \in E$), such that E is conflict-closed ($\ell_i \rightsquigarrow \ell_j$, $i \in E \Rightarrow j \in E$). The enabled actions of all threads in E then form a weakly persistent set.

Algorithm 1 shows the complete procedure. We reduce the computation of a conflict-closed set E to the computation of strongly connected components of a graph where nodes represent threads and edges represent conflicts. Additional edges ensure compatibility with the preference order $lex(\preccurlyeq)$. To guarantee conflict-closedness, we select a topologically maximal component for E, i.e., a component where no node has an edge to another component. In the following, we denote by π the function implemented by Algorithm 1; and define $\pi_{\mathfrak{S}}$ analogously to Section 6.2 as $\pi_{\mathfrak{S}}(\langle q, S\rangle) = \pi(q) \setminus S$.

Proposition 7.1. *π is compatible with the preference order lex($\lessdot$), and maps states to weakly persistent membranes.*

Thus, Algorithm 1 satisfies **(C1)** in the specification given above. To see that it also satisfies **(C2)**, we note that the preprocessing step, i.e., the computation of the conflict relation $\rightsquigarrow$, can be done in $O(\sum_{1\leq i,j\leq n} |T_i| \cdot |T_j|)$, i.e., in $O(size(P)^2)$ time. Given the conflict relation $\rightsquigarrow$, the computation of SCCs and choosing one topologically maximal SCC can then be done in $O(n^2)$, i.e., in quadratic time in the number of threads. For **(C3)**, recall from Section 4 that not all preference orders admit linear-size reduction. However, for the case discussed there, our construction is optimal:

Theorem 7.2. *If the mapping $\lessdot$ is thread-uniform and non-positional ($<_q = <_{q'}$ for all q, q') and under full commutativity, the automaton $\mathfrak{S}_{\lessdot}(P)_{\downarrow\pi_{\mathfrak{S}}}$ has $O(size(P))$ reachable states.*

These results serve as validation for our method to compute weakly persistent membranes: The computed recognizers are sound, and typically significantly smaller than they would be for the trivial choice of weakly persistent membranes, in the ideal case even exponentially smaller. This is achieved through only polynomial effort for the computation of weakly persistent membranes, i.e., computing the compact recognizer for the reduction is not overly expensive.

7.2 Proof-Sensitive Sequentialization On The Fly

We integrate the combined sequentialization approach in our verification scheme in order to check if the current proof candidate is sufficient to prove correctness of the computed reduction. Our verification scheme takes as input DFA representing the threads of a concurrent program. If it terminates, it returns a proof that suffices to prove correctness for a (sound) reduction of the concurrent program. Our algorithm consists of *refinement rounds*. Each new refinement round takes the trace that is returned as a counterexample by the failed proof check in the previous refinement round (a trace in the reduction that is not covered by the proof), constructs a sequence of Hoare triples for the proof of the trace, and augments the set of assertions from the previous refinement round with the assertions from those Hoare triples. In the initial round, the set of assertions is empty. The terminal refinement round is the one where the proof check succeeds. The subprocedure that constructs a sequence of Hoare triples for the proof of the trace can be implemented, for example, by an interpolant-generating SMT solver.

The proof check subprocedure takes as input the DFA representing program threads and a set of assertions (the candidate proof). It computes a reduction DFA *on the fly* while simultaneously checking that the candidate proof covers each trace in the reduction. Indeed, the fact that we build up a set of assertions for the candidate proof also opens up a opportunity: The commutativity of statements in different threads may depend on an assumption about the context in which the execution of the two statements starts. A candidate proof may establish the validity of an assertion and thus guarantee the assumption, allowing for smaller reductions.

Formally, the candidate proof is given by a Floyd/Hoare automaton [19] for a pre/postcondition-pair (pre, $post$). A Floyd/Hoare automaton is conceptually similar to a (partial) annotation of the control flow graph (here, the interleaving product P) with inductive assertions. For the precise definition, we refer the reader to Heizmann et al. [19]. There, it is also shown how to construct and iteratively refine Floyd/-Hoare automata in a counterexample-guided verification scheme. Here, we focus on our efficient proof check based on sequentialization.

In the following, let $\mathcal{A}$ be a Floyd/Hoare automaton. As already shown in Section 2, our verification approach takes advantage of information provided by the proof candidate to compute even simpler reductions. To this end, we introduce the following definition:

Definition 7.3 (Proof-Sensitive Commutativity)**.** *Let φ be an assertion of $\mathcal{A}$. Statements a and b commute under condition φ, denoted $a \leftrightarrows_\varphi b$, iff the compositions ab and ba have the same semantics when starting from a state satisfying φ.*

A classical example where proof-sensitivity helps is the case of two statements that write to memory via different pointers. These statements do not commute generally, as the pointers may alias one another. If however φ states that they do not alias, then the statements commute under condition φ. The concept is more general though, as e.g. seen for the statements $enter_i$ and $exit_j$ from Section 2.

The attentive reader will note that this relation does not quite fit into the notion of commutativity relation as defined so far: It is parametrized in an assertion φ of $\mathcal{A}$. To accommodate this, we extend the notion of commutativity and Mazurkiewicz equivalence. As before, we say that a trace τ_1 and τ_2 are equivalent, denoted $\tau_1 \sim \tau_2$, if we can get from τ_1 to τ_2 by repeatedly swapping adjacent commuting letters. The difference is that commutativity of letters now depends on the prefix, specifically on the assertion guaranteed to hold by $\mathcal{A}$. Formally, $\sim$ is the least reflexive-transitive relation such that $uabv \sim ubav$ for all $u, v \in \Sigma^*$ and $a, b \in \Sigma$ with $a \leftrightarrows_\varphi b$, where $\varphi = \delta^*_{\mathcal{A}}(pre, u)$. All results presented so far still hold for this extended setting, with two exceptions: Firstly, we only guarantee that our construction over-approximates lexicographic reductions. Proof-sensitivity allows reductions to contain fewer traces, lowering the threshold for minimality. Our approach benefits from this less restrictive setting, and removes more traces. However, it does not quite improve all the way to the new minimality threshold. Secondly, our weakly persistent sets are not proof-sensitive: There, we only consider statements a and b commutative if they commute under condition $\top$. Proof-sensitive commutativity does not meet additional guarantees (e.g. weak uniformity [17]) needed to ensure soundness of persistent

A priori: $V \leftarrow \emptyset$

Procedure `CheckProof`(q, φ, S)**:**

Input: state q of program P, sleep set $S \subseteq \Sigma$, assertion φ of Floyd/Hoare automaton $\mathcal{A}$

if $\langle q, \varphi, S\rangle \in V$ **then return**

else if $q \in F$ *and* $\varphi \not\models$ *post* **then** "ctex. found"

$V \leftarrow V \cup \{\langle q, \varphi, S\rangle\}$

for $a \in$ `CompatiblePersistentSet`$(q) \setminus S$ **do**

$S' \leftarrow \{b \in \mathit{enabled}(q) \mid (b \in S \vee b <_q a) \wedge a \circlearrowright_\varphi b\}$

`CheckProof`$(\delta_P(q, a), \delta_{\mathcal{A}}(\varphi, a), S')$

end

Algorithm 2: Recursive procedure used by the proof check with proof-sensitive sequentialization on the fly.

sets in this setting. Combining normal weakly persistent sets with proof-sensitive sleep sets is sound and preserves the additional reduction (in terms of the language) afforded by proof-sensitivity.

For proof-sensitive commutativity, it is sufficient to prove correctness of one representative per equivalence class to conclude correctness of the entire program. Algorithm 2 takes advantage of this, by checking whether the proof candidate $\mathcal{A}$ suffices to prove correctness of a reduction of P. We do not necessarily construct the entire reduction if a counterexample is found early. Computing the reduction on the fly during proof checking further allows us to take advantage of proof-sensitive commutativity wrt. the states of $\mathcal{A}$. While this may seem self-referential, it is not a threat to soundness: The proof candidate $\mathcal{A}$ is always *correct*; we only check here if it is *sufficient*. Thus we only use already-proven facts about the program to reduce it. We could equally well take this information from a separate program analysis.

Theorem 7.4 (Soundness)**.** *If* `CheckProof`$(q_{init}, \mathit{pre}, \emptyset)$ *does not find a counterexample, the program is correct, i.e., P satisfies the pre/postcondition-pair* $(\mathit{pre}, \mathit{post})$.

In addition to soundness, we are interested in the efficiency of our proof check. We can show that for certain preference orders (as before), the proof check can in the best case achieve polynomial complexity in the size of the program:

Theorem 7.5 (Efficiency)**.** *For a non-positional thread-uniform mapping $\preccurlyeq$, and under full commutativity, the time required by Algorithm 2 is polynomial in size(P).*

We optimize proof checking across refinement rounds to avoid repeatedly reducing parts of the program that have already been proven correct: If during a proof check a state $\langle q, \varphi, S\rangle$ – with program location q, Floyd/Hoare assertion φ, and sleep set S – cannot reach a counterexample, we mark this state as "useless". In later refinement rounds, we conclude that a state of the form $\langle q, \varphi \wedge \psi, S\rangle$ – the same state with a strengthened assertion $\varphi \wedge \psi$ – is again useless. We prune all outgoing edges of this state, without changing the reduction language. For the soundness of this optimization, we rely on the *monotonicity* of proof-sensitive commutativity: If statements commute under some condition, they also commute under stronger conditions. Thus, as we strengthen the Floyd/Hoare-assertions in each round, we have more and more commutativity, and prune more (never fewer) edges.

The candidate proof may identify sections of the state space as unreachable. For instance, the proof may establish that mutual exclusion holds for two critical regions of threads. In such cases, we can forgo the sequentialization of all (typically non-equivalent) interleavings in which both threads are in the critical region at the same time, because no such interleaving can be executed by the program. Thus, on-the-fly sequentialization may avoid the construction of large parts of the state space. An upfront sequentialization (decoupled from a subsequent verification algorithm) would not be able to do so.

8 Empirical Evaluation

We implemented our approach in a tool called Ultimate GemCutter. GemCutter analyses C programs that use pthread-style primitives to dynamically create and manipulate threads.[5] GemCutter attempts to prove that such a program only uses a bounded number of threads, and if successful, then verifies the bounded-thread program. For programs with unbounded threads, GemCutter may be able to find bugs. It will never unsoundly declare an unbounded-thread program as correct.

To determine commutativity between statements, we combine an efficient check of a sufficient condition – neither statement writes a variable accessed by the other statement; roughly speaking the heap is here represented as a single array variable – with a more precise (and proof-sensitive) SMT-based check. In cases where the SMT solver is unable to determine commutativity within a small timeout, we fall back to assume non-commutativity. It is always sound to declare things that do commute as non-commutative. In practice, we rarely observe such cases, typically in connection with C memory management features that are modeled with quantified array formulae in SMT.

We implemented several preference orders[6]: *(1)* a non-positional preference order (called "seq") that approximates sequential composition of threads; *(2)* a positional preference order that approximates lockstep scheduling; and *(3)* a (non-positional) preference order that uses a pseudo-random generator with a fixed seed to order statements. Unless otherwise stated, data for GemCutter refers to a portfolio-aggregation of the best preference order for each benchmark (among *(1)* and *(2)* described above, and three versions of the random

[5] To accommodate such programs (given an upper bound on the number of threads), we adapted Algorithm 1 to deal with dynamic thread management.

[6] Information on how to specify preference orders and other settings for GemCutter is available at https://github.com/ultimate-pa/ultimate/blob/dev/releaseScripts/default/adds/gemcutter/README.md.

	Automizer				GemCutter			
	#	time (s)	mem (GB)	rounds	#	time (s)	mem (GB)	rounds
SV-COMP Benchmarks								
successful	940	35 247	2 091	9 951	960	16 626	528	6 163
- correct	163	3 014	148	1 394	169	2 971	89	1073
- incorrect	777	32 233	1 943	8 557	791	13 655	439	5 090
Weaver Benchmarks								
successful	61	2 602	44	1 140	91	2 518	44	954
- correct	60	2 589	43	1 133	90	2 512	44	949
- incorrect	1	13	1	7	1	7	< 1	5

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

order *(3)* with different seeds). The portfolio terminates as soon as the analysis for any preference order terminates. The state-of-the-art software model checker Ultimate Automizer [18] serves as baseline for comparison. Other than the algorithms discussed in this paper, our implementation uses the same components as Automizer.

For the empirical evaluation of our approach, we are interested in the following research questions:

RQ 1: Can sequentialization make finding proofs easier? We examine *(i)* the number of successfully verified programs; *(ii)* the number of refinement rounds needed to find a proof for a program; and *(iii)* the size of the computed proof, as number of Floyd/Hoare-style assertions.

RQ 2: Can sequentialization make proof checking more efficient? We measure the average time per refinement round; as well as the overall memory consumption.

RQ 3: How does the choice of preference order matter? We compare the discussed measures for instantiations of our approach with different preference orders.

Benchmarks. We evaluated both tools using two sets of benchmarks. The first set contains the benchmarks in the *ConcurrencySafety* category of the Competition on Software Verification (SV-COMP'21) [3], a standard benchmark set in software verification with 203 correct and 847 incorrect programs. We excluded programs that our approach does not target (such as safe programs with an unbounded number of threads). The second set is the benchmark set of the Weaver verifier [15] with 182 correct and 1 incorrect programs. These benchmarks require complex proof arguments and are thus good to stress test GemCutter's ability to find simple proofs.

Results. The evaluation was performed using the benchexec benchmarking tool [5] on a Debian 10.10 machine with an AMD Ryzen Threadripper 3970X 32-Core processor. We set a timeout of 900 s and a memory limit of 8000 MB for each instance. The artifact accompanying this paper [14] contains the detailed evaluation results as well as all components needed to reproduce the results.

The quantile plots in Figure 6 (note the logarithmic y-axis) compare the performance of the tools and clearly illustrate the overall time and memory advantage of GemCutter over Automizer. A point (x, y) denotes that the x-th fastest program to be successfully analysed required y seconds of CPU time, resp. the program with the x-th least memory consumption required y MB of memory. Table 1 presents the data in more detail, and additionally includes the total number of refinement rounds. GemCutter analyses 50 additional programs using significantly fewer resources. The increase of ca. 50 % for the Weaver benchmarks is particularly noteworthy.

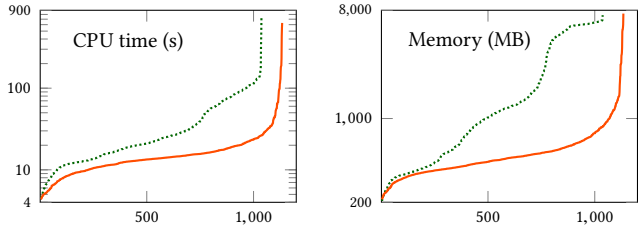


Figure 6. Quantile Plots comparing performance of Automizer (green, dotted) with GemCutter (orange, solid).

Figure 7 illustrates the improvements in proof size and number of refinement rounds for the set of benchmarks that both GemCutter and Automizer can solve (note the logarithmic axes). The number of refinement rounds is reduced by a factor of up to 25 and the proof size by a factor of up to 65. This concludes our investigation of **RQ 1**.

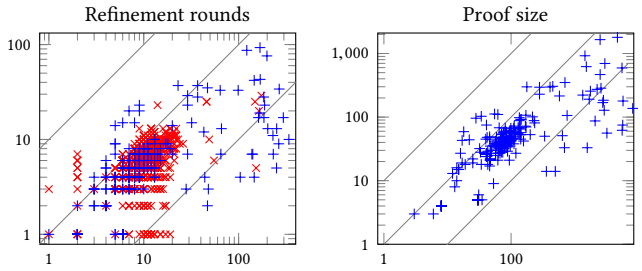


Figure 7. Scatter plots comparing Automizer (x-axis) with GemCutter (y-axis) for correct (+) and incorrect (×) programs

Regarding the efficiency of proof checking (**RQ 2**), we have already presented evidence of overall time and memory improvements. Table 2 shows the average time per refinement round, across all successfully analysed programs, for Automizer, GemCutter (portfolio), variations of GemCutter with only sleep set resp. only persistent set reduction, and GemCutter with both reduction algorithms but only the lockstep preference order. Persistent set reduction makes the most significant contribution to proof check efficiency.

The last column in Table 2 shows that the choice of preference order can significantly impact proof check efficiency (**RQ 3**); orders that do not approximate sequential composition may incur additional costs. However, this does not mean

	Automizer	Portfolio	sleep	persistent	lockstep
Proof size for successfully verified correct programs					
total	147.7	110.9	136.1	155.7	119.6
- SV-COMP	89.0	71.2	71.9	129.4	79.0
- Weaver	307.1	185.3	256.7	215.0	197.1
Time per refinement round (in s) for successfully analysed programs					
total	3.41	2.69	3.04	2.16	3.22
- SV-COMP	3.54	2.70	3.00	2.10	3.21
- Weaver	2.28	2.64	3.22	2.79	3.33

Table 2. Comparison of proof size and proof check efficiency for Automizer vs different variations of GemCutter.

that the preference order *seq* is always the best choice: 18 benchmark programs (16 correct, 2 incorrect) are successfully analysed by the portfolio approach, but not when using only the sequential preference order. Figure 8 illustrates how well different preference orders work for different benchmarks in the set.

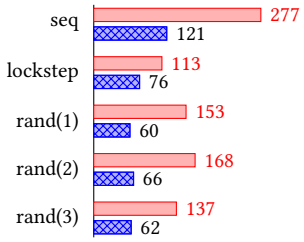


Figure 8. Incorrect (red) and correct (blue, hatched) programs with the best preference order.

We also evaluate the impact of proof-sensitive commutativity. Without proof-sensitivity, 8 fewer programs from the benchmark set can be analysed. The average proof size of successfully proven programs increases (by 2.53% for SV-COMP benchmarks and 4.96% for Weaver benchmarks), as does the total number of refinement rounds for all successfully analysed programs (by 0.80 % for SV-COMP benchmarks and 4.53 % for Weaver benchmarks). The average time per refinement round remains roughly the same. However, we save around 44 GB of memory across all successfully analysed programs.

We attribute the success of GemCutter primarily to the fact that it finds simpler proofs faster for sequentializations. This is reflected in the significant decrease of the number of refinement rounds for correct programs, and the decrease in average proof size. It is noteworthy that GemCutter does better at finding bugs as well; 14 additional incorrect programs are detected in the benchmark set. In these cases, fewer spurious counterexamples from the same equivalence class have to be explored before encountering a real error. This is reflected in the significantly decreased number of refinement iterations needed for incorrect programs.

GemCutter is competitive against bug finding tools. In SV-COMP'22 [4], the state-of-the-art sequentialization-based bounded model checker CSeq [20, 21] won second place in the category *ConcurrencySafety* and was able to find bugs in 303 programs (note that the benchmark set differed from the SV-COMP'21 benchmarks used in this paper). GemCutter also competed [23] and found 299 bugs[7]. In the demo category *NoDataRace*, GemCutter found data races in 69 programs, while CSeq found data races in 61 programs[8]. At the same time, GemCutter soundly verified many benchmark programs, winning third place in *ConcurrencySafety*. CSeq on the other hand is unsound for verification, because it only shows absence of bugs within a certain unrolling bound for loops and for interleavings up to a bounded number of context switches [20].

Limitations. We only evaluate two principled preference orders as well as three randomized orders. Additional (principled) preference orders accompanied by strategies or heuristics to pick a suitable one might yield better results. In particular, it would be interesting to have an approach that can dynamically adjust a choice of a preference order based on partial verification efforts. The fact that the distribution in Figure 8 is relatively even (there is no always-optimal order) suggests that there may be untapped potential here.

Currently, we use the proof to boost commutativity information, but the generation of the proof is unaware of this usage. It is unclear whether the current modest additional benefits of proof-sensitive commutativity can be substantially boosted if proof generation actively tries to help with commutativity. The results from the Siever tool [16] suggest that this merits further investigation.

9 Related Work

Existing research on sequentialization for the analysis of concurrent programs can be roughly classified into two areas, based on whether the aim is to *find bugs* or to *prove correctness*. Sequentialization has been proposed through a transformation of the concurrent program to a sequential program that simulates all its interleavings (e.g. [29]), which does not have any advantage in the form of simplifying proofs or dealing with the state explosion. Using Petri net unfoldings in [10] somewhat improves the state explosion issue, but still requires to prove all interleavings correct. The focus of the discussion in this section is on techniques that do not analyze all program interleavings.

Sequentialization for Bug Finding. Sequentialization for bug finding, introduced by Qadeer and Wu [32], analyzes a subset of all program interleavings chosen by limiting the number of context switches between threads. There is a large body of work on sequentialization for bug finding [20, 21, 26, 27, 33, 34] which looks at different program models (e.g. recursive programs) and considers different bounding and scheduling algorithms for selecting the under-approximation of the program. We forgo an in-depth discussion of these techniques since they cannot verify correctness.

[7] https://sv-comp.sosy-lab.org/2022/results/results-verified/
[8] https://sv-comp.sosy-lab.org/2022/demo.php

Reductions for Semi-Interactive Techniques. Program reductions, although not under this terminology, have been used as a means of simplifying proofs of concurrent and distributed programs before. Lipton's theory of left- resp. right-movers [28] has been used to simplify programs for verification. Elmas et al. [11] introduce an interactive proof technique, where alternating reduction and abstraction steps are used to prove shared memory concurrent programs correct. Kragl and Qadeer [25] build on this and introduce the notion of *layered programs*, a notation that allows a proof engineer to compactly lay out the alternating reduction and abstraction steps. Later work by the same authors [24] builds on this notation and allows further removal of redundant interleavings, up to completely sequential programs.

In the context of message-passing distributed systems, synchronous (rather than sequential) programs are the desired form for reductions: In such reductions, the message buffers are bounded. Preference orders can help identify such reductions, in which the buffers effectively disappear. *Almost-synchronous reductions* [9] and round-based schemes [35] fall in this category.

Reductions for Fully-Automated Techniques. Wachter et al. [36] integrate partial order reduction (POR) in the Impact verification algorithm. They employ monotonic POR [22] on the abstract reachability tree (ART) constructed by Impact, and hence the algorithm ends up verifying a reduction of the program. Additional expansion of the ART is necessary to guarantee soundness, and because of this the computed reduction is not minimal. Wachter et al. employ conditional commutativity, but based on a separate lightweight alias analysis of the ART rather than the proof itself. Chu and Jaffar [8] similarly integrate POR with Impact. They further allow *"property-driven"* commutativity, by identifying certain fixed patterns in the usage of variables and in the checked property.

Cassez and Ziegler [6] combine POR with Trace Abstraction refinement. The reduction implemented in the paper is not minimal. Since the reduction is computed only once up-front before beginning the verification process, it can not take advantage of any information derived from the proof. By contrast, we use proof assertions for conditional commutativity and we compute the reduction on the fly and never construct a full reduction if a feasible error trace is found.

Farzan and Vandikas [15, 16] present a new variant of Trace Abstraction that integrates reduction, trying to find one provable reduction in an infinite class of reductions. Enumeration of all reductions comes at a high complexity cost. They have a guarantee of finding the simplest proof when it matters, but lose on scalability when it does not matter greatly. By contrast, our approach simply chooses one reduction and commits to it, in the hope it can be proven correct. Our aim in this work is to strike a balance between proof simplicity and proof check efficiency. On a more technical level, we give details about the relation between the different classes of minimal reductions in Section 4.

References

[1] 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

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT Press.

[3] Dirk Beyer. 2021. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12652)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 401–422. https://doi.org/10.1007/978-3-030-72013-1_24

[4] 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

[5] 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

[6] 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

[7] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. 2006. Verifying Concurrent Message-Passing C Programs with Recursive Calls. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3920)*, Holger Hermanns and Jens Palsberg (Eds.). Springer, 334–349. https://doi.org/10.1007/11691372_22

[8] 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

[9] Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 709–725. https://doi.org/10.1145/2660193.2660211

[10] Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Mehdi Naouar, Andreas Podelski, and Claus Schätzle. 2021. Verification of Concurrent Programs Using Petri Net Unfoldings. 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, 174–195. https://doi.org/10.1007/978-3-030-67067-2_9

[11] 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

[12] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2014. Proofs that count. 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, 151–164. https://doi.org/10.1145/2535838.2535885

[13] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. *Appendix to: What Is Your Preference Order?* Technical Report. Uploaded as supplementary material to this paper in the ACM Digital Library.

[14] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. *Artifact for PLDI'22 paper "What Is Your Preference Order?"*. https://doi.org/10.5281/zenodo.6409448

[15] 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

[16] 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

[17] 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

[18] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 447–451. https://doi.org/10.1007/978-3-319-89963-3_30

[19] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 69–85. https://doi.org/10.1007/978-3-642-03237-0_7

[20] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 585–602. https://doi.org/10.1007/978-3-319-08867-9_39

[21] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Lazy-CSeq: A Lazy Sequentialization Tool for C - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 398–401. https://doi.org/10.1007/978-3-642-54862-8_29

[22] 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

[23] 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

[24] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive sequentialization of asynchronous programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 227–242. https://doi.org/10.1145/3385412.3385980

[25] 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

[26] Akash Lal and Thomas W. Reps. 2008. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5123)*, Aarti Gupta and Sharad Malik (Eds.). Springer, 37–51. https://doi.org/10.1007/978-3-540-70545-1_7

[27] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. 2008. Interprocedural Analysis of Concurrent Programs Under a Context Bound. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 282–298. https://doi.org/10.1007/978-3-540-78800-3_20

[28] 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

[29] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9938)*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). 174–191. https://doi.org/10.1007/978-3-319-46520-3_12

[30] 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

[31] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. 2007. Spade: Verification of Multithreaded Dynamic and Recursive Programs. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin,*

Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590), Werner Damm and Holger Hermanns (Eds.). Springer, 254–257. https://doi.org/10.1007/978-3-540-73368-3_28

[32] Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William Pugh and Craig Chambers (Eds.). ACM, 14–24. https://doi.org/10.1145/996841.996845

[33] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. 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, 477–492. https://doi.org/10.1007/978-3-642-02658-4_36

[34] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2012. Sequentializing Parameterized Programs. In *Proceedings Fourth Workshop on Foundations of Interface Technologies, FIT 2012, Tallinn, Estonia, 25th March 2012 (EPTCS, Vol. 87)*, Sebastian S. Bauer and Jean-Baptiste Raclet (Eds.). 34–47. https://doi.org/10.4204/EPTCS.87.4

[35] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 59:1–59:30. https://doi.org/10.1145/3290372

[36] 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/