Commutativity Simplifies Proofs of Parameterized Programs

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

Commutativity has proven to be a powerful tool in reasoning about concurrent programs. Recent work has shown that a commutativity-based *reduction* of a program may admit simpler proofs than the program itself. The framework of lexicographical program reductions was introduced to formalize a broad class of reductions which accommodate sequential (thread-local) reasoning as well as synchronous programs. Approaches based on this framework, however, were fundamentally limited to program models with a *fixed/bounded* number of threads. In this paper, we show that it is possible to define an effective parametric family of program reductions that can be used to find simple proofs for *parameterized programs*, i.e., for programs with an unbounded number of threads. We show that reductions are indeed useful for the simplification of proofs for parameterized programs, in a sense that can be made precise: A reduction of a parameterized program may admit a proof which uses *fewer* or *less sophisticated ghost variables*. The reduction may therefore be within reach of an automated verification technique, even when the original parameterized program is not. As our first technical contribution, we introduce a notion of reductions for parameterized programs such that the reduction $\mathcal{R}$ of a parameterized program $\mathcal{P}$ is again a parameterized program (the thread template of $\mathcal{R}$ is obtained by source-to-source transformation of the thread template of $\mathcal{P}$). Consequently, existing techniques for the verification of parameterized programs can be directly applied to $\mathcal{R}$ instead of $\mathcal{P}$. Our second technical contribution is that we define an appropriate family of *pairwise preference orders* which can be effectively used as a parameter to produce different lexicographical reductions. To determine whether this theoretical foundation amounts to a usable solution in practice, we have implemented the approach, based on a recently proposed framework for parameterized program verification. The results of our preliminary experiments on a representative set of examples are encouraging.

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

Additional Key Words and Phrases: commutativity, parameterized programs, constrained Horn clauses

1 INTRODUCTION

The framework of *trace theory* (formulated by Mazurkiewicz in 1987) formalizes equivalence relations for concurrent program runs based on a commutativity relation over the set of atomic steps taken by individual program threads. Two program statements of different threads *commute* if the order in which we execute them is irrelevant to the outcome of the execution. Two program runs are *equivalent* up to commutativity if one can be acquired from another through successive

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

swaps of adjacent commutative program steps. For any program A, we call a program B a *reduction* of A if and only if B includes at least one representative from each (commutativity) equivalence class of behaviours in A. Recent work [Farzan 2023; Farzan et al. 2022; Farzan and Vandikas 2019, 2020] has shown that some reductions of a program admit *simpler* proofs than the program itself. More specific versions of this observation had already been made in the literature of concurrent and distributed program verification. In particular, it is exploited in the context of verification of distributed programs by favouring the verification of synchronous (or almost synchronous) programs in place of asynchronous programs with the rationale that the synchronous program admits a simpler proof [Genest et al. 2007; Kragl and Qadeer 2018; von Gleissenthall et al. 2019].

The common thread in all these contexts is that there is often a lot of redundancy in the set of behaviours of a concurrent program, and removing redundant behaviours with complicated proofs in favour of those with simpler proofs simplifies the entire reasoning task. The choice of a program *reduction*, then, is a choice of which *representatives* from equivalence classes of program behaviours stay and which ones go. Traditionally, people have opted for canonical choices: those that maximize sequential (local) reasoning in the case of concurrent programs [Elmas et al. 2009; Kragl and Qadeer 2018], or those that get as close as possible to a *synchronous* program [Genest et al. 2007; von Gleissenthall et al. 2019] for distributed protocols. As such, each such framework makes an a priori assumption about a particular type of reduction. In recent work, however, a family of *parametric lexicographical program reductions* [Farzan et al. 2022; Farzan and Vandikas 2019, 2020] were introduced that formalized a broad (infinite) class of reductions that would include both canonical choices. The idea is that different program verification tasks may respond best to different strategies for picking representatives. By taking a *lexicographic order* as a parameter to a reduction that chooses the (lexicographically) *least* representative of each equivalence class, one controls the composition of the reduction.

These frameworks, however, were fundamentally built based on an assumption that the alphabet of program actions is *finite*, and therefore, they can only be applied to program models with a *fixed/bounded* number of threads. This brings us to the central research question in this paper: "For programs with unboundedly many threads, is it possible to define an effective parametric family of program reductions that can be exploited for finding simple proofs?" This paper presents an affirmative answer to this question for *parameterized concurrent programs*. A *parameterized program* $\mathcal{P}$ stands for an infinite family of programs $\mathcal{P}(n)_{n \in \mathbb{N}}$. Each program $\mathcal{P}(n)$ arises from taking a number n of threads, where n is not bounded. Each thread runs an instance of the same given thread template. This is without the loss of generality, since well-known encoding tricks [Hoenicke et al. 2017] accommodate the use of multiple thread templates.

It is well-understood, even outside the realm of algorithmic verification, that modular reasoning techniques for parameterized programs (e.g. Owicki-Gries for parameterized programs [Nieto 2001]) are only *complete* in the presence of the full power of *history* variables. Therefore, program proofs may require highly nontrivial ghost variables, which are notoriously hard to compute and reason about automatically. In contrast, in the fixed thread case, the canonical choice of *program counters* is always available and mainly becomes a time/complexity issue for verification algorithms. This paper argues that reductions can help simplify proofs of parameterized concurrent programs, in a sense that can be made precise based on the ghost variables required for the proof. We make the observation that a reduction of a parameterized program may admit a proof which uses *fewer or less sophisticated ghost variables* and may therefore have a higher chance of being within the reach of an automated verification technique.

As a simple example to make this observation concrete, consider the the parameterized program $\mathcal{P}^{\pm}$, given by the thread template in Fig. 1.
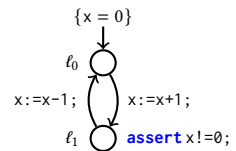

Fig. 1. Template for $\mathcal{P}^{\pm}$

The goal is to prove the property that whenever a thread is in location ℓ_1, the global variable x is non-zero, assuming x is initially 0. It can be shown there does not exist a proof (formally, a proof in the form of an *Ashcroft invariant*) if one does not introduce a ghost variable [Hoenicke et al. 2017]. Intuitively, the proof needs to keep track of the number of threads that have already executed their increment but not yet the matching decrement. Now consider the reduction where the threads are executed sequentially one after the other (sequential composition). The reduction is sound because all statements of two different threads commute (since we do not model the specification **assert** x!=0 as a statement, we are not concerned with its commutativity). The proof for the reduction does not need any ghost variables. We will use this program later as a running example (see Section 4).

Our first technical contribution is a notion of a reduction for a parameterized program. The reduction $\mathcal{R}$ of the parameterized program $\mathcal{P}$ can be viewed as *a family of lexicographical reductions*. This means that $\mathcal{R}$ stands for an infinite family of programs $\mathcal{R}(n)_{n \in \mathbb{N}}$ where for each n, $\mathcal{R}(n)$ is a (lexicographical) reduction of $\mathcal{P}(n)$. Crucially, the infinite family can be *finitely represented*. In fact, the reduction $\mathcal{R}$ is again a parameterized program, and the thread template of $\mathcal{R}$ is obtained by source-to-source transformation of the thread template of $\mathcal{P}$. The key benefit of this observation is that existing techniques for verification of parameterized programs can now be directly applied to $\mathcal{R}$ instead of $\mathcal{P}$.

Reductions that favour program behaviours with long sequential blocks, like the sequential composition for the example in Fig. 1, can be generated using lexicographical reductions based on thread orders; i.e. when statements of each thread are grouped together and ordered wrt. statements of other threads according to their thread identifiers. In Section 2, we present an example that demonstrates why, in the context of parameterized program verification, other reductions like *lockstep* reductions, may be essential if proof simplification is the desired outcome.

Our second technical contribution is that we define an appropriate family of orders, called *pairwise preference orders*, that can be effectively used as a parameter to produce many different lexicographical reductions of the same program given the same commutativity relation (including the above-mentioned lockstep reduction). This generalizes similar results from the literature on how reductions for a fixed number threads are generated parametric on order relations [Farzan et al. 2022; Farzan and Vandikas 2019, 2020]. We show that, as in the case of thread orders, reductions of a parameterized program $\mathcal{P}$ parametric on pairwise preference orders can also be finitely represented as parameterized program $\mathcal{R}$, with the same correspondence between $\mathcal{P}(n)$ and $\mathcal{R}(n)$ for all n.

The two technical contributions outlined so far put forward an algorithmic path for verifying parameterized concurrent programs using a broad family of reductions. To determine whether this amounts to a usable solution in practice, we selected the proof method based on *thread-modular proofs at many levels* [Hoenicke et al. 2017] to instantiate and evaluate this solution. The proof method encodes the existence of a proof of a specific form (an Ashcroft invariant with a number k of universal quantifiers over thread IDs) for an input parameterized program $\mathcal{P}$ as a satisfiability problem of a set of constraints in a specific form (CHC, for Constrained Horn Clauses). To use the proof method for verifying a reduction of the input parameterized program, we apply the proof method to our proposed parameterized reduction, i.e., to the parameterized program $\mathcal{R}$.

We implemented the construction of the parameterized program $\mathcal{R}$ and the constraint generation according to Hoenicke et al. [2017]. We evaluated the approach on a set of 19 parameterized programs taken from the literature, by discharging the generated constraints with several off-the-shelf CHC solvers. The results are very encouraging: The implementation succeeded in verifying the reductions of 14 programs, only 4 of which can be verified without the use of reductions.

It is noteworthy that our proposal for parameterized reductions (and therefore, the corresponding set of CHC constraints) have the desired property that any Ashcroft invariant of the original program

is also a valid invariant for the reduced program. The converse does not hold; i.e., the reduction $\mathcal{R}$ may admit an Ashcroft invariant that is not a valid invariant of the original program $\mathcal{P}$, and a proof in the form of an Ashcroft invariant may not exist for $\mathcal{P}$ even though it does for $\mathcal{R}$.

The property of the *conservative extension* of the validity of an Ashcroft invariant from $\mathcal{R}$ to $\mathcal{P}$ does not, however, mean that we are (in practice) able to compute a proof in the form of an Ashcroft invariant for $\mathcal{R}$ whenever we are able to compute one for $\mathcal{P}$. In fact, the parameterized program uses a set of additional variables as the means of encoding the reduction. It is thus natural to wonder whether the task of the CHC solver could somehow become harder because it has to deal with constraints over a larger set of variables, and, if so, whether anything can be done to alleviate this issue. We investigate this question systematically in Section 6 and propose an alternative encoding with fewer variables. This new encoding is an orthogonal contribution of this paper. It is inspired by the idea of *symmetry reduction* [Clarke et al. 1998]. Intuitively, in the encoding based on Hoenicke et al. [2017], the solver is forced to prove the correctness of symmetry-equivalent classes of reductions. In Section 6.2 we demonstrate how the CHC encoding can be modified so that this redundancy is eliminated.

To conclude, this paper proposes a way of incorporating commutativity-based reductions into, in principle, any existing parameterized verification methodology. In particular, it makes the following contributions:

- We observe that reductions simplify proofs of parameterized programs in a precise sense: Proofs of reductions require less complex ghost state than the proofs of original programs; this can manifest as the need for less complicated information to be recorded in ghost variables, or that simply fewer ghost variables are needed overall (Section 2).
- The theoretical formulation of a parameterized reduction in two parts:
  (1) We formulate a lexicographical reduction of a parameterized program and show that it can be finitely represented, namely again as a parameterized program (Section 4).
  (2) We propose an appropriate notion of preference orders for the parameterized context and show that the construction of a lexicographical reduction from a parameterized program can be made parametric on the preference order (Section 5).
- We give an improved formulation of the search problem for an Ashcroft invariant, by breaking some inherent but redundant symmetries in the search space and the corresponding solution space without affecting soundness or completeness of the methodology (Section 6.2).

2 MOTIVATING EXAMPLE

We demonstrate the benefits of commutativity for proof simplification using the parameterized program $\mathcal{P}^{\text{notify}}$ shown in Fig. 2. This program models a distributed system, in which one thread (called notifier) generates data through some computation (line 6-9), and broadcasts it to an unbounded number of listener threads (line 11-13). The threads communicate via a message queue, which is here modeled via an infinite queue array along with an integer current pointing to the head of the queue (specifically, to the first invalid entry).

Each listener thread joins the conversation by setting its thread-local idx variable to the value of current. The listener then continuously waits for new data to appear in the queue (line 22). When data has arrived, it reads the message from the queue (line 23-24). In the next step, the listener checks the integrity of the received message. In particular, it checks that the received value is greater than the previous message (line 26-28).

Showing correctness of this program is non-trivial; even with ghost variables, a proof is challenging. An unbounded amount of time may pass between the moment when a message is sent by the

```
1  notifier() {                              16  listener() {
2                                            17    idx  := current;
3    last := 0;                              18    prev := 0;
4                                            19
5    while (true) {                          20    while (true) {
6      // generate data                      21      // receive data
7      havoc data;                           22      assume idx < current;
8      assume data > last;                   23      msg := queue[idx];
9      last := data;                         24      idx := idx + 1;
10                                           25
11     // send new data                      26      // check data
12     queue[current] := data;               27      assert prev < msg;
13     current := current + 1;               28      prev := msg;
14   }                                       29    }
15 }                                         30  }
```

Fig. 2. The program $\mathcal{P}^{\text{notify}}$. The variables current (an integer) and queue (an integer array) are global, all other variables are local. An instance $\mathcal{P}^{\text{notify}}(n)$ consists of a single notifier thread and n listener threads.

notifier thread, and when the last listener receives it. Thus, for certain traces, one must keep track of the idx variables of unboundedly many listener threads, not just a finite subset of them.

There exists however a subset of traces, for which the correctness argument is much simpler. Namely, consider those traces where every message sent by the notifier is immediately received and checked by all listeners that have already joined the conversation (i.e., all listeners that will *ever* receive the message). Let us call these traces *synchronous*. In synchronous traces, the difficulty of reasoning about an unbounded number of messages already sent but not yet received by some listener completely disappears. At any point, there is at most one such message, and consequently, the proof has to reason only about one message.

Of course, synchronous traces make up only a small fragment of the many interleavings of the program. To show correctness of the program, we must establish that every trace is correct. Here, commutativity comes to the rescue: We observe that for many statements of the program, the order in which they are executed does not affect the outcome. We say that such statements *commute* with each other. We exploit this observation by repeatedly swapping commuting statements, and thereby reorder any arbitrary trace of the program to an *equivalent* synchronous trace. Through a meta-argument (i.e., the soundness theorem of our approach), we establish that any trace that is equivalent to a correct synchronous trace must itself be correct. Thus, it suffices for a proof to show correctness of synchronous traces, in order to conclude that the program is correct.

Consider for instance the statements `last:=data` (line 9) and `prev:=msg` (line 28). Executing these statements in either order yields the same result, i.e., the statements commute with each other. Similarly, we can argue that all statements of the notifier thread commute with the statement `prev:=msg`. Therefore, we consider for instance the following traces to be equivalent:

	havoc data	data > last	last:=data	queue[current]:=data	current:=current+1	prev:=msg
∼	havoc data	data > last	last:=data	queue[current]:=data	prev:=msg	current:=current+1
∼	havoc data	data > last	last:=data	prev:=msg	queue[current]:=data	current:=current+1
∼	havoc data	data > last	prev:=msg	last:=data	queue[current]:=data	current:=current+1
∼	havoc data	prev:=msg	data > last	last:=data	queue[current]:=data	current:=current+1
∼	prev:=msg	havoc data	data > last	last:=data	queue[current]:=data	current:=current+1

These equivalences allows us reorder entire iterations of the `notifier` thread, i.e., the computation and broadcast of new data, wrt. the statement `prev:=msg`. We proceed similarly with respect to the other statements of the `listener` thread, as well as for statements of different `listener` threads.

For some of these other statements, we must consider broader notions of commutativity. As an example, we cannot generally claim that the order in which the statements `queue[current]:=data` and `msg:=queue[idx]` are executed does not affect the outcome. Specifically, if we have `current = idx`, the order is in fact crucial. However, observe that the program ensures that, whenever the statement `msg:=queue[idx]` is executed, it actually holds that `idx < current`. In such contexts, the order in which the statements are executed is indeed irrelevant. Hence we can say that the statements commute *within this particular program*.

The essential insight of commutativity reasoning is this: It suffices for a proof to cover a so-called *reduction* of a program, i.e., a subset of traces such that each program trace is equivalent to a trace in the reduction. In our example, the reduction is formed by the set of synchronous traces. By soundness of commutativity, we can conclude that, if the reduction is proven correct, the entire program must be correct. In this manner, our approach can verify the program $\mathcal{P}^{\text{notify}}$ by giving a proof for synchronous traces. As discussed, a proof for the set of synchronous traces is much simpler than a proof for all traces, as it does not require complex ghost state or quantified invariants.

As another example where commutativity simplifies the proof, let us consider the program $\mathcal{P}^{K\pm}$, with the thread template shown in Fig. 3. The program has a global variable x, which is initially 0. The program uses a constant K for which we assume a fixed value. Each thread repeatedly checks if the current value of x is less than K, and if so, increments x. It asserts that x is non-zero, eventually decrements x again, and begins the loop anew.

This program is similar to the example discussed in the introduction, yet due to the guard using the constant K, the proof is in some sense simpler: The value of a ghost variable counting the number of threads in location ℓ_1 can never exceed K. Thus, we can alternatively consider the local state of K other threads as ghost state. Specifically, if a thread is in location ℓ_1, and some number m (with $0 \leq m \leq K$) of the K other threads are also in location ℓ_1, we know that $x \geq m + 1$, and therefore, decrementing x does not violate the assert statement in any thread: Either we have $m > 0$, in which case x is still positive after the decrement, or $m = 0$, in which case none of the threads is in location ℓ_1.

Fig. 3. Template for $\mathcal{P}^{K\pm}$

It has been shown that for any value of K, a proof does indeed need to consider at least K additional threads as ghost state (overall considering $K + 1$ threads at a time) to show correctness of this program [Hoenicke et al. 2017]. However, commutativity simplifies the required ghost state.

Let us investigate the commutativity in $\mathcal{P}^{K\pm}$. Two statements `x:=x-1` and `x:=x-1` of different threads commute, as do two statements `assume x<K; x:=x+1` and `assume x<K; x:=x+1` of different threads. For the statements `x:=x-1` and `assume x<K; x:=x+1`, the order of execution may indeed matter. But whenever it is possible to execute the sequence `assume x<K; x:=x+1` `x:=x-1`, it is also possible to execute the sequence `x:=x-1` `assume x<K; x:=x+1` with the same effect (x is not modified), i.e., the latter sequence allows a strict superset of executions. Thus we can verify traces containing the sequence `x:=x-1` `assume x<K; x:=x+1` and conclude that traces containing the sequence `assume x<K; x:=x+1` `x:=x-1` are also correct.

Analogously to the example in Fig. 1, we exploit this commutativity (or *semi-commutativity*) to reorder any trace of the program such that all statements of a thread are executed in a single block. For the resulting reduction of the program, it is sufficient to consider the local state of a single additional thread as ghost state, rather than K threads. If a thread is in ℓ_1, and the other thread (which serves as ghost state) is also in ℓ_1, we know that $x \geq 2$, so x remains positive
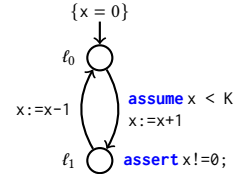
after a decrement. If the "ghost thread" is not in ℓ_1, neither thread executes the assert statement. Commutativity has again simplified the ghost state required to prove correctness of the program.

3 PARAMETERIZED CONCURRENT PROGRAMS

A parameterized program $\mathcal{P}$ is given by its thread template (a control flow graph) and a set of thread-local variables, i.e., $\mathcal{P} = \langle \textbf{Loc}, \Delta, \ell_{\text{init}}, \textbf{Var}_{\text{local}} \rangle$ with a finite set of locations $\textbf{Loc}$, a finite transition relation $\Delta \subseteq \textbf{Loc} \times \textbf{Stmt} \times \textbf{Loc}$ (where $\textbf{Stmt}$ is the set of atomic program statements), an initial location $\ell_{\text{init}} \in \textbf{Loc}$, and a set of thread-local variables $\textbf{Var}_{\text{local}}$. Any variable not in $\textbf{Var}_{\text{local}}$ is considered global. We denote the set of global variables as $\textbf{Var}_{\text{global}}$.

The enabled statements $enabled(\ell)$ of a location ℓ are the statements $\textit{st}$ such that $\langle \ell, \textit{st}, \ell' \rangle \in \Delta$ for some ℓ'. We assume that the only case in which $enabled(\ell)$ contains more than one statement is the case of a branch (or loop head), and thus $enabled(\ell) = \{\textbf{assume}\ e, \textbf{assume}\ \neg e\}$ for some branching condition (or loop guard) e. This assumption is only required for the minimality of our reduction (Proposition 4.10); the soundness of our approach does not rely on it.

A parameterized program describes a family of programs. For each number of threads $n \in \mathbb{N}$, the instance of the program with n threads is denoted by $\mathcal{P}(n)$. The variables of the program instance $\mathcal{P}(n)$ consist of the global variables, as well as indexed local variables x_i for each $i \in \{1, \dots, n\}$ and $x \in \textbf{Var}_{\text{local}}$. The program instance $\mathcal{P}(n)$ uses $\textit{indexed statements}$ $\textit{st}{:}i$, where $\textit{st} \in \textbf{Stmt}$ is a statement as it appears in the thread template, and the thread index $i \in \{1, \dots, n\}$ indicates which thread executes the statement.

Traces. A thread template defines a languages L over the alphabet $\textbf{Stmt}$, consisting of all sequences of statements that label any path from the initial location (regardless of the reached location).

The language of an instance $\mathcal{P}(n)$ of the parameterized program $\mathcal{P}$ is a language of *traces*, i.e., sequences of *indexed* statements. For the language L defined by the thread template of $\mathcal{P}$, let $L[i]$ be the language L where every statement $\textit{st}$ has been replaced by the indexed statement $\textit{st}{:}i$. The program instance $\mathcal{P}(n)$ then defines the language of all traces allowed by the control flow of $\mathcal{P}$:

$$\mathcal{P}(n) = L[1] \parallel \dots \parallel L[n],$$

where $\parallel$ denotes the shuffle operation on languages.

Semantics. We assume that each statement $\textit{st} \in \textbf{Stmt}$ has an associated semantics $[\![\textit{st}]\!]$, given by a binary input/output relation between valuations of the program variables. In particular, the semantics of assignment statements $x\,{:}{=}e$ and assume statements $\textbf{assume}\ e$ is as one would expect.

We extend this semantics to indexed statements. Executing the indexed statement $\textit{st}{:}i$ may modify the global variables as well as the indexed local variables x_i, but leaves local variables of other threads unmodified. Formally, we define the semantics of an indexed statement as follows:

$$[\![\textit{st}{:}i]\!] := \big\{ (s_1, s_2) \mid (s_1|_i, s_2|_i) \in [\![\textit{st}]\!] \land \forall x \in \textbf{Var}_{\text{local}} . \forall j \neq i . s_2(x_j) = s_1(x_j) \big\},$$

where s_1, s_2 are valuations of the variables of $\mathcal{P}(n)$, and $s|_i$ is the unique valuation of the program variables such that $s|_i(x) = s(x_i)$ for local variables x and $s|_i(g) = s(g)$ for global variables g.

Based on these semantics of atomic statements, we define the semantics of each program instance. A *configuration* of $\mathcal{P}(n)$ is a pair $\langle \vec{\ell}, s \rangle$, where $\vec{\ell} = \langle \ell_1, \dots, \ell_n \rangle \in \textbf{Loc}^n$ denotes the control locations of the running threads, and s is a valuation of the variables of the program instance $\mathcal{P}(n)$. We say that the configuration $\langle \vec{\ell}, s \rangle$ is *initial* if $\vec{\ell} = \langle \ell_{\text{init}}, \dots, \ell_{\text{init}} \rangle$.

Let $\langle \vec{\ell}, s \rangle$ be a configuration, such that $\langle \ell_i, \textit{st}, \ell'_i \rangle \in \Delta$ is a transition of the thread template, and such that there is a successor valuation s' with $(s, s') \in [\![\textit{st}{:}i]\!]$. From this configuration, the program can execute $\textit{st}{:}i$. Thread i moves to control location ℓ'_i, whereas all other threads remain at the same location ($\ell'_j = \ell_j$ for all $j \neq i$). We write $\langle \vec{\ell}, s \rangle \xrightarrow{\textit{st}{:}i} \langle \vec{\ell'}, s' \rangle$. A trace $\tau = \textit{st}_1{:}i_1 \ \dots \ \textit{st}_m{:}i_m$ is *feasible*

if there exists a corresponding sequence of configurations (called an *execution*) $\langle \vec{\ell}^{(1)}, s_1 \rangle \xrightarrow{\mathit{st}_1 : i_1}$ $\ldots \xrightarrow{\mathit{st}_n : i_n} \langle \vec{\ell}^{(m)}, s_m \rangle$, and $\langle \vec{\ell}^{(1)}, s_1 \rangle$ is initial. If a trace is not feasible, it is *infeasible*.

Synchronous Statements. Our approach uses a particular kind of statements, so-called *synchronous statements* [Hoenicke et al. 2017]. A thread executes a synchronous statement to (atomically) update the local variables of all (unboundedly many) other threads. Synchronous statements have the form

$$\textbf{for } j \neq i : x_j := e$$

where j and i are symbolic indices representing the threads whose variables are updated (j) and the thread that executes the statement (i). The updated variable x must be a local variable ($x \in \textbf{Var}_{\text{local}}$). The expression e may refer to global variables, as well as local variables y_i, y_j indexed by i or j. Additionally, we allow e to refer to special variables pc_i and pc_j, which represent the current control locations of thread i resp. j.

Correctness and Proofs. A specification for a parameterized program $\mathcal{P}$ consists of a precondition *pre*, and a partial map *assert* from program locations to formulae over the program variables. Both the precondition *pre* and an assertion *assert*(ℓ) may refer to global and local variables. The program $\mathcal{P}$ satisfies the specification $\langle pre, assert \rangle$ if for all numbers of threads n, the following holds: For every execution $\langle \vec{\ell}^{(1)}, s_1 \rangle \xrightarrow{\mathit{st}_1 : i_1} \ldots \xrightarrow{\mathit{st}_n : i_n} \langle \vec{\ell}^{(m)}, s_m \rangle$ of the program instance $\mathcal{P}(n)$, such that $s_1|_i \models pre$ for all $i \in \{1, \ldots, n\}$ and such that *assert*($\ell_j^{(m)}$) is defined for some $j \in \{1, \ldots, n\}$, we have that $s_m|_j \models assert(\ell_j^{(m)})$. In the remainder of the paper, we always assume that a parameterized program is accompanied by a specification $\langle pre, assert \rangle$. For examples, we annotate the specification in the thread template (as in Fig. 1). We simply say that $\mathcal{P}$ *is correct* if $\mathcal{P}$ satisfies this specification.

As an aside, our approach can be extended to more general notions of (safety) specifications, e.g. a set of error states given by a *generator set* as in [Hoenicke et al. 2017]. Such specifications allow for instance a direct encoding of mutual exclusion. However, since this is orthogonal to our contributions, we focus here on the simpler notion of specification as defined above.

In Section 6, as well as several examples, we consider a particular notion of *proofs* for parameterized programs: *Ashcroft invariants*. An Ashcroft invariant is a formula of the form

$$\forall i_1, \ldots, i_k . (\bigwedge_{1 \leq r < s \leq n} i_r \neq i_s) \rightarrow \varphi$$

where φ is a quantifier-free formula, whose variables range over the global program variables, indexed local variables x_{i_r} (for $x \in \textbf{Var}_{\text{local}}, r \in \{1, \ldots, k\}$) and variables pc_{i_r} (for $r \in \{1, \ldots, k\}$) representing the current control location of thread i_r. The quantified variables symbolically represent k threads of the program. The premise $\bigwedge_{1 \leq r < s \leq n} i_r \neq i_s$ expresses the fact that $i_1, \ldots, i_k$ indeed refer to k distinct threads. Thus, the conclusion φ expresses a relation between the global variables, as well as the locations and local variables of any subset of k distinct threads of the program. We call the number of quantified variables k the *width* of the Ashcroft invariant.

An Ashcroft invariant is *inductive* for the parameterized program $\mathcal{P}$, if it is an inductive invariant for every instance $\mathcal{P}(n)$, assuming the precondition *pre* initially holds for every thread. Since we only consider inductive Ashcroft invariants, we omit the adjective from now on.

Finally, let us define what it means for an Ashcroft invariant to prove correctness of a parameterized program $\mathcal{P}$. We say that an Ashcroft invariant $\forall i_1, \ldots, i_k . (\bigwedge_{r \neq s} i_r \neq i_s) \rightarrow \varphi$ is *safe*, if it is inductive, and for every location ℓ where *assert*(ℓ) is defined, the following entailment holds:

$$\forall i_1, \ldots, i_k . (\bigwedge_{1 \leq r < s \leq n} i_r \neq i_s) \rightarrow \varphi \models \forall i . pc_i = \ell \rightarrow assert(\ell)$$

If a safe Ashcroft invariant for a program $\mathcal{P}$ and a specification $\langle pre, assert \rangle$ exists, then $\mathcal{P}$ satisfies the specification $\langle pre, assert \rangle$. However, the reverse is not true.

Other Program Models. Parameterized programs are a natural model for certain classes of concurrent programs, e.g. GPU code and distributed protocols. More generally, most classes of concurrent programs can be *encoded* in parameterized programs. Hence our theoretical results can be expected to hold for a wide class of concurrent programs. In practical terms, such encodings may present a challenge for verification algorithms. For example, for structured parallel programs with sophisticated dependence graphs implemented using fork/join, the best practice would not be to encode the program in this model and try to verify it with our verification algorithm. The main burden in these cases is that the inductive invariant for the program may have to recover part or all of the structure lost from the original model, and this can be unreasonable to expect from an automated invariant generator. Smaller extensions of the model, such as allowing a finite number of different thread templates, as in Fig. 2, are more straightforward and are indeed supported by our implementation.

4 REDUCTIONS OF PARAMETERIZED PROGRAMS

In this section, we discuss commutativity-based reductions. We introduce the underlying formalism, which has previously been used for fixed-thread programs, and discuss how it generalizes to parameterized programs. Then we present our first key contribution: a finite representation of an infinite family of commutativity-based reductions.

To begin, let us quickly summarize the basics of commutativity theory. The most fundamental notion is a *commutativity relation* between statements. Specifically, in this work we say that two (indexed) statements $st_1 : i$ and $st_2 : j$ (with $i \neq j$) *commute*, denoted $st_1 : i \mathbin{\circlearrowright} st_2 : j$, if executing them in either order yields the same semantics, i.e., $[\![st_1 : i \; st_2 : j]\!] = [\![st_2 : j \; st_1 : i]\!]$. We discuss broader notions of sound commutativity in Section 7.

The commutativity relation over statements defines an equivalence relation on traces. We say that two traces τ_1 and τ_2 are *equivalent* if τ_2 can be derived from τ_1 by repeatedly swapping adjacent commuting statements. Note that, by repeated application of the definition of commutativity, equivalent traces have the same semantics. Consequently, it suffices to show that one trace satisfies a specification in order to conclude that all equivalent traces are correct as well.

Motivated by this observation, one can introduce the concept of a *reduction*. A set of traces L' is a *reduction* of another set of traces L if $L' \subseteq L$, and for each trace in L there exists an equivalent trace in L'. It follows that if we prove that all traces in a reduction L' are correct, we can soundly conclude that all traces in the set L are correct. Specifically, we are interested in reductions of the language of traces given by a program instance $\mathcal{P}(n)$ for a fixed number of threads n.

4.1 A Family of Reductions

It has been shown that commutativity-based reduction can lead to simpler proofs for concurrent programs with a fixed number of threads. In particular, the proof for a (suitably chosen) reduction of a program may be within reach of algorithmic verification, even when a proof for the entire program is not.

Example 4.1. Let us consider the program $\mathcal{P}^{\pm}$ as discussed in the introduction, with the template shown in Fig. 1. For any fixed number of threads n, the instance $\mathcal{P}^{\pm}(n)$ is correct. In this case, the proof for the (unreduced) program is comparatively simple: The instance $\mathcal{P}^{\pm}(n)$ can be proven correct with the assertions $x \geq 0$, $x \geq 1$, ..., up to $x \geq n$. Note however that the proof size, i.e., the required number of assertions, grows with the number of threads.

Since the increment and decrement of x commute, as do two increments resp. two decrements, we can apply commutativity to simplify the proof. We define, for each number of threads n, a reduction $\mathcal{R}^\pm(n)$: a set of traces that contains, for each equivalence class of traces in $\mathcal{P}^\pm(n)$, the representative trace in which each thread executes all its statements in the trace in a single block. Thus $\mathcal{R}^\pm(n)$ can be written as $\mathcal{R}^\pm(n) = L_1 L_2 \ldots L_n$, where $L_i = \left(\boxed{\text{x:=x+1:}i} \; \boxed{\text{x:=x-1:}i} \right)^* \left(\varepsilon + \boxed{\text{x:=x+1:}i} \right)$. In traces of this reduction, the value of x reaches a value ≥ 2 only if the last statement executed by some thread i is an increment without a matching decrement. In this case, x never falls below 2 again, as every future decrement is preceded by a matching increment. The reduction $\mathcal{R}^\pm(n)$ can be proven correct with only the assertions $x \geq 0$, $x \geq 1$, $x \geq 2$, for any number of threads n.

In this work, we are concerned with proof simplification for *parameterized* concurrent programs, with an unbounded number of threads. Thus, we are searching for one uniform proof that proves a program $\mathcal{P}$ correct for all numbers of threads n. A key insight is that commutativity can similarly lead to proof simplification in this setting.

Specifically, suppose that for each n, we have proven correctness of a reduction $\mathcal{R}(n)$ of the program instance $\mathcal{P}(n)$ with n threads. Then, by soundness of commutativity for a fixed number of threads, we can conclude that each $\mathcal{P}(n)$ is correct, i.e., the parameterized program $\mathcal{P}$ is correct. Furthermore, if the proofs of reductions for different n have a similar structure, we can hope to find one uniform, finite proof for the parameterized program $\mathcal{P}$.

Example 4.2 (continued from Example 4.1). Let us consider again the program $\mathcal{P}^\pm$, and the claim that each reduction $\mathcal{R}^\pm(n)$ can be proven correct with the assertions $x \geq 0$, $x \geq 1$ and $x \geq 2$. Specifically, each trace in the reduction $\mathcal{R}^\pm(n)$ can be given a correctness proof (an annotation of the trace) using the following Hoare triples, instantiated for all $i \in \{1, \ldots, n\}$:

$$\{x \geq 0\} \; \boxed{\text{x:=x+1:}i} \; \{x \geq 1\}, \qquad \{x \geq 1\} \; \boxed{\text{x:=x+1:}i} \; \{x \geq 2\},$$

$$\{x \geq 2\} \; \boxed{\text{x:=x+1:}i} \; \{x \geq 2\}, \qquad \{x \geq 1\} \; \boxed{\text{x:=x-1:}i} \; \{x \geq 0\}$$

The proof simplification is significant: Without reduction, a proof of the program $\mathcal{P}^\pm$ requires a ghost variable that counts the number of threads that have incremented but not yet decremented x.

Up to this point, the basis for our considerations has been an infinite family of reductions $\mathcal{R}(n)_{n \in \mathbb{N}}$. In order to arrive at an effective proof method for parameterized programs, one crucial step is missing: We need a way to effectively construct a finite representation of this family.

4.2 Parameterized Reductions

The key insight behind our first contribution is this:

OBSERVATION 4.3. *For every parameterized program $\mathcal{P}$, there exists an infinite family of reductions $\mathcal{R}(n)_{n \in \mathbb{N}}$ such that the entire family can again be represented as a parameterized program.*

Representing a family of reductions as a parameterized program enables us to reuse the many mature existing methods for verification of parameterized programs, and to combine them with commutativity-based reduction.

For a fixed n, a finite automaton recognizing a reduction $\mathcal{R}(n)$ can be constructed using the concept of *sleep sets* [Farzan et al. 2022]: In addition to the control locations of the threads, the *sleep set automaton* tracks a set of (indexed) program statements, the eponymous sleep set. In each state, the sleep set automaton prevents transitions labeled by statements in the state's sleep set. After each transition, the sleep set is updated, i.e., statements are removed and added depending on their commutativity with the statement labeling the transition. Consider the illustration of an automaton for $\mathcal{R}(2)$ in Fig. 4. Initially, the sleep set is empty. When traversing the edge labeled $\boxed{st_2\text{:}2}$, we add $\boxed{st_1\text{:}1}$ to the

sleep set, because it has a smaller thread index than $s_2{:}2$, and because we assume in this example that $s_1{:}1$ and $s_2{:}2$ commute. In the next state, as $s_1{:}1$ is in the sleep set, we prune the corresponding edge. Any trace that would be accepted via this edge is equivalent to a trace where $s_2{:}2$ and $s_1{:}1$ are swapped, and this trace is already accepted by a run via the left-most $s_1{:}1$-transition. If we assume that $s_1{:}1$ also commutes with $s_3{:}2$, we can keep $s_1{:}1$ in the sleep set after traversing $s_3{:}2$, and again prune the corresponding edge in the right-most state. If on the other hand $s_1{:}1$ and $s_3{:}2$ did not commute, we would instead remove $s_1{:}1$ from the sleep set and preserve the transition.

Fig. 4. Illustration of the sleep set mechanism. Sleep sets (blue) are updated after every edge, and lead to removal of transitions.

The sleep set technique as explained here can be applied for any fixed number of threads n. However, each n yields a different language, and this approach does not lead to a uniform representation for the family of reductions. The key insight which enables such a uniform finite representation is the observation that for the correctness, we are only interested in *feasible* traces. Thus, we can encode the family of reductions through an *instrumentation* of the original program's thread template. For each thread instance i, we add a boolean variable $sleep_i$, which keeps track of whether thread i (resp. its currently enabled statements) are in the sleep set. Consequently, when $sleep_i$ is true, thread i must not make a move. In other words, any trace where thread i makes a move while $sleep_i$ is true must be *infeasible*. By shifting from an *explicit* mechanism (computing sleep sets, and removing edges from an automaton) to a *symbolic* approach, we thus arrive at a uniform finite representation of the family of reductions.

This instrumentation deviates slightly from the explanation above. Instead of tracking statements in the sleep set, we track the threads that would execute these statements. I.e., the variable $sleep_i$ of a thread i is true, if the thread's next enabled statements $enabled(\ell_i)$ are in the sleep set. (In case multiple statements are enabled, i.e., at a branch or loop head with enabled statements **assume** e and **assume** $\neg e$, either both statements are in the sleep set, or neither is.) This shift from tracking individual letters in the sleep set to tracking the threads pays off in terms of the complexity added to the state space of the instrumented program: We need only a single boolean variable, rather than one variable for each statement that appears in the thread template.

We define a formula which expresses that thread j is in a control location whose enabled statements commute with a given statement $s{:}i$ executed by a different thread i.

Definition 4.4 (Commutativity Test). The *commutativity test* $\mathrm{comm}(j, s{:}i)$ is the formula

$$\mathrm{comm}(j, s{:}i) :\equiv \bigvee \{\, pc_j = \ell \mid \ell \in \mathbf{Loc} \land \forall s' \in enabled(\ell) \,.\, s'{:}j \mathbin{\rotatebox{45}{$\circlearrowright$}} s{:}i \,\}.$$

The commutativity test is used in the instrumentation of statements.

Definition 4.5 (Instrumented Statements). Let s be a statement. We define the instrumented statement $\iota(s)$ as the atomically executed block of statements

$$\iota(s) := \left[\begin{array}{l} \textbf{assume} \, \neg sleep \\ \textbf{for} \, j \neq i : sleep_j := (sleep_j \lor id_j < id_i) \land \mathrm{comm}(j, s{:}i) \\ s \end{array} \right]$$

An instrumented statement $\iota(s)$ first checks if its thread is in the sleep set, and if so, blocks. Otherwise, i.e., if the statement is allowed to execute, the instrumentation performs the update of the sleep set through a *synchronized* statement [Hoenicke et al. 2017] that modifies the *sleep*
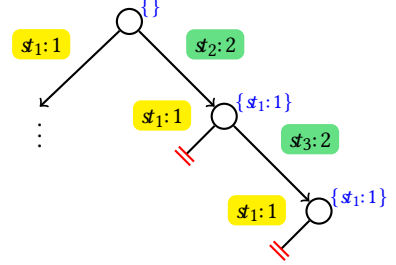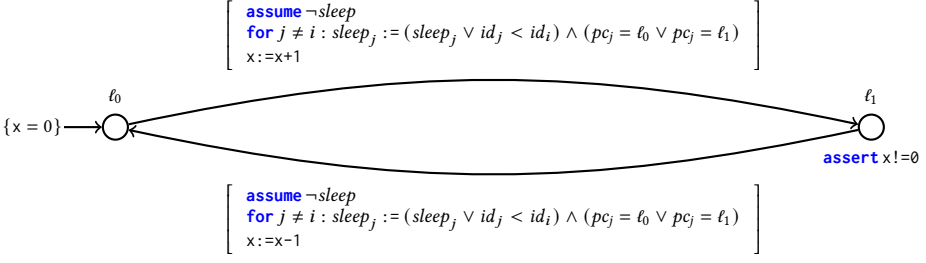
Fig. 5. Template for $\mathcal{P}^{\pm}_{sleep}$

variables of all (unboundedly many) other threads j. Finally, the original statement st executes. Recall that the symbols i and j are part of the syntax of synchronized statements rather than logical variables: i represents the thread executing the statement, and j represents any other thread.

Note that the instrumentation refers to a thread-local integer variable id. We add such a (nondeterministically initialized) ID variable to the thread template to serve as a tie-break. If two threads can move, but allowing both to execute statements would result in equivalent traces, we must identify which thread should go first. We assume that all thread IDs are pairwise distinct. Thus, these thread IDs allow us to distinguish the thread instances and to decide: If the enabled statements of two threads commute, the thread with a smaller ID moves first.

Definition 4.6 (Sleep-Instrumented Program). Let $\mathcal{P} = \langle \mathbf{Loc}, \Delta, \ell_{\text{init}}, \mathbf{Var}_{\text{local}} \rangle$ be a parameterized program. We define the *sleep-instrumented* program $\mathcal{P}_{sleep} := \langle \mathbf{Loc}, \Delta_{sleep}, \ell_{\text{init}}, \mathbf{Var}'_{\text{local}} \rangle$ with local variables $\mathbf{Var}'_{\text{local}} := \mathbf{Var}_{\text{local}} \cup \{id, sleep\}$, and the transitions given by

$$\Delta_{sleep} := \{ \langle \ell, \iota(\mathit{st}), \ell' \rangle \mid \langle \ell, \mathit{st}, \ell' \rangle \in \Delta \} .$$

Example 4.7 (Continued from Example 4.2). Figure 5 shows the thread template for the sleep-instrumented program $\mathcal{P}^{\pm}_{sleep}$ corresponding to the program $\mathcal{P}^{\pm}$ shown in Fig. 1. Since all statements of $\mathcal{P}^{\pm}$ commute, the commutativity tests comm(j, x:=x+1:i) and comm(j, x:=x-1:i) both resolve to the formula $pc_j = \ell_0 \vee pc_j = \ell_1$.

The sleep-instrumented program $\mathcal{P}_{sleep}$ serves as uniform finite representation for the family of reductions $\mathcal{R}(n)_{n \in \mathbb{N}}$. To formalize this relationship, let π be the inverse of ι, i.e., a mapping between statements such that $\pi(\iota(\mathit{st})) = \mathit{st}$. We extend this mapping to indexed statements, traces, and sets of traces in the natural way. The following key result formally expresses that the sleep-instrumented program describes a family of reductions $\mathcal{R}(n)_{n \in \mathbb{N}}$ of the original program, modulo feasibility.

THEOREM 4.8 (REDUCTION). *Let* **Feas** *be the set of all feasible traces. For each number of threads n, the set of traces $\pi(\mathcal{P}_{sleep}(n) \cap \mathbf{Feas})$ is a reduction of $\mathcal{P}(n) \cap \mathbf{Feas}$, i.e., the feasible traces of $\mathcal{P}(n)$.*

We consider only feasible traces, since the reduction works based on the guards ($\neg sleep$) added to each transition. This is necessary to describe the reduction as a parameterized program: Only when we fix the number of threads n, the sleep guards and updates can be evaluated.

THEOREM 4.9 (SOUNDNESS). *The sleep-instrumented program $\mathcal{P}_{sleep}$ is correct iff $\mathcal{P}$ is correct.*

The reduction achieved by the instrumentation is *minimal*: We retain only one representative per equivalence class, and hence a strict subset cannot be a reduction. This means that we do not unnecessarily burden the verification with the proof of redundant traces; the instrumentation fully realizes the benefit of commutativity.

PROPOSITION 4.10 (MINIMALITY). *For every feasible trace τ of $\mathcal{P}$, the sleep-instrumented program $\mathcal{P}_{sleep}$ has exactly one feasible trace τ' such that $\pi(\tau')$ is equivalent to τ.*

As demonstrated in Section 2, there exist programs such that no proof of the program without non-trivial ghost state exists, but where some reduction of the program has a simple proof. We investigate this phenomenon for the sleep-instrumented program. To make this precise, we fix Ashcroft invariants as our notion of *proof*, and consider a simple example.

Example 4.11 (continued from Example 4.7). Consider again the program $\mathcal{P}^{\pm}$, with the template shown in Fig. 1. There does not exist a safe Ashcroft invariant of any width for the program $\mathcal{P}^{\pm}$ [Hoenicke et al. 2017]. Intuitively, the invariant would have to express the information that the value of the global variable x is always greater than or equal

$$\forall i, j \, . \, i \neq j \rightarrow (x \geq 0$$
$$\wedge (pc_i = \ell_1 \rightarrow x \geq 1)$$
$$\wedge (pc_j = \ell_1 \rightarrow x \geq 1)$$
$$\wedge (pc_i = \ell_1 \wedge pc_j = \ell_1 \rightarrow x \geq 2 \vee (sleep_i \wedge sleep_j))$$
$$\wedge (id_i < id_j \wedge pc_i = \ell_1 \rightarrow sleep_i)$$
$$\wedge (id_i > id_j \wedge pc_i = \ell_1 \rightarrow sleep_j))$$

Fig. 6. Safe Ashcroft invariant for $\mathcal{P}^{\pm}_{sleep}$

to the number of threads that have executed the increment but not yet the decrement. Ashcroft invariants cannot express this information (for a formal argument, see [Hoenicke et al. 2017]).

Figure 6 shows a safe Ashcroft invariant (of width 2) for the sleep-instrumented program $\mathcal{P}^{\pm}_{sleep}$ (shown in Fig. 5). This invariant uses the fact that, in the traces of the reduction, if the value of x exceeds 2, it never falls below 2 again. Any trace of the original program is equivalent to a trace in the reduction, since increments and decrements commute and can be arbitrarily reordered.

Let us examine some traces of $\mathcal{P}^{\pm}_{sleep}(2)$ to see how the Ashcroft invariant proves the correctness of traces in $\mathcal{R}^{\pm}(2)$ as well as outside $\mathcal{R}^{\pm}(2)$ by using the variables $sleep_i$ and $sleep_j$. We assume that $id_1 < id_2$. Given a trace of $\mathcal{P}^{\pm}_{sleep}(2)$, we instantiate $i := 1$ and $j := 2$ in the Ashcroft invariant, insert concrete values for pc_1 and pc_2, and simplify the formula to get an inductive annotation of the trace. For instance, the trace `x:=x+1:1` `x:=x+1:2` `x:=x-1:1` of $\mathcal{P}^{\pm}(2)$ is not included in the reduction $\mathcal{R}^{\pm}(2)$. We get the following annotation for the corresponding instrumented trace:

$\{x \geq 0\}$ `ι(x:=x+1):1` $\{x \geq 1\}$ `ι(x:=x+1):2` $\{x \geq 1 \wedge (x \geq 2 \vee sleep_2) \wedge sleep_1\}$ `ι(x:=x-1):1` $\{x \geq 1\}$

Consider in particular the last Hoare triple. Since $sleep_1$ holds, and `ι(x:=x-1):1` assumes $\neg sleep_1$, the last statement cannot be executed, i.e., the trace is infeasible (hence, any postcondition holds afterwards). By contrast, consider the annotated trace corresponding to `x:=x+1:1` `x:=x+1:2` `x:=x-1:2`:

$\{x \geq 0\}$ `ι(x:=x+1):1` $\{x \geq 1\}$ `ι(x:=x+1):2` $\{x \geq 1 \wedge (x \geq 2 \vee sleep_2) \wedge sleep_1\}$ `ι(x:=x-1):2` $\{x \geq 1\}$

Note again the last Hoare triple. The assumption $\neg sleep_2$ by the statement `ι(x:=x-1):2` together with the precondition ensures that $x \geq 2$, and thus $x \geq 1$ still holds after the decrement. The final postcondition $x \geq 1$ however does not prevent us from extending the trace with the statement `ι(x:=x-1):1`, yielding the postcondition $x \geq 0$. While the resulting trace would not correspond to a trace in the reduction $\mathcal{R}^{\pm}(2)$, the Ashcroft invariant does not prove its infeasibility. Instead, it simply proves that the trace satisfies the specification.

Even in cases where a safe Ashcroft invariant of some width k exists, sleep instrumentation can simplify the proof. Specifically, sleep instrumentation can reduce the minimum width of a safe Ashcroft invariant.

Example 4.12. Consider the program $\mathcal{P}^{K\pm}$ with the template shown in Fig. 3, where x is a global integer variable. Given a fixed value for the constant K, a safe Ashcroft invariant for this program must have at least width $K + 1$ [Hoenicke et al. 2017]. However, the Ashcroft invariant shown in Fig. 6 has width 2, and is safe for the sleep-instrumented program $\mathcal{P}^{K\pm}_{sleep}$, for every value of K.

The following theorem states that, if we already have a proof (i.e., a safe Ashcroft invariant) for the original program $\mathcal{P}$, there also exists a safe Ashcroft invariant for the sleep-instrumented program. Hence, more (and by Example 4.11, *strictly* more) programs can be proven correct with our instrumentation than without. Additionally, the theorem shows that a proof of the sleep-instrumented program $\mathcal{P}_{sleep}$ need never be more complicated than a proof of the original program $\mathcal{P}$; and in cases such as Example 4.12, it may be strictly simpler.

THEOREM 4.13 (CONSERVATIVE EXTENSION). *Every safe Ashcroft invariant for a program $\mathcal{P}$ is a safe Ashcroft invariant for the corresponding sleep-instrumented program $\mathcal{P}_{sleep}$.*

5 REDUCTIONS BEYOND SEQUENTIAL COMPOSITION

Up to this point, we have considered a very restricted class of reductions based on thread ordering: A thread i could only change $sleep_j$ from false to true if $id_j < id_i$. If all statements of different threads commute, the resulting reduction is the sequential composition of threads: As soon as a thread with a higher ID takes a step, all threads with lower ID are "put to sleep" and never awakened again. If not all statements commute, the reduction overapproximates sequential composition.

Recall the program $\mathcal{P}^{notify}$ discussed in Section 2 (shown Fig. 2). For this program, it is crucial to align the "sends" (i.e., writes to the queue array) in the notifier thread with all the "receives" (i.e., reads from the queue array) in the listener threads in order to find a simple proof. The (approximation of) sequential composition would not provide sufficient opportunity for simplification. In this section, we widen our view to consider the larger class of *lexicographical reductions*, which have been shown to be practically useful reductions for program verification [Farzan et al. 2022].

Farzan et al. [2022] use *preference orders* to describe different reductions of fixed-thread programs. A preference order is a total order over program traces (or, more generally, words over some alphabet). It can be used to define a reduction as follows:

Definition 5.1 (Definition 4.2 in [Farzan et al. 2022]). Let L be a language over an alphabet Σ, and let $\leq$ be a total order over Σ^*. The *reduction of L induced by $\leq$* is denoted $red_{\leq}(L)$ and contains, for each equivalence class, only the minimal trace wrt. the preference order.

$$red_{\leq}(L) = \{\, \min_{\leq}[w] \mid w \in L \,\}$$

In this work, we focus on the class of *positional lexicographic preference orders* [Farzan et al. 2022]. Positional lexicographic preference orders are a generalization of a lexicographic orders over program traces, where the underlying order on statements may differ depending on the current program locations of all threads. The reductions induced by positional lexicographic preference orders are called *lexicographical reductions*.

We extend the concept of (positional lexicographic) preference orders to parameterized programs.

Definition 5.2 (Parameterized Preference Order). A *parameterized preference order* is a family of functions $(\preccurlyeq^n)_{n \in \mathbb{N}}$, where $\preccurlyeq^n \colon \mathbf{Loc}^n \to TO_{\{1,\ldots,n\}}$ maps n-tuples of locations $\vec{\ell}$ to total orders $\preccurlyeq^n_{\vec{\ell}}$ over thread indices $\{1, \ldots, n\}$. For $\vec{\ell} \in \mathbf{Loc}^n$, we write $\preccurlyeq_{\vec{\ell}}$ instead of $\preccurlyeq^n_{\vec{\ell}}$ (we omit the superscript n).

A parameterized preference order is thus given by the choice of the underlying ordering of threads (all statements of the same thread are ordered the same). As the threads move to different control locations, the ordering of threads assigned by a parameterized preference order may change. Thus the reduction may differ significantly from the sequential composition of threads.

We focus on a subclass of finitely describable parameterized preference orders:

Definition 5.3 (Pairwise Preference Order). Let $R \subseteq \mathbf{Loc}^2$ be a total and transitive relation. The *pairwise preference order* induced by R is the parameterized preference order $(\preccurlyeq^n)_{n \in \mathbb{N}}$ such that

$$i \preccurlyeq_{\vec{\ell}} j \quad \Longleftrightarrow \quad \langle \ell_i, \ell_j \rangle \in R \land (\langle \ell_j, \ell_i \rangle \in R \to id_i \leq id_j)$$

Thus, the ordering of threads i, j wrt. a pairwise preference order only depends on the control locations ℓ_i, ℓ_j of i and j in the tuple $\vec{\ell}$. The locations of other threads do not play a role. If the pair $\langle \ell_i, \ell_j \rangle$ is in R, and the reversed pair $\langle \ell_j, \ell_i \rangle$ is *not* in R, we prefer thread i. If both the pairs $\langle \ell_i, \ell_j \rangle$ and $\langle \ell_j, \ell_i \rangle$ are in R, we prefer the thread with a smaller ID. By totality of R (i.e., $\langle \ell, \ell' \rangle \in R$ or $\langle \ell', \ell \rangle \in R$ for all $\ell, \ell' \in \mathbf{Loc}$), one of two compared threads must always be preferable over the other.

PROPOSITION 5.4. *Each total, transitive relation $R \subseteq \mathbf{Loc}^2$ defines a parameterized preference order.*

PROOF. We have to show that for every n and $\vec{\ell} \in \mathbf{Loc}^n$, the induced relation $\preccurlyeq_{\vec{\ell}}$ is a total order over the set of thread indices $\{1, \ldots, n\}$.

Reflexivity Follows from totality, which is shown below.

Antisymmetry Let $i \preceq_{\vec{\ell}} j$ and $j \preceq_{\vec{\ell}} i$. It follows that $\langle \ell_i, \ell_j \rangle \in R$, $\langle \ell_j, \ell_i \rangle \in R$, and, by the respective implications, also $id_i \le id_j$ and $id_j \le id_i$. Thus we have $id_i = id_j$, and by uniqueness of thread IDs, we conclude $i = j$.

Transitivity Let $i \preceq_{\vec{\ell}} j \preceq_{\vec{\ell}} k$. Thus we have $\langle \ell_i, \ell_j \rangle \in R$ and $\langle \ell_j, \ell_k \rangle \in R$, and we know that the implications $\langle \ell_j, \ell_i \rangle \in R \rightarrow id_i \le id_j$ and $\langle \ell_k, \ell_j \rangle \in R \rightarrow id_j \le id_k$ hold.

By transitivity of R we know that $\langle \ell_i, \ell_k \rangle \in R$. It remains to show that the implication $\langle \ell_k, \ell_i \rangle \in R \rightarrow id_i \le id_k$ holds. Suppose that $\langle \ell_k, \ell_i \rangle \in R$. By transitivity of R, we have $\langle \ell_k, \ell_j \rangle \in R$ and thus $id_j \le id_k$. Furthermore, again by transitivity, we have $\langle \ell_j, \ell_i \rangle \in R$ and thus $id_i \le id_j$. It follows that indeed $id_i \le id_j \le id_k$.

Totality Let $i, j \in \{1, \ldots, n\}$, and wlog. $id_i \le id_j$. By totality of R, we must have $\langle \ell_i, \ell_j \rangle \in R$ or $\langle \ell_j, \ell_i \rangle \in R$. If $\langle \ell_i, \ell_j \rangle \in R$, we have that $i \preceq_{\vec{\ell}} j$ (the implication $\langle \ell_j, \ell_i \rangle \in R \rightarrow id_i \le id_j$ holds, because the conclusion holds). Otherwise, if $\langle \ell_j, \ell_i \rangle \in R$ but $\langle \ell_i, \ell_j \rangle \notin R$, we have $j \preceq_{\vec{\ell}} i$ (the implication $\langle \ell_i, \ell_j \rangle \in R \rightarrow id_j \le id_i$ holds, because the premise does not hold). □

Section 4 considers the special case that $R = \mathbf{Loc}^2$, such that we have $i \preccurlyeq_{\vec{\ell}} j \iff id_i \le id_j$. The class of pairwise preference orders also includes other interesting orders.

Example 5.5 (Lockstep Order). For each $\ell \in \mathbf{Loc}$, let $d(\ell)$ be the minimum length of a path (in the thread template) from the initial location to ℓ. We define the transitive and total relation $R = \{ \langle \ell, \ell' \rangle \mid d(\ell) \le d(\ell') \}$. The induced pairwise preference order mimics lock-step execution: Whenever a thread i has "fallen behind" a thread j (i.e., $d(\ell_i) < d(\ell_j)$), thread i is preferred over thread j and is thus allowed to "catch up". When the locations of both threads have the same depth, i.e., $\langle \ell_i, \ell_j \rangle \in R$ and $\langle \ell_j, \ell_i \rangle \in R$, the thread with the smaller ID is preferred and takes the next step.

Let us once again consider the program $\mathcal{P}^{\mathrm{notify}}$. For this program, the reduction which admits the simple proof discussed in Section 2 is induced by lockstep order.

The construction of our instrumented program $\mathcal{P}_{sleep}$ can be generalized to arbitrary pairwise preference orders. To this end, and for the remainder of the paper, let $R \subseteq \mathbf{Loc}^2$ be a total and transitive relation. In order to represent the reduction wrt. any pairwise preference order again as a parameterized program, we define:

Definition 5.6 (Preference Test). The *preference test* for the pairwise preference order induced by the total and transitive relation $R \subseteq \mathbf{Loc}^2$ is defined as the following formula:

$$\mathrm{pref}(i, j) :\equiv \langle pc_i, pc_j \rangle \in R \land (\langle pc_j, pc_i \rangle \in R \rightarrow id_i \le id_j)$$

The preference test $\mathrm{pref}(i, j)$ evaluates to true if, in the current program configuration $\langle \vec{\ell}, s \rangle$, our pairwise preference order prefers the statements of thread i over the statements of thread j, i.e., if $i \preceq_{\vec{\ell}} j$. We modify the instrumentation of statements (Definition 4.5) to use the preference test.

Definition 5.7 (Instrumented Statement with Preference Test). Let s be a statement. We define the instrumented statement $\iota(s)$ as the atomically executed block of statements

$$\iota(s) := \left[\begin{array}{l} \textbf{assume } \neg sleep \\ \textbf{for } j \neq i : sleep_j := (sleep_j \vee \text{pref}(j, i)) \wedge \text{comm}(j, s{:}i) \\ s \end{array} \right]$$

Our results in Section 4 (Theorems 4.8, 4.9 and 4.13 and Proposition 4.10) still hold for the modified instrumentation, and for every pairwise preference order.

6 FINDING ASHCROFT INVARIANTS FOR A REDUCTION

We apply the approach of *thread-modular verification at many levels* [Hoenicke et al. 2017] to find proofs of parameterized programs in the form of Ashcroft invariants. We show how this approach can be applied to sleep-instrumented programs to verify reductions of parameterized programs.

6.1 Thread-Modular Verification of Reductions

In thread-modular verification at many levels [Hoenicke et al. 2017], the existence of a safe Ashcroft invariant of some fixed width k for a program $\mathcal{P}$ is encoded through a constrained Horn clause (CHC) system. This CHC system, which we denote by $\mathbb{TM}(\mathcal{P}, k)$, uses a single uninterpreted predicate symbol $Inv(g, pc_1, x_1, \ldots, pc_k, x_k)$. The parameter g represents the global variables of the program. The parameters pc_i and x_i represent the current control locations resp. the thread-local variables of k different thread instances.

We can apply an off-the-shelf CHC solver to check satisfiability of this CHC system. If the system is unsatisfiable, there does not exist a safe Ashcroft invariant of width k. However, this does not mean that the program is incorrect. It might simply be that every safe Ashcroft invariant has a width larger than k, or that there does not exist a safe Ashcroft invariant of any width, yet the program is still correct. If on the other hand the CHC system is satisfiable, we can construct an Ashcroft invariant from a solution.

LEMMA 6.1 (LEMMAS 1 AND 3 IN [HOENICKE ET AL. 2017]). *If* Φ_{Inv} *is a solution of* $\mathbb{TM}(\mathcal{P}, k)$, *then*

$$\forall i_1, \ldots, i_k \, . \, (\bigwedge_{1 \leq r < s \leq n} i_r \neq i_s) \rightarrow \Phi_{Inv}(g, pc_{i_1}, x_{i_1}, \ldots, pc_{i_k}, x_{i_k})$$

is a safe Ashcroft invariant (of width k) for the program $\mathcal{P}$.

We apply the same methodology to the sleep-instrumented program $\mathcal{P}_{sleep}$, yielding the CHC system $\mathbb{TM}(\mathcal{P}_{sleep}, k)$. Since $\mathcal{P}_{sleep}$ is again a parameterized program, no conceptual changes are required. In particular, the thread-modular CHC encoding supports the *synchronized statements* used by our instrumentation to update the sleep variables of all (unboundedly many) other threads [Hoenicke et al. 2017]. Figure 7 shows the resulting CHC encoding for $\mathbb{TM}(\mathcal{P}_{sleep}, k)$. We call the encoding $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ the *symbolic-sleep* encoding, to distinguish it from the *explicit-sleep* encoding introduced in Section 6.2.

Intuitively, the clauses describe an invariant predicate Inv that must hold for any subset of k distinct threads (mirroring the structure of Ashcroft invariants). The clause ***Initial*** establishes that the invariant holds initially. For any k threads $i_1, \ldots, i_k$, the ***Inductivity*** clauses demand that the invariant must be preserved if any of the threads $i_1, \ldots, i_k$ makes a step, whereas ***Non-Interference*** imposes that the invariant is preserved if another thread (denoted $\star$) makes a step. Replacing any of the threads i_r by the thread $\star$ yields another set of k distinct threads, and we may assume that the invariant Inv holds for any of these sets. This yields the additional premises in the ***Non-Interference*** clause. Finally, ***Safety*** ensures that a solution to the CHC system describes a *safe* Ashcroft invariant.

Initial:

$$\left(\bigwedge_{i=1}^{k} pre(g, x_1)\right) \wedge \left(\bigwedge_{i=1}^{k} pc_i = \ell_{\text{init}}\right) \wedge \left(\bigwedge_{i=1}^{k} \neg sleep_i\right) \wedge \left(\bigwedge_{i \neq j} id_i \neq id_j\right) \tag{1}$$
$$\rightarrow Inv(g, id_1, pc_1, sleep_1, x_1, \ldots, id_k, pc_k, sleep_k, x_k)$$

Inductivity (for each edge $\langle \ell, \mathfrak{s}, \ell' \rangle \in \Delta$ and each $i \in \{1, \ldots, k\}$):

$$Inv(g, id_1, pc_1, sleep_1, x_1, \ldots, id_k, pc_k, sleep_k, x_k) \wedge pc_i = \ell \wedge pc_i' = \ell' \wedge \mathfrak{s}(g, x_i, g', x_i') \tag{2}$$
$$\wedge \neg sleep_i \wedge \bigwedge_{j \neq i} \left(sleep_j' \leftrightarrow (sleep_j \vee \text{pref}(j, i)) \wedge \text{comm}(j, \mathfrak{s} : i) \right)$$
$$\rightarrow Inv(g', id_1, pc_1, sleep_1', x_1, \ldots, id_i, pc_i', sleep_i', x_i', \ldots, id_k, pc_k, sleep_k, x_k)$$

Non-Interference (for each edge $\langle \ell, \mathfrak{s}, \ell' \rangle \in \Delta$):

$$Inv(g, id_1, pc_1, sleep_1, x_1, \ldots, id_k, pc_k, sleep_k, x_k) \tag{3}$$
$$\wedge Inv(g, id_\star, pc_\star, sleep_\star, x_\star, \ldots, id_k, pc_k, sleep_k, x_k) \wedge \ldots \wedge Inv(g, id_1, pc_1, sleep_1, x_1, \ldots, id_\star, pc_\star, sleep_\star, x_\star)$$
$$\wedge pc_\star = \ell \wedge pc_\star' = \ell' \wedge \mathfrak{s}(g, x_\star, g', x_\star')$$
$$\wedge \neg sleep_\star \wedge \bigwedge_{j} \left(sleep_j' \leftrightarrow (sleep_j \vee \text{pref}(j, \star)) \wedge \text{comm}(j, \mathfrak{s} : \star) \right)$$
$$\rightarrow Inv(g', id_1, pc_1, sleep_1', x_1, \ldots, id_k, pc_k, sleep_k', x_k)$$

Safety (for each $i \in \{1, \ldots, k\}$ and $\ell \in \mathbf{Loc}$ where $assert(\ell)$ is defined):

$$Inv(g, id_1, pc_1, sleep_1, x_1, \ldots, id_k, pc_k, sleep_k, x_k) \wedge pc_i = \ell \wedge \neg assert(\ell) \rightarrow \bot \tag{4}$$

Fig. 7. *Symbolic-sleep* CHC encoding $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ for the existence of a safe Ashcroft invariant of width k for the sleep-instrumented program $\mathcal{P}_{sleep}$. Differences from the CHC encoding $\mathbb{TM}(\mathcal{P}, k)$ for the unreduced program are highlighted in red.

PROPOSITION 6.2. *If the CHC system $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ is satisfiable, the program $\mathcal{P}$ is correct.*

PROOF. Follows from Lemma 6.1 and Theorem 4.9. □

In analogy to Theorem 4.13, the symbolic-sleep CHC encoding $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ behaves conservatively wrt. the encoding $\mathbb{TM}(\mathcal{P}, k)$ of the original program.

OBSERVATION 6.3. *Any solution to $\mathbb{TM}(\mathcal{P}, k)$ is also a solution to $\mathbb{TM}(\mathcal{P}_{sleep}, k)$. Moreover, there are cases where $\mathbb{TM}(\mathcal{P}, k)$ has no solution, but $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ does.*

6.2 Breaking Symmetry with the Explicit-Sleep Encoding

Despite Observation 6.3, it is not clear that a CHC solver will be faster to find a solution when applied to the symbolic-sleep encoding $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ than when applied to the encoding $\mathbb{TM}(\mathcal{P}, k)$. In order to gain a systematic understanding of how easy or difficult it is for a CHC solver to find a solution, let us introduce the notions of *search* and *solution space*.

Definition 6.4 (Search and Solution Space). Let C be a CHC system over a single predicate symbol $p(v_1, \ldots, v_m)$ of arity m. The search space $\mathbf{Search}(C)$ of possible solutions to C is the set of all first-order formulae $\Phi_p(v_1, \ldots, v_m)$ whose free variables lie in $\{v_1, \ldots, v_m\}$.

The solution space $\mathbf{Sol}(C)$ denotes the subset of the search space $\mathbf{Search}(C)$ containing exactly all those predicates $\Phi_p(v_1, \ldots, v_m)$ that satisfy the given CHC system C.

A larger solution space means that a solver is more likely to find a satisfying solution to a CHC system, whereas a larger search space is indicative of potential additional effort to rule out other predicates. In particular, while sleep instrumentation does somewhat increase the search space (it introduces new variables), it leads to a significantly and *qualitatively* larger solution space: Most importantly, for some programs, $\mathbf{Sol}(\mathbb{TM}(\mathcal{P}, k))$ is empty while $\mathbf{Sol}(\mathbb{TM}(\mathcal{P}_{sleep}, k))$ is not.

Our evaluation (Section 8) shows that for some programs which can be proven without reduction, we observe a notable overhead for the instrumented version, due to the increased search space. To minimize this overhead, we further improve upon the CHC system $\mathbb{TM}(\mathcal{P}_{sleep}, k)$, by *decreasing* the search space. The improvement is quantitative, i.e., does not increase the expressivity of the approach, but rather serves to allow CHC solvers to find a solution faster.

We observe that solutions to the symbolic-sleep encoding $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ typically include redundant information due to symmetry: Since the ordering of threads expressed by the thread IDs is nondeterministic, solutions often need to make case distinctions covering all possible orderings. Intuitively, we force the solver to prove correctness of a symmetry-equivalence class of reductions. To illustrate this, consider again the Ashcroft invariant in Fig. 6. In the last two conjuncts, the invariant makes a case distinction over the ordering of thread IDs. While this is a small example, and the Ashcroft invariant in Fig. 6 is still relatively simple, in general (for larger programs, and larger k), such case distinctions can result in a factorial (in k) number of *symmetric* conjuncts.

In order to avoid paying this additional cost, we take advantage of the symmetry between threads. *Symmetry reductions* [Clarke et al. 1998] have been widely used for parameterized systems to reduce the search space of analyses. The idea behind symmetry reductions is closely connected to our observations: Instead of naïvely enumerating all possible cases of a nondeterministically chosen order, and recovering symmetric results for each case, one focuses on a single fixed order.

In our case, we fix the ordering of the k threads considered by the CHC predicate symbol $Inv(g, id_1, pc_1, x_1, \ldots, id_k, pc_k, x_k)$ such that we always have $id_1 < \ldots < id_k$. This allows us to simplify the CHC system. In particular, we resolve the comparisons of thread IDs in the preference test (Definition 5.6) statically.

Definition 6.5 (Explicit-sleep Preference Test). For $i \in \{1, \ldots, k\}$, $j \in \{1, \ldots, k, \star\}$ and $j' \in \{1, \ldots, k\}$, we define the *explicit-sleep preference test* as the formula

$$\widetilde{\mathrm{pref}}(i, j/j') :\equiv \begin{cases} \langle pc_i, pc_j \rangle \in R & \textbf{if } i \leq j' \\ \langle pc_i, pc_j \rangle \in R \wedge \neg \langle pc_j, pc_i \rangle \notin R & \textbf{otherwise} \end{cases}$$

Recall that R is a total and transitive relation which induces a pairwise preference order (Definition 5.3). We pass two parameters j, j' for the second thread, in order to account for the arbitrary ordering (represented by j') of the interfering thread (represented by $j = \star$) wrt. to the k other threads (represented by i). We could have $id_\star < id_1$ (i.e., $j' = 1$), or $id_1 < id_\star < id_2$ (i.e., $j' = 2$), ..., or $id_k < id_\star$ (i.e., $j' = k + 1$). To cover all cases, we introduce one non-interference clause for each of these $k + 1$ possible orderings. Thanks to this explicit case distinction, all comparisons between thread IDs are resolved statically, and we eliminate the thread IDs from the CHC system.

Furthermore, we must take care to reorder the variables in the second line of Eq. (3), such that we preserve the assumption that the threads to whose variables the predicate symbol Inv is applied are in increasing order of their ID. To this end, we define a permutation: For $i \in \{1, \ldots, k + 1\}$, $r \in \{1, \ldots, k\}$, let $\sigma_i^r : \{1, \ldots, k\} \rightarrow \{1, \ldots, k, \star\} \setminus \{r\}$ be the bijective mapping such that

- for all $j_1 \neq j_2$ with $\sigma_i^r(j_1) \neq \star$ and $\sigma_i^r(j_2) \neq \star$, we have $j_1 < j_2 \iff \sigma_i^r(j_1) < \sigma_i^r(j_2)$, and
- for all $j_1 \neq j_2$ with $\sigma_i^r(j_1) = \star$ and $\sigma_i^r(j_2) \neq \star$, we have $j_1 < j_2 \iff i \leq j_2$.

Intuitively, σ_i^r describes the sequence $1, 2, \ldots, k$, where we insert $\star$ at position i and then delete the number r from the sequence. The index i here represents the preference ordering of the interfering thread $\star$ wrt. the other threads $1, \ldots, k$; index r represents the thread that is replaced by $\star$.

Figure 8 shows the resulting *explicit-sleep* CHC encoding $\mathbb{TM}_{sleep}(\mathcal{P}, k)$, for a program $\mathcal{P}$ and width k. Note that the explicit-sleep encoding receives the original program $\mathcal{P}$ as input; the sleep instrumentation is performed as part of the encoding. Nevertheless, we semantically connect this encoding to the sleep-instrumented program $\mathcal{P}_{sleep}$, in a manner analogous to Lemma 6.1.

Initial:

$$\left(\bigwedge_{i=1}^{k} pre(g, x_1)\right) \wedge \left(\bigwedge_{i=1}^{k} pc_i = \ell_{\text{init}}\right) \wedge \left(\bigwedge_{i=1}^{k} \neg sleep_i\right) \tag{5}$$
$$\rightarrow Inv(g, pc_1, sleep_1, x_1, \ldots, pc_k, sleep_k, x_k)$$

Inductivity (for each edge $\langle \ell, st, \ell' \rangle \in \Delta$ and each $i \in \{1, \ldots, k\}$):

$$Inv(g, pc_1, sleep_1, x_1, \ldots, pc_k, sleep_k, x_k) \wedge pc_i = \ell \wedge pc_i' = \ell' \wedge st(g, x_i, g', x_i') \tag{6}$$
$$\wedge \neg sleep_i \wedge \bigwedge_{j \neq i} \left(sleep_j' \leftrightarrow (sleep_j \vee \widetilde{\text{pref}}(j, i/i)) \wedge \text{comm}(j, st{:}i)\right)$$
$$\rightarrow Inv(g', pc_1, sleep_1', x_1, \ldots, pc_i', sleep_i, x_i', \ldots, pc_k, sleep_k, x_k)$$

Non-Interference (for each edge $\langle \ell, st, \ell' \rangle \in \Delta$ and each $i \in \{1, \ldots, k+1\}$):

$$Inv(g, pc_1, sleep_1, x_1, \ldots, pc_k, sleep_k, x_k) \tag{7}$$
$$\wedge \left(\bigwedge_{r=1}^{k} Inv(g, pc_{\sigma_i^r(1)}, sleep_{\sigma_i^r(1)}, x_{\sigma_i^r(1)}, \ldots, pc_{\sigma_i^r(k)}, sleep_{\sigma_i^r(k)}, x_{\sigma_i^r(k)})\right)$$
$$\wedge pc_\star = \ell \wedge pc_\star' = \ell' \wedge st(g, x_\star, g', x_\star')$$
$$\wedge \neg sleep_\star \wedge \bigwedge_{j \neq i} \left(sleep_j' \leftrightarrow (sleep_j \vee \widetilde{\text{pref}}(j, \star/i)) \wedge \text{comm}(j, st{:}\star)\right)$$
$$\rightarrow Inv(g', pc_1, sleep_1', x_1, \ldots, pc_k, sleep_k', x_k)$$

Safety (for each $i \in \{1, \ldots, k\}$ and $\ell \in \textbf{Loc}$ where $assert(\ell)$ is defined):

$$Inv(g, pc_1, sleep_1, x_1, \ldots, pc_k, sleep_k, x_k) \wedge pc_i = \ell \wedge \neg assert(\ell) \rightarrow \bot \tag{8}$$

Fig. 8. *Explicit-sleep* CHC encoding $\mathbb{TM}_{sleep}(\mathcal{P}, k)$ for the existence of a safe Ashcroft invariant of width k for the sleep-instrumented program $\mathcal{P}_{sleep}$. The encoding does not include the *id* variables. Further differences to the symbolic-sleep encoding are highlighted in red.

PROPOSITION 6.6 (EXPLICIT-SLEEP SOUNDNESS). *Let* Ψ_{Inv} *be a solution to* $\mathbb{TM}_{sleep}(\mathcal{P}, k)$. *Then*

$$\forall i_1, \ldots, i_k \,.\, id_{i_1} < \ldots < id_{i_k} \rightarrow \Psi_{Inv}(g, pc_{i_1}, sleep_{i_1}, x_{i_1}, \ldots, pc_{i_k}, sleep_{i_k}, x_{i_k})$$

is a safe Ashcroft invariant (of width k) for $\mathcal{P}_{sleep}$.

COROLLARY 6.7. *If the explicit-sleep encoding* $\mathbb{TM}_{sleep}(\mathcal{P}, k)$ *is satisfiable, the program* $\mathcal{P}$ *is correct.*

PROOF. Follows from Proposition 6.6 and Theorem 4.9. □

The following proposition states that the symbolic-sleep encoding and the explicit-sleep encoding are equi-satisfiable. Consequently, the explicit-sleep encoding still encodes the existence of an Ashcroft invariant of width k for the sleep-instrumented program.

PROPOSITION 6.8 (EQUISATISFIABILITY). *The explicit-sleep encoding* $\mathbb{TM}_{sleep}(\mathcal{P}, k)$ *is satisfiable iff the symbolic-sleep encoding* $\mathbb{TM}(\mathcal{P}_{sleep}, k)$ *is satisfiable.*

PROOF IDEA. If Φ_{Inv} is a solution for the symbolic-sleep encoding, then

$$\Psi_{Inv}(g, pc_1, sleep_1, x_1, \ldots, pc_k, sleep_k, x_k)$$
$$:\equiv \exists id_1, \ldots, id_k \,.\, \left(id_1 < \ldots < id_k \wedge \Phi_{Inv}(g, id_1, pc_1, sleep_1, x_1, \ldots, id_k, pc_k, sleep_k, x_k)\right)$$

is a solution for the explicit-sleep encoding. If Ψ_{Inv} is a solution for the explicit-sleep encoding, then

$$\Phi_{Inv}(g, id_1, pc_1, sleep_1, x_1, \ldots, id_k, pc_k, sleep_k, x_k)$$
$$:\equiv \bigvee_{\sigma \in \mathcal{S}_k} \left(id_{\sigma(1)} < \ldots < id_{\sigma(k)} \wedge \Psi_{Inv}(g, pc_{\sigma(1)}, sleep_{\sigma(1)}, x_{\sigma(1)}, \ldots, pc_{\sigma(k)}, sleep_{\sigma(k)}, x_{\sigma(k)})\right)$$

is a solution for the symbolic-sleep encoding, where $\mathcal{S}_k$ denotes the set of all permutations over the set $\{1, \ldots, k\}$. □

The factorial explosion inherent in the case distinction over all permutations of threads is precisely the cost we seek to avoid through the explicit-sleep encoding. Because the explicit-sleep encoding does not use variables for the thread IDs, the search space $\textbf{Search}(\mathbb{TM}_{sleep}(\mathcal{P}, k))$ for the explicit-sleep encoding is a strict subset of the search space $\textbf{Search}(\mathbb{TM}(\mathcal{P}_{sleep}, k))$ for the symbolic-sleep encoding. The above proposition clarifies that we neither lose expressivity, nor do we gain *qualitative* proof simplification, i.e., the solution space $\textbf{Sol}(\mathbb{TM}(\mathcal{P}_{sleep}, k))$ is empty if and only if the solution space $\textbf{Sol}(\mathbb{TM}_{sleep}(\mathcal{P}, k))$ is empty. Beyond that, the solution spaces are difficult to compare, as solutions range over different sets of variables. However, symmetry reduction has been shown to be practically beneficial in many settings [Clarke et al. 1998]. And indeed, Section 8 confirms empirically that the explicit-sleep encoding has significant practical benefit over the symbolic-sleep encoding when using state-of-the-art CHC solvers.

6.3 Inductive Invariants of Reduction Families

Proposition 4.10 states that the sleep-instrumented program represents a family of *minimal* reductions: Every equivalence class of traces is represented by a single trace in the reduction; if that representative is removed, the remaining set of traces is no longer a reduction. The intention is to not burden the verification with the proof of any redundant traces.

However, this "minimality" refers to the family of *infinite-state* programs $\mathcal{P}_{sleep}(n)_{n \in \mathbb{N}}$. When we fix a notion of *finite* proofs for the parameterized program $\mathcal{P}_{sleep}$, we are settled with a particular expressiveness to describe this infinite family of programs. It is not clear a priori that a certain kind of proof is expressive enough to fully benefit from the minimality of the reduction. And in fact, if we consider Ashcroft invariants, we observe that the expressiveness Ashcroft invariants gain through sleep instrumentation depends crucially on the invariants' width. Specifically, for Ashcroft invariants of width 1, no expressivity is gained through the reduction.

PROPOSITION 6.9 (COLLAPSE AT WIDTH 1). *Suppose there exists an Ashcroft invariant of width 1 for the sleep-instrumented program* $\mathcal{P}_{sleep}$. *Then there also exists an Ashcroft invariant of width 1 for the original program* $\mathcal{P}$.

Intuitively, the additional expressive power through sleep instrumentation can only be harnessed through *relational* assertions, i.e., assertions that relate the local variables (including program counter and sleep variables) of different threads. An Ashcroft invariant of width 1 does not include such relational assertions. It cannot even distinguish two threads. Hence, the Ashcroft invariant can either claim that all threads are asleep (which is unsound, as there is always at least one thread awake), or that none of the threads are asleep (i.e., there is no reduction).

By contrast, we have seen that for Ashcroft invariants of width 2 (and consequently, any higher width), we gain expressivity through sleep instrumentation. However, the fact that such invariants can *benefit* from reduction does not imply that they can *precisely capture* the infinite family of *minimal* reductions $\mathcal{R}(n)_{n \in \mathbb{N}}$ corresponding to a sleep-instrumented program $\mathcal{P}_{sleep}$. An Ashcroft invariant may simply capture overapproximations of the minimal reductions, which nevertheless allows for significant (qualitative) proof simplification (as in Example 4.11). Indeed we observe:

OBSERVATION 6.10. *There exist programs for which no Ashcroft invariant of any width precisely captures the reachable configurations of the reduction.*

EXPLANATION. Consider a program $\mathcal{P}$ with the template

$$\xrightarrow{} \underset{\ell_0}{\bigcirc} \xrightarrow{\mathit{st}_1} \underset{\ell_1}{\bigcirc} \xrightarrow{\mathit{st}_2} \underset{\ell_2}{\bigcirc}$$
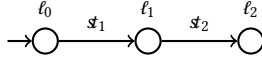
such that all statements (of different threads) commute, except for the fact that $\mathit{st}_2{:}i \not\leftrightarrows \mathit{st}_2{:}j$. An Ashcroft invariant of width k would have to capture that, when the program $\mathcal{P}(k)$ reaches the control locations $\langle \ell_1, \ldots, \ell_1 \rangle$, all threads except one are in the sleep set. The last thread to take a step must have had the maximal thread ID, otherwise it would have been added to the sleep set earlier. But then, in the last step, the next enabled statement of every other thread (st_2) commutes with the executed statement st_1, thus all threads with a lower thread ID are added to the sleep set.

Thus, if an Ashcroft invariant $\forall i_1, \ldots, i_k . \varphi$ precisely captures the reduction, we must have

$$\varphi \wedge (id_{i_1} < \ldots < id_{i_k}) \wedge pc_{i_1} = \ell_1 \wedge \ldots \wedge pc_{i_k} = \ell_1 \models sleep_{i_1} \wedge \ldots \wedge sleep_{i_{k-1}} \tag{9}$$

However, in the program $\mathcal{P}(k+1)$, we can reach a configuration with the control locations $\langle \ell_1, \ldots, \ell_1, \ell_2 \rangle$, such that the sleep set is empty. In particular, if the last step is the execution of $\mathit{st}_2{:}k+1$, this statement does not commute with the enabled statements of all other threads, thus the sleep set is emptied. We instantiate the Ashcroft invariant $\forall i_1, \ldots, i_k . \varphi$ such that it considers the first k threads ($\{i_1, \ldots, i_k\} = \{1, \ldots, k\}$) in increasing order of IDs ($id_{i_1} < \ldots < id_{i_k}$). Equation (9) prescribes that the sleep variables $sleep_{i_1}, \ldots, sleep_{i_{k-1}}$ are true, when indeed for this configuration, they are all false. Thus the configuration, while reachable, does not satisfy the Ashcroft invariant. □

Note that the key obstacle to precisely capturing the reduction in the above proof was the non-commutativity of statements $\mathit{st}_2{:}i$ and $\mathit{st}_2{:}j$. We observe:

OBSERVATION 6.11. *If all statements of different threads commute, an Ashcroft invariant of width 2 can capture a tight overapproximation of the reduction inherent in $\mathcal{P}_{sleep}$.*

EXPLANATION. The following Ashcroft invariant precisely captures the control flow:

$$\forall i, j . id_i < id_j \wedge pc_j \neq \ell_{\mathrm{init}} \rightarrow sleep_i$$

In other words, as soon as a thread j takes a step, all threads with smaller IDs are put to sleep and never awakened again. This is satisfied by all reachable configurations of $\mathcal{P}_{sleep}$, and the Ashcroft invariant is *precise*: While it may include unreachable configurations $\langle \vec{\ell}, s \rangle$ of $\mathcal{P}_{sleep}$, such configurations are either *(i)* unreachable due to data constraints (not due to the reduction), or *(ii)* there is a reachable configuration $\langle \vec{\ell}, s' \rangle$ with the same control locations and variable values, except that s may assign more *sleep* variables to $\bot$. The latter case is not problematic however, because $\langle \vec{\ell}, s \rangle$ and $\langle \vec{\ell}, s' \rangle$ satisfy the same invariants over program variables, and all executions possible from $\langle \vec{\ell}, s \rangle$ are also possible from $\langle \vec{\ell}, s' \rangle$. □

7 BROADER NOTIONS OF SOUND COMMUTATIVITY

We have so far focused on one particular notion of commutativity (see Section 4): Executing commuting statements in either order must yield the same semantics. The framework of commutativity theory however admits more general notions of commutativity, from which verification can benefit. Specifically, we extend our approach along two lines:

Contextual Commutativity. The position of statements inside a program, and in an execution, provides a rich context which can benefit commutativity. Consider for instance the statements `queue[current]:=data` (line 12) and `msg:=queue[idx]` (line 23) from the program $\mathcal{P}^{\mathrm{notify}}$ shown in Fig. 2. These statements do not, in general, commute; in the case that idx = current, executing the statements in different orders yields different semantics.

However, it is clear from the code of $\mathcal{P}^{\mathrm{notify}}$ that, in every state where these statements are enabled, it holds that idx < current. In such contexts, the order of execution does indeed not matter. Hence, we say that the statements commute in the context idx < current (or, more broadly, in the context idx ≠ current).

Semi-Commutativity. Commutativity as in Section 4 defines a symmetric relation: If $\mathit{st}_1{:}i$ commutes with $\mathit{st}_2{:}j$, then $\mathit{st}_2{:}j$ commutes with $\mathit{st}_1{:}i$. The semantics of both execution orders are equal, and consequently we can swap the statements in either direction to get an equivalent trace. Let us entertain a non-symmetric variation. Consider for instance the statements current:=current+1 (line 13) and assume idx < current (line 22) from the program $\mathcal{P}^{\mathrm{notify}}$ shown in Fig. 2. These statements do not commute. Specifically, the execution of assume idx < current:i current:=current+1:j blocks if we have idx = current, the execution of current:=current+1:j assume idx < current:i does not. Generally, executing the increment of current first allows a strict superset of executions. Thus, it is sound to eliminate a trace in which the increment happens after the assume statement in favor of a trace with the opposite order, but the reverse is not true.

Without these generalized notions of commutativity, the program $\mathcal{P}^{\mathrm{notify}}$ would not admit a simple proof. We thus extend our approach.

Definition 7.1 (Contextual Semi-Commutativity). Let φ be a formula over global program variables and local variables indexed by i or j. Statements $\mathit{st}_1{:}i$ and $\mathit{st}_2{:}j$ *semi-commute in the context φ*, denoted $\mathit{st}_1{:}i \curvearrowright_\varphi \mathit{st}_2{:}j$, if for all states s, s' such that s satisfies φ, the following implication holds:

$$(s, s') \in [\![\mathit{st}_1\mathit{st}_2]\!] \Rightarrow (s, s') \in [\![\mathit{st}_2\mathit{st}_1]\!]$$

The general framework of commutativity theory is adapted accordingly. In place of an equivalence relation, we now consider a preorder over traces (i.e., we lose symmetry). Specifically, we say that a trace τ_1 is *covered by* a trace τ_2 if τ_2 can be derived from τ_1 by a sequence of swaps of adjacent statements, where for every swap from a trace τ' $(\mathit{st}_1{:}i)$ $(\mathit{st}_2{:}j)$ τ'' to a trace τ' $(\mathit{st}_2{:}j)$ $(\mathit{st}_1{:}i)$ τ'', we must have $\mathit{st}_1{:}i \curvearrowright_\varphi \mathit{st}_2{:}j$ for some φ that always holds after the execution of the prefix τ'. A reduction language of traces L is then a subset L' where for every trace $\tau \in L$, there exists some trace $\tau' \in L'$ such that τ is covered by τ'.

We modify the sleep instrumentation to account for contextual semi-commutativity by redefining the commutativity test. To this end, we assume the existence of mapping from indexed statements $\mathit{st}_1{:}i, \mathit{st}_2{:}j$ to *commutativity conditions* $\varphi_{\mathrm{comm}}(\mathit{st}_1{:}i, \mathit{st}_2{:}j)$, i.e., formulae over global variables as well as local variables of threads i and j, such that $\mathit{st}_1{:}i \curvearrowright_{\varphi_{\mathrm{comm}}(\mathit{st}_1{:}i,\mathit{st}_2{:}j)} \mathit{st}_2{:}j$.

Definition 7.2 (Contextual Semi-Commutativity Test). The *contextual semi-commutativity test* $\mathrm{comm}(j, \mathit{st}{:}i)$ is defined as the formula

$$\mathrm{comm}(j, \mathit{st}{:}i) :\equiv \bigvee_{\ell \in \mathbf{Loc}} \left(pc_j = \ell \wedge \bigwedge_{\mathit{st}' \in \mathit{enabled}(\ell)} \varphi_{\mathrm{comm}}(\mathit{st}'{:}j, \mathit{st}{:}i) \right).$$

At this point it is crucial that in the instrumentation $\iota(\mathit{st})$ of a statement st, the update of the *sleep* variables, including the evaluation of the contextual semi-commutativity test, is performed *before* the original statement st. Otherwise the instrumentation would not faithfully reflect contextual semi-commutativity and might become unsound.

In the implementation of our approach (see Section 8), we generate commutativity conditions $\varphi_{\mathrm{comm}}(\mathit{st}_1{:}i, \mathit{st}_2{:}j)$ by encoding semi-commutativity as a first-order logic formula and applying an abduction algorithm to find sufficient conditions to guarantee it.

The modified sleep set instrumentation with contextual semi-commutativity tests still represents a reduction (Theorem 4.8) and satisfies soundness (Theorem 4.9) as well as conservative extension (Theorem 4.13). Furthermore, the CHC encodings introduced in Section 6 can be used with the contextual semi-commutativity test in place of the commutativity test, and remain sound. However, the represented lexicographical reductions are not necessarily minimal [Farzan and Vandikas 2020], i.e., Proposition 4.10 does not hold. This is because the covering relation is not symmetric. There may exist traces τ_1, τ_2 such τ_1 is not covered by any lexicographically smaller trace, τ_2 covers τ_1, and τ_2 is only covered by itself. Then both τ_1 and τ_2 appear in the lexicographical reduction, yet including τ_2 would suffice.

8 EVALUATION

As a proof of concept, we have developed a tool that integrates reduction in parameterized verification. In particular, we implemented the different CHC encodings for the existence of an Ashcroft invariant for the sleep-instrumented program $\mathcal{P}_{sleep}$, as discussed in Section 6. Our tool reads Boogie [Leino 2008] programs, generates the CHC clauses, and executes different CHC solvers to check if the CHC system is satisfiable. In particular, we used the state-of-the-art CHC solvers ELDAR-ICA (github.com/uuverifiers/eldarica), GOLEM (verify.inf.usi.ch/Golem) and Z3/SPACER (github.com/Z3Prover/z3). We evaluated the tool on a number of parameterized programs from the literature as well as custom benchmarks. The purpose of this evaluation is to answer the following questions:

Q1: Can the modular approach of (1) encoding reductions through sleep instrumentation and (2) subsequently verifying the resulting parameterized program work in practice?

Q2: Can we observe a practical benefit of the symmetry-aware *explicit-sleep* CHC encoding in comparison to the default *symbolic-sleep* encoding?

We executed the benchmarks on a Debian 10.10 machine with a AMD Ryzen Threadripper 3970X 32-Core Processor using the BenchExec benchmarking tool [Beyer et al. 2019]. Each verification run was given a timeout of 30 min and a memory limit of 15 GB. Our suite of 19 benchmarks is comprised of a number of variations (inc-b?dec-*) of the program $\mathcal{P}^{\pm}$ (see Fig. 1), where a variable is incremented and decremented by each thread and compared with 0; we also included variants where a nondeterministic value is added to and subtracted from the variable (add-sub-*). Several examples (namely lock, ticket, and mutex-*) are taken from [Hoenicke et al. 2017]; the mutex-* examples correspond to the program $\mathcal{P}^{K\pm}$ (see Fig. 3). As more complex programs, we included the bluetooth example in the form presented in [Farzan et al. 2014], the example presented in Section 2 (notify-listeners), the thread-pooling example from [Farzan et al. 2015], a program in which each thread computes the same sum of array elements (equalsum-ghost), and a custom example involving communication via queues (line-queue). All benchmarks, as well as the generated CHC encodings, can be found on zenodo [Farzan et al. 2023a].

Table 1 shows the benchmark results. The reported CPU time encompasses both the time required to generate the CHC clauses (typically quite small) and the time required by the fastest successful CHC solver, if any solver is successful.

Regarding **Q1**, we observe that the approach (in the explicit-sleep configuration) is able to verify 14 out of 19 benchmarks. In particular, we successfully show correctness of non-trivial benchmarks such as bluetooth and notify-listeners. Without reductions, these programs do not have a safe Ashcroft invariant; a proof would require complex ghost state and/or quantified invariants.

At the same time, even for the most successful configuration (explicit-sleep), three state-of-the-art CHC solvers are unable to solve 5 of our benchmarks. Beyond the possibility of general improvements in CHC solving, a possible way to improve the situation may be to guide the solvers to specifically take advantage of the reduction. This could be beneficial in two scenarios: First,

Table 1. Benchmark results. sat indicates that an Ashcroft invariant (of width k) was found, unsat indicates that a CHC solver proved that no such Ashcroft invariant exists, "TO" indicates a timeout (> 30 min).

Program	k	no reduction		symbolic-sleep		explicit-sleep	
		status	CPU time (s)	status	CPU time (s)	status	CPU time (s)
add-sub-nondet	2	unsat	20.5	sat	416.0	sat	74.5
add-sub-positive-nondet	2	unsat	51.1	sat	1 590.0	sat	144.0
bluetooth	2	unsat	6.5	TO	–	sat	532.5
equalsum-ghost	2	TO	–	TO	–	TO	–
inc-bdec	2	unsat	5.8	sat	76.3	sat	51.3
inc-dec-eq0-locked-assert	2	sat	59.6	TO	–	sat	726.0
inc-dec-eq0-locked	2	unsat	110.0	TO	–	TO	–
inc-dec-eq0	2	unsat	8.9	sat	112.0	sat	24.7
inc-dec-geq0	2	unsat	4.3	sat	5.8	sat	5.7
line-queue	2	TO	–	TO	–	TO	–
lock	1	sat	4.0	sat	4.6	sat	4.7
mutex-3	2	unsat	5.2	sat	5.3	sat	4.5
	4	sat	8.7	sat	95.5	TO	–
mutex-4	2	unsat	3.5	sat	5.6	sat	4.3
	5	sat	57.4	sat	723.0	TO	–
mutex-5	2	unsat	4.5	sat	5.3	sat	4.0
	6	sat	354.0	TO	–	TO	–
mutex-unbounded	2	unsat	4.5	sat	6.6	sat	4.2
notify-listeners	1	TO	–	TO	–	sat	379.0
numbered-array	2	sat	4.0	sat	5.8	sat	5.4
thread-pooling	2	TO	–	TO	–	TO	–
ticket	2	sat	332.0	TO	–	TO	–

for programs which do not have an Ashcroft invariant without reduction, one could prevent the solver from considering solutions that ignore the instrumentation. Second, one could try to prevent the solver from considering solutions that use the *sleep* variables in "exotic" ways unsuitable to express reduction. The second case could also reduce the overhead from the instrumentation for programs where an Ashcroft invariant exists without reduction. As an example, consider the program inc-dec-eq0-locked-assert, which has an Ashcroft invariant even without reduction. Here, the CHC solvers spend significantly longer to find a solution when the instrumentation is present. For the program ticket, the solvers even time out, even though the program can be proven without reduction.

The evaluation data clearly shows the performance advantage of the explicit-sleep encoding. With this encoding, our tool is able to verify 13 programs, compared to only 11 programs with the symbolic-sleep encoding. Notice in particular that the complex program notify-listeners is only proved correct by the explicit-sleep encoding. Furthermore, for programs solved by both the symbolic-sleep and explicit-sleep encoding, the explicit-sleep encoding can lead to significant speedup, up to a factor of 10x in the most extreme case (add-sub-positive-nondet). Despite the increased number of clauses, we do not observe any overhead for the explicit-sleep encoding.

9 RELATED WORK

There is a huge body of work on verification of parameterized programs. It is noteworthy that this paper does not put forward a new (algorithmic) framework for verifying parameterized programs, but rather suggests a generic way of incorporating commutativity into any existing framework. As such, we will only very briefly survey a few techniques only to justify why we chose a particular one as the framework to use for our proof of concept application.

9.1 Parameterized Program Verification

In *invisible invariants* [Arons et al. 2001; Pnueli et al. 2001], a candidate for an Ashcroft invariant is constructed by first computing the set of reachable states of the instance of the program with k threads, and then generalizing the concrete thread identifiers in the reachable states. The candidate (a universally quantified formula with k variables over thread identifiers) is then verified using a syntactic *cutoff theorem*. This approach, as well as other heuristic searchers [Emmi et al. 2010] for Ashcroft invariants, do not have a guarantee of completeness. Therefore they suffer from the problem that, if they fail, one does not know whether there is no proof with k quantifiers or whether the heuristic did not find it. This is why we opted to build our reduction framework on top of thread-modular proofs [Hoenicke et al. 2017], which come with the guarantee of finding Ashcroft invariants when one exists (modulo incompleteness of the Horn clause solver) or proving that no Ashcroft invariant exists. This allows for a more principled comparison of the power of the framework in proving the original program or a lexicographical reduction of it.

In [Farzan et al. 2014; Kaiser et al. 2014], *counting proofs* are constructed automatically. This can be viewed as a partial solution to the problem of discovering the required ghost state automatically; partial, in the sense that only ghost counters can be discovered. Such techniques are complementary to the proposal in this paper; the simpler the proof, the more likely that a combination of this technique can succeed in discovering it automatically.

Grebenshchikov et al. [2012], Hojjat et al. [2014], Gurfinkel et al. [2016], and Monniaux and Gonnord [2016] study Horn constraints for k-thread-modular proofs, closely related to the framework we chose to demonstrate our approach [Hoenicke et al. 2017].

9.2 Commutativity for Proof Simplification

There has been extensive work in incorporating commutativity into verification of concurrent programs. One big cluster of such work appears under the title of partial order reduction (POR) [Abdulla et al. 2014; Flanagan and Godefroid 2005; Godefroid 1996; Kahlon et al. 2009], and much of this work is concerned with finite-state systems or executions of bounded length.

In the context of proofs of infinite-state programs, the focus of commutativity reasoning in algorithmic verification so far has been on programs with a bounded number of threads [Chu and Jaffar 2014; Farzan et al. 2022; Farzan and Vandikas 2019, 2020; Wachter et al. 2013].

Popeea et al. [2014] integrate the theory of Lipton's *movers* [Lipton 1975] with compositional proofs in the style of Owicki and Gries, to verify programs with a bounded number of threads. The approach defines a complex Horn clause system that combines compositional reasoning, the determination of *mover annotations* (i.e., commutativity checks) and the search for reducible blocks.

In interactive proofs [Elmas et al. 2009; Kragl and Qadeer 2018], commutativity reasoning based on the principle of Lipton's *movers* has been incorporated in a way applicable to programs with a bounded number of threads as well as programs with an unbounded number of threads, despite not explicitly using the modeling formalism of parameterized programs. Essentially, the input program is alternatingly reduced and further abstracted. Each abstraction step may allow more statements to commute, which enables further reduction. Since being a *mover* can be viewed as a local property of an atomic problem step, the size of the environment (finite vs unbounded number of threads) makes no difference in how larger atomic blocks are formed out of smaller ones by reasoning about movers, and thus a successfully verified program is correct for any number of threads.

Flanagan and Freund [2020] also apply mover reasoning to simplify verification of programs with an unbounded number of threads. Data structures are annotated with *synchronization specifications* that indicate mover types (i.e., semi-commutativity) of read and write accesses to the data structure. Users specify a reduction of a concurrent program by manually instrumenting the program with

yield points indicating where interleaving with other threads may occur in the reduction. The verifier then checks if this instrumentation is indeed sound, i.e., encodes a reduction of the program.

As discussed in [Farzan and Vandikas 2020], however, the kinds of program reductions that result from Lipton's movers are not comparable with those that are produced as lexicographical reductions of (binary) commutativity relations. Besides, the locality advantages of movers disappear in the context where the goal is anything but large block reasoning: for example, a *lockstep* reduction. Such reductions are by definition not local to a single thread/process.

In *inductive sequentialization* [Kragl et al. 2020], a vaguely similar philosophy about proof simplification is used: Rather than reason about arbitrarily complicated executions of distributed protocol, one can reason about their equivalence to simpler ones and as such only give a proof of correctness for the simpler ones. It is important to note that the notion of equivalence employed is not the simple syntactic one (based on commutativity) used in this paper. As such, even the reasoning about such equivalences may involve the use of invariants, and other proof-type constructs. The final product is a proof of refinement between the complex and the simple protocols, and the ingredients of the proof are provided by a user.

10 CONCLUSION AND FUTURE WORK

This paper proposes a methodology for incorporating commutativity reasoning into algorithmic verification of parameterized programs. We put forward the thesis that this is a worthwhile cause, because commutativity-based reductions can simplify the proofs of these programs in a precise sense: a possible substantial complexity reduction in the nature of the ghost state required for the proof. The solution was devised with an eye on practical concerns, in the sense that rather than devising a whole new algorithmic framework, one should be able to use existing frameworks for parameterized program verification with little effort.

Our investigation of this problem has led us to several new research questions that would be interesting to explore in the future. Our results from Section 6.3 highlight the fact that Ashcroft invariants, as a standard family of global invariants for parameterized programs, lack the expressive power to encode optimal reductions for the entire family of programs represented by the parameterized program for an arbitrary commutativity relation. It would be interesting to investigate whether this lack of expressivity is shared by other ways of giving a finitely-representable proof to a parameterized program, for instance *proof spaces* [Farzan et al. 2015].

Classical trace theory, which studies commutativity in a principled way, relies on a *finite* alphabet of program actions. For parameterized programs, one needs an infinite (indexed) alphabet of actions to model the program behaviour faithfully. Most of the work on program reductions relies on a classic result from trace theory that says "the set of lexicographical representatives of a regular and (commutativity) closed language is regular". The notion of regularity for indexed alphabets is less standard, and can be defined based on a number of *data* automata like register, nominal, or predicate automata. It will be interesting to investigate if an analogous result for these automata exists and whether it can suggest fundamentally different ways of incorporating commutativity in verification of parameterized programs.

DATA AVAILABILITY STATEMENT

The benchmark programs used in the evaluation (Section 8), as well as the generated CHC encodings for the three evaluated configurations are available on zenodo [Farzan et al. 2023a]. The generation of the different CHC encodings is implemented as part of the ULTIMATE program analysis framework (`ultimate-pa.org`) and is available on github (`github.com/ultimate-pa/ultimate`). The archive describes how ULTIMATE can be used to generate and solve the different CHC encodings. An extended version of the paper (including proofs) is also available [Farzan et al. 2023b].

ACKNOWLEDGMENTS

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 373–384. https://doi.org/10.1145/2535838.2535845

Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. 2001. Parameterized Verification with Automatically Computed Inductive Assertions. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 221–234. https://doi.org/10.1007/3-540-44585-4_19

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

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

Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. 1998. Symmetry Reductions in Model Checking. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1427)*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer, 147–158. https://doi.org/10.1007/BFb0028741

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

Michael Emmi, Rupak Majumdar, and Roman Manevich. 2010. Parameterized verification of transactional memories. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 134–145. https://doi.org/10.1145/1806596.1806613

Azadeh Farzan. 2023. Commutativity in Automated Verification. In *LICS*. 1–7. https://doi.org/10.1109/LICS56636.2023.10175734

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

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2015. Proof Spaces for Unbounded Parallelism. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 407–420. https://doi.org/10.1145/2676726.2677012

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

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2023a. *Benchmarks for POPL'24 Paper "Commutativity Simplifies Proofs of Parameterized Programs"*. https://doi.org/10.5281/zenodo.10119773

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2023b. *Commutativity Simplifies Proofs of Parameterized Programs (Extended Version)*. Technical Report. https://dominik-klumpp.net/publications/popl24/extended.pdf

Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 200–218. https://doi.org/10.1007/978-3-030-25540-4_11

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

Cormac Flanagan and Stephen N. Freund. 2020. The Anchor verifier for blocking and non-blocking concurrent software. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 156:1–156:29. https://doi.org/10.1145/3428224

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

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

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science, Vol. 1032. Springer. https://doi.org/10.1007/3-540-60761-7

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 405–416. https://doi.org/10.1145/2254064.2254112

Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. 2016. SMT-based verification of parameterized systems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 338–348. https://doi.org/10.1145/2950290.2950330

Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. 2017. Thread modularity at many levels: a pearl in compositional verification. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 473–485. https://doi.org/10.1145/3009837.3009893

Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. 2014. Horn Clauses for Communicating Timed Systems. In *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014 (EPTCS, Vol. 169)*, Nikolaj S. Bjørner, Fabio Fioravanti, Andrey Rybalchenko, and Valerio Senni (Eds.). 39–52. https://doi.org/10.4204/EPTCS.169.6

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

Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2014. Lost in Abstraction: Monotonicity in Multi-threaded Programs. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8704)*, Paolo Baldan and Daniele Gorla (Eds.). Springer, 141–155. https://doi.org/10.1007/978-3-662-44584-6_11

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

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

K. Rustan M. Leino. 2008. This is Boogie 2. (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

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

David Monniaux and Laure Gonnord. 2016. Cell Morphing: From Array Programs to Array-Free Horn Clauses. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 361–382. https://doi.org/10.1007/978-3-662-53413-7_18

Leonor Prensa Nieto. 2001. Completeness of the Owicki-Gries System for Parameterized Parallel Programs. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, April 23-27, 2001.* IEEE Computer Society, 150. https://doi.org/10.1109/IPDPS.2001.925138

Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2031)*, Tiziana Margaria and Wang Yi (Eds.). Springer, 82–97. https://doi.org/10.1007/3-540-45319-9_7

Corneliu Popeea, Andrey Rybalchenko, and Andreas Wilhelm. 2014. Reduction for compositional verification of multi-threaded programs. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014.* IEEE, 187–194. https://doi.org/10.1109/FMCAD.2014.6987612

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

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/