On the Complexity of Checking Soundness of Natural Reductions

Constantin Enea[1], Azadeh Farzan[2], and Dominik Klumpp[1(✉)]

[1] LIX – CNRS – École Polytechnique, Palaiseau, France
{cenea,klumpp}@lix.polytechnique.fr
[2] University of Toronto, Toronto, Canada
azadeh@cs.toronto.edu

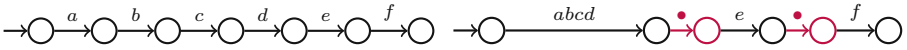
Abstract. The verification of *reductions*, representative subsets of interleavings, simplifies correctness proofs of parameterized concurrent programs. We introduce an expressive class of syntactic reductions, which we call *natural reductions*. Natural reductions are specified by introducing atomic blocks and global rendezvous points in the parameterized program's thread template. We study the problem of deciding whether a given natural reduction is sound wrt. a given (semi-)commutativity relation. In the case that there is no synchronization between threads, we present a sound and complete polynomial-time algorithm. In the case where synchronization is considered, we provide a general lower bound for the problem (parametric in the synchronization mechanism), and show that the problem is coNP-hard already for a simple mechanism like locking.

1 Introduction

A *reduction* of a concurrent program is a program with a subset of the behaviors of the original that can be *soundly* verified in its place. A big body of research in program verification, in both algorithmic and deductive traditions, has argued that the use of reductions can be critical to the success of the verification task. On the algorithmic verification side, the reductions either come from an a priori fixed set [10,11] or the verification algorithm has to pay the steep cost of searching for them [13]. On the deductive verification side, proof systems like CSPEC [2], Armada [22], Anchor [14] and Civl [19] include proof tactics based on Lipton's reduction [21] for showing that a user-provided reduction is sound, which can be interleaved with other proof tactics that relate to inductive invariant reasoning, for instance. These proof systems have been used to verify real-world examples such as a concurrent garbage collector [19], an implementation of the Paxos protocol [20], or concurrent data structures [24]. Reductions play a critical role in the proofs of these examples.

In this paper, we investigate the complexity of checking whether some given reduction of a concurrent program is sound. We focus on *parameterized concurrent programs*, that is, concurrent programs in which an arbitrary number of threads can be instantiated from a single thread template. These programs

E. Darulova et al. (Eds.): CAV 2026, LNCS 16682, pp. 19–41, 2026.
https://doi.org/10.1007/978-3-032-32519-8_2



(a) Thread template (control flow graph). (b) Thread template of a natural reduction.

Fig. 1. Example of natural reductions. The symbol • synchronizes all threads.

appear everywhere in distributed and parallel computing. We consider a class of reductions that we call *natural* that can be specified syntactically with ease and make it straightforward for programmers to envision the reduced program and reason about it. Hence, *natural reductions* enable programmers to interact with verification tools by proposing reductions that may simplify verification tasks.

Natural reductions are defined through two syntactic program transformations, which both have deep connections to traditions of reasoning about concurrent programs:

- *Atomic Blocks*: The programmer can specify any number of syntactic atomic blocks in the program, which accordingly prunes the space of possible concurrent program behaviours, excluding all behaviours in which one thread's execution of the atomic block is interrupted by another thread.
- *Synchronous Reductions:* The programmer uses a unique symbol as a *global rendezvous point* among all existing threads, and as such restricts the space of program behaviours to those in which the threads meet at the rendezvous point before continuing.

The former has been broadly used as the foundation of *local reasoning* in concurrency [2,6,14,18,19,21,22], and the latter has been used to simplify reasoning about distributed protocols [3,5,7,16]. In many distributed protocols, the behavior of each process is structured as a sequence of rounds and it can be shown that processes executing rounds synchronously and in lock-step is a sound reduction of the original set of asynchronous behaviors (where processes may be executing different rounds at a point of time). The user can use a combination of both to restrict the set of concurrent program behaviours and hence *specify* a reduction.

Example. Consider a program whose thread template is given by the control flow graph in Fig. 1a, where a, b, c, d, e, f are the atomic actions of the program. A user can specify a synchronous reduction by inserting global rendezvous points between the actions d and e resp. between e and f. This eliminates all program behaviours in which the actions e and f interleave with the actions a, b, c, d resp. with each other. The behaviours of the reduced program are thus partitioned into three *rounds* or *phases*: a first phase where all threads execute a, b, c and d; a second phase where all threads execute e; and a third phase where all threads execute f. Furthermore, the user can specify that the first phase executes atomically; thereby allowing the verification to use local reasoning without interference by other threads (that also execute a, b, c, or d). The specified natural reduction is represented by the thread template in Fig. 1b. The global rendezvous points are represented by the symbol •, and the atomic execution of the first phase is represented by a single edge labeled with the concatenated actions *abcd*.

We say a (specified) reduction is *sound* if every behaviour of the original program is equivalent to a behaviour in the reduction up to an *equivalence relation* induced by a *sound commutativity relation* on their common set of actions. A sound commutativity relation is a (not necessarily symmetric[1]) relation on the set of atomic actions that includes a pair (a_1, a_2) if and only if the semantics of $a_2 a_1$ includes all behaviours of $a_1 a_2$. A *sound reduction* can be soundly verified in place of the original program (i.e., the reduction satisfies the same postconditions as the original program). Hence, given a *natural reduction* specified by the user, we are interested in the problem of determining whether it is sound.

Example. Let the following be a sound commutativity relation:

$$\mathrm{I} = \big(\{e\} \times \{a, b, c, d\} \big) \cup \big(\{f\} \times \{a, b, c, d, e\} \big) \cup \{a, b, c, d\}^2$$

Then the natural reduction given in Fig. 1 is sound. The first term of the union ensures e can always be commuted to the second phase of an execution, and similarly the second term ensures f can be commuted to the third phase. The third term suffices to ensure the soundness of considering only atomic executions of the first phase (though not all commutation pairs are necessary for this).

In this paper, we investigate the soundness problem of natural reductions for two different abstract models of program behaviours. First, we consider a model in which program data is entirely abstracted away, and every syntactic behaviour of the program is valid behaviour in the abstract model. In a sense, the only information remaining about program semantics is given through the *sound commutativity* relation on the set of actions. This is the model that underlies previous works on reductions in deductive verification, in particular those based on Lipton's reduction [21]. We show that under this model, any natural reduction can be checked in *polynomial time* for soundness (Theorem 4.14). Our argument is constructive. We present an algorithm that checks soundness of a reduction proposed by atomic blocks in polynomial time. We also present an algorithm that checks soundness of a synchronous reduction in polynomial time. We then argue how the algorithms compose, and hence, a natural reduction with several atomic blocks and a rendezvous point can be checked for soundness in polynomial time.

Second, we consider a model in which the original program uses locks for synchronization, and the abstraction level at which the semantics of locking is visible and respected. This adds a layer of complication to the problem because one has to argue that every behaviour of the original program, *that respects the locking semantics*, has an equivalent up-to-commutativity behaviour in the specified natural reduction. We show that checking the soundness of a natural reduction in this setup is coNP-hard, no matter how much we limit the use of constructs. In other words, it is coNP-hard even if a natural reduction is specified using a single atomic block, and it is coNP-hard even if it is specified only using a rendezvous point. Consequently, while fully respecting the locking semantics is theoretically more precise, it is typically too costly for practical application in deductive proof systems.

[1] In the non-symmetric case, the equivalence relation degenerates to a preorder.

Example. Consider the parameterized program where each thread executes the code shown in Fig. 2 (ignoring the instrumentation in purple). At the most abstract level (where all program data is abstracted away), this program is represented by the thread template shown in Fig. 1a. The statements have been replaced by action names a, b, c, d, e, f without semantic information. The commutativity relation is the only semantic information. For the specific actions a, b, c, d, e, f given in Fig. 2, we have the commutativity relation $I = \{a, b, c, d, e, f\}^2 \setminus \{(b, c), (c, b), (c, c), (d, a), (e, f)\}$. The instrumentation in Fig. 2 is a syntactic representation of the natural reduction in Fig. 1b. However, this natural reduction is *unsound* in the fully abstract model; in particular, *abcd* cannot be atomic. Consider the case that thread 1 executes b, then thread 2 executes c, and finally thread 1 executes c. As $(b, c) \notin I$ and $(c, c) \notin I$, this behaviour has no representative in the reduction (where *abcd* executes atomically).

```
atomic {
  acq(m)                          // a
  idx := ctr                      // b
  ctr := idx + 1                  // c
  rel(m)                          // d
}
sync()                            // •
  arr[idx] := 42                  // e
sync()                            // •
  assume idx == 0 ||
         arr[idx-1] == 42         // f
```

Fig. 2. Concrete source code for threads (`idx` is local; `ctr`, the array `arr` and the lock `m` are shared). The instrumentation in purple specifies a natural reduction.

However, if we consider the program at an abstraction level that respects locks, represented by the thread template in Fig. 3, it becomes clear that the described behaviour is not possible, as both threads would have to hold the lock `m`. In this model, the specified reduction is sound. Finally, note that the introduction of the two global rendezvous points is sound in either model.

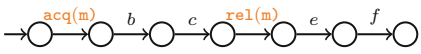


Fig. 3. Thread template model at abstraction level that respects lock synchronization.

Roadmap. Section 2 introduces our formalism for parameterized concurrent programs and (general) reductions. We define *natural* reductions in Sect. 3. Section 4 contains the key results of the paper on deciding soundness of atomic blocks (Sect. 4.1), global rendezvous points (Sect. 4.2) and their combination (Sect. 4.3). We contrast our approach with the classical technique of *Lipton reductions* in Sect. 5. Then, Sect. 6 studies the soundness problem for programs with lock synchronization. We give an overview of related work in Sect. 7 and conclude in Sect. 8.

An extended version of our paper, including proofs, is available online [8].

2 Parameterized Programs and Reductions

We use finite sequences to represent behaviours (*interleavings*) of concurrent programs. X^* denotes the set of all finite sequences σ over elements in a given set X (the free monoid generated by X). $\wp(X)$ is the powerset of X. We call a function $\mu : X \to \wp(Y^*)$ a *morphism* and extend it to a function $\mu : X^* \to \wp(Y^*)$ (with $\mu(\varepsilon) = \{\varepsilon\}$, $\mu(x\sigma) = \mu(x)\mu(\sigma)$ for $x \in X, \sigma \in X^*$) and further to a function $\mu : \wp(X^*) \to \wp(Y^*)$ (with $\mu(L) = \bigcup_{\sigma \in L} \mu(\sigma)$). By $\tau|_Z$, we denote the projection of τ to Z (the morphism with $z|_Z = \{z\}$ for $z \in X \cap Z$ and $x|_Z = \{\varepsilon\}$ for $x \in X \setminus Z$). The expression $|\sigma|$ denotes the length of σ, and $|\sigma|_x$ is the number of occurrences of x in σ. Square brackets $[x, y, z]$ denote multisets.

2.1 Parameterized Programs

A *parameterized program* is a concurrent program, where, at runtime, n threads all execute the same thread template (the number n is the *parameter*). For the purpose of this paper, we model atomic actions in the thread template abstractly as elements $a, b, c, \ldots$ of an infinite set Act.

Given a set of actions A, we write A_{idx} for $A \times \mathbb{N}$, the set of indexed actions, where an *indexed action* $\langle a : i \rangle \in A_{\mathsf{idx}}$ represents action a being executed by the i-th thread (the thread executing an action may matter, e.g. if the action accesses thread-local variables). We represent the interleavings of actions from different threads by *indexed traces* $\tau \in A^*_{\mathsf{idx}}$, and write $\langle \tau : i \rangle$ for $\tau \in A^*$ to denote the corresponding indexed trace where all actions are executed by thread i. By $\lfloor \tau \rfloor_i$, we denote the trace of all actions a where $\langle a : i \rangle$ appears in τ (the morphism with $\lfloor \langle a : i \rangle \rfloor_i = \{a\}$ and $\lfloor \langle a : j \rangle \rfloor_i = \{\varepsilon\}$ for $j \neq i$). We lift a morphism $\mu : X \to \wp(Y^*)$ to indexed actions by defining the morphism $\widehat{\mu}$ with $\widehat{\mu}(\langle a : i \rangle) = \{\, \langle \tau : i \rangle \mid \tau \in \mu(a) \,\}$.

Though we generally abstract away from specific actions and their semantics, we need to model the semantics of certain actions for the purpose of *synchronization* between the threads, through locks, semaphores, rendez-vous etc.

Definition 2.1. *A* synchronization alphabet $(\mathbb{S}, \mathsf{sync})$ *consists of a set* $\mathbb{S}$ *of synchronization actions (disjoint from* Act*) and a predicate* sync *over traces in* $(\mathsf{Act} \cup \mathbb{S})^*_{\mathsf{idx}}$*, where* sync *is preserved by all permutations of thread indices.*

In particular, the *lock synchronization alphabet* over the infinite set M of lock variables is given by $\mathbb{S}_{\mathsf{lock}} = \{\, \mathtt{acq}(m), \mathtt{rel}(m) \mid m \in M \,\}$ and the predicate $\mathsf{sync}_{\mathsf{lock}}$ such that for all $\tau \in (\mathsf{Act} \cup \mathbb{S}_{\mathsf{lock}})^*_{\mathsf{idx}}$, we have $\mathsf{sync}_{\mathsf{lock}}(\tau)$ if and only if

$$\forall m \in M \,.\, \tau|_{\{\mathtt{acq}(m), \mathtt{rel}(m)\}_{\mathtt{idx}}} \in \{\langle \mathtt{acq}(m)\mathtt{rel}(m) : i \rangle \mid i \in \mathbb{N}\}^* \cdot (\{\varepsilon\} \cup \{\langle \mathtt{acq}(m) : i \rangle \mid i \in \mathbb{N}\}) \tag{2.1}$$

The *trivial synchronization alphabet* is $(\emptyset, \mathsf{sync}_\top)$ where $\mathsf{sync}_\top(\tau)$ holds for all τ.

The thread template of a parameterized program is a control flow graph.

Definition 2.2. *Given a synchronization alphabet* $(\mathbb{S}, \mathsf{sync})$*, a* control flow graph $G = (\mathbb{L}, \Delta, \ell_{\mathsf{init}}, \ell_{\mathsf{exit}})$ *consists of a finite set of locations* $\mathbb{L}$*, a finite transition relation* $\Delta \subseteq \mathbb{L} \times (\mathsf{Act} \cup \mathbb{S}) \times \mathbb{L}$*, an initial location* $\ell_{\mathsf{init}} \in \mathbb{L}$ *and an exit location* $\ell_{\mathsf{exit}} \in \mathbb{L}$*, with* $\ell_{\mathsf{init}} \neq \ell_{\mathsf{exit}}$*.*

We assume that every location is reachable from ℓ_{init}, and ℓ_{exit} is reachable from every location. Furthermore, for each $a \in \mathsf{Act}$, there exists at most one edge in Δ labeled by a (this assumption is only for notational convenience). We denote the set of actions occurring in G by $\Sigma(G)$ (i.e., $\Sigma(G) \subseteq \mathsf{Act} \cup \mathbb{S}$). We write $\ell \xrightarrow{\tau}_{\Delta} \ell'$ for $\ell, \ell' \in \mathbb{L}$ and $\tau \in \Sigma(G)^*$ if there is a path from ℓ to ℓ' labeled by τ. $\mathcal{T}(G)$ is the language of all traces $\tau \in \Sigma(G)^*$ such that $\ell_{\mathsf{init}} \xrightarrow{\tau}_{\Delta} \ell_{\mathsf{exit}}$ holds.

Definition 2.3. *A* parameterized program $P = ((\mathbb{S}, \mathsf{sync}), G)$ *consists of a synchronization alphabet* $(\mathbb{S}, \mathsf{sync})$ *and a control flow graph* G *(the* thread template*).*

The interleavings of a program $P = ((\mathbb{S}, \mathsf{sync}), G)$ are given by

$$\mathcal{L}(P) = \{\tau \in \mathsf{Act}_{\mathsf{idx}}^* \mid \exists \hat{\tau} \in (\mathsf{Act} \cup \mathbb{S})_{\mathsf{idx}}^* \,.\, \tau = \hat{\tau}|_{\mathsf{Act}_{\mathsf{idx}}} \text{ and } \mathsf{sync}(\hat{\tau}) \text{ and } \forall i \in \mathbb{N} \,.\, \lfloor \hat{\tau} \rfloor_i \in \mathcal{T}(G) \cup \{\varepsilon\}\} \tag{2.2}$$

Every interleaving τ corresponds to a trace $\hat{\tau} \in (\mathsf{Act} \cup \mathbb{S})_{\mathsf{idx}}^*$ which follows the synchronization discipline (e.g., no two threads hold the same lock at the same time), and in which every thread follows a path from the initial to the exit location (or does not run at all). In the interleaving τ itself, the synchronization actions are dropped; their only purpose is to enforce the synchronization discipline. As every thread must reach the exit location, this language of interleavings is suitable for the verification of *postconditions*.

2.2 Reductions

The central idea behind *commutativity* and *Mazurkiewicz equivalence* [23] is that certain actions (of different threads) in an interleaving can be swapped without changing its behaviour. This idea generalizes to *semi-commutativity* [4], in which some pairs of actions can be swapped in one direction (yielding a superset of behaviours) but not vice versa. As a result, Mazurkiewicz equivalence degenerates to a preorder (the relation is not necessarily symmetric).

For a program $P = ((\mathbb{S}, \mathsf{sync}), G)$, we presume a given (semi-)commutativity relation $I \subseteq (\Sigma(G) \cap \mathsf{Act})^2$, not necessarily symmetric, which defines the actions that may be swapped if performed by different threads (actions from the same thread can never be reordered). Formally, the *covering preorder* $\sqsubseteq_I$ is the smallest reflexive-transitive relation over $\mathsf{Act}_{\mathsf{idx}}^*$ with $\rho \langle a : i \rangle \langle b : j \rangle \sigma \sqsubseteq_I \rho \langle b : j \rangle \langle a : i \rangle \sigma$ for all $\rho, \sigma \in \mathsf{Act}_{\mathsf{idx}}^*$, $(a, b) \in I$ and $i \neq j \in \mathbb{N}$.

Definition 2.4. *Given* $L_1, L_2 \subseteq \mathsf{Act}_{\mathsf{idx}}^*$*,* L_1 *is a* Mazurkiewicz reduction *of* L_2 *if* $L_1 \subseteq L_2$ *holds, and for all* $\tau \in L_2$*, there exists some* $\tau' \in L_1$ *with* $\tau \sqsubseteq_I \tau'$*.*

Note that I is a relation over actions in Act only; we do not consider commutativity of synchronization actions in $\mathbb{S}$. The reason is that I is an abstraction of the concrete semantics of actions (which we do not model). In particular, it can be thought of as the information which actions can be swapped without affecting the semantics. By contrast, we explicitly model the semantics of synchronization actions with the predicate sync, and only consider interleavings that obey the semantics. Hence, there is no need to abstract this semantics with I.

3 Natural Reductions

This section formally defines our notion of *natural reductions* based on the syntactic introduction of *atomic blocks* and *global rendez-vous points* in a thread template. The definitions here describe the permissible syntactic changes; they do not guarantee by construction that the reduced program soundly represents the original program. Rather, our definitions specify permissible inputs for the problem of *deciding* the soundness. We discuss the corresponding decision problems (as well as concrete algorithms) in the subsequent sections.

In the following, and for the remainder of the paper, we fix a parameterized program $P = ((\mathbb{S}, \mathsf{sync}), G)$ and a commutativity relation $I \subseteq (\Sigma(G) \cap \mathsf{Act})^2$.

3.1 Atomic Blocks

The following definition captures the syntactic introduction of an atomic block in our control flow graph-based formalism:

Definition 3.1. *An* atomic fusion $\mathcal{F} = (G', (\beta_1, G_{\beta_1}), \ldots, (\beta_n, G_{\beta_n}))$ *for* P *consists of a control flow graph* G'*, distinct actions* $\beta_1, \ldots, \beta_n \in \Sigma(G')$*, and control flow graphs* $G_{\beta_1}, \ldots, G_{\beta_n}$*, where for all* $k = 1, \ldots, n$*, we have*

- $\Sigma(G_{\beta_k}) \subseteq \mathsf{Act} \setminus \{\beta_1, \ldots, \beta_n\}$*, i.e.,* G_{β_k} *contains neither synchronization actions nor any* $\beta_{k'}$*,*
- $\mathcal{T}(G_{\beta_k}) \neq \emptyset$*, i.e.,* G_{β_k} *has at least one path from the initial to the exit location,*
- $G = G'[\beta_1 \mapsto G_{\beta_1}, \ldots, \beta_n \mapsto G_{\beta_n}]$*, i.e.,* G *can be derived from* G' *by inserting each* G_{β_k} *in place of the (unique) edge labeled by the respective* β_k*.*

In the definition above, G' is a new thread template. Each *block symbol* β_k represents an (atomic) execution of an atomic block, whereas G_{β_k} describes the possible paths through the body of the atomic block. There may be multiple paths through an atomic block, due to branching and loops.

Proposition 3.2. *Let* $\mathcal{F} = (G', (\beta_1, G_{\beta_1}), ..., (\beta_n, G_{\beta_n}))$ *be an atomic fusion for* P*, and* $\mu_{\mathcal{F}} : \mathsf{Act} \cup \mathbb{S} \to \wp((\mathsf{Act} \cup \mathbb{S})^*)$ *the morphism with* $\mu_{\mathcal{F}}(\beta_k) = \mathcal{T}(G_{\beta_k})$ *for all* k*, and* $\mu_{\mathcal{F}}(a) = \{a\}$ *for all other* $a \in \mathsf{Act} \cup \mathbb{S}$*. It holds that* $\mathcal{T}(G) = \mu_{\mathcal{F}}(\mathcal{T}(G'))$*.*

We call $\mu_{\mathcal{F}}$ the *fusion morphism* and make use of it throughout the paper.

An atomic fusion induces a new program $P \lhd_{\mathsf{at}} \mathcal{F} := ((\mathbb{S}, \mathsf{sync}), G')$. The interleavings of $P \lhd_{\mathsf{at}} \mathcal{F}$ correspond to interleavings of P where the atomic block is executed without interruption by other threads (the precise path through the atomic block is abstracted by a block symbol β_k). Hence, $\widehat{\mu_{\mathcal{F}}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F}))$ always contains a subset of the interleavings in $\mathcal{L}(P)$. However, it is not a priori clear that this subset of interleavings is representative, i.e., forms a sound reduction.

Definition 3.3. *An atomic fusion $\mathcal{F}$ of P is* sound *(wrt. I) if $\widehat{\mu_{\mathcal{F}}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F}))$ is a Mazurkiewicz reduction (up to I) of $\mathcal{L}(P)$.*

Section 4.1 discusses the problem of *deciding* if an atomic fusion is sound.

3.2 Synchronous Reductions

To support synchronous reductions, that for instance align multiple *rounds* performed by threads, we introduce a notion of *global rendezvous points*, or *sync-points* for short. When a thread encounters a sync-point, it waits until all other threads also reached a sync-point. Only then, all threads can pass the sync-point together; afterwards, each thread continues with its respective computation. The only exception is that a thread that has already reached its exit location need not participate in sync-points (the still-running threads can pass their sync-points without it).

We formalize sync-points as a synchronization alphabet $(\mathbb{S}_\bullet, \mathsf{sync}_\bullet)$, with a unique *sync-point symbol* $\bullet$ (i.e., $\mathbb{S}_\bullet = \{\bullet\}$). To define the synchronization predicate, given a finite set $T = \{t_1, \ldots, t_n\} \subseteq \mathbb{N}$, we let $\langle \bullet : T \rangle$ denote the sequence $\langle \bullet : t_1 \rangle \ldots \langle \bullet : t_n \rangle$. We then define inductively, for all finite sets $T \subseteq \mathbb{N}$:

$$\mathsf{sync}_{\bullet,T}(\tau) \iff \tau = \tau_1\tau_2 \wedge \tau_1 \in ((\mathsf{Act} \times T)^* + \langle \bullet : T \rangle)^* \wedge \exists T' \subsetneq T \,.\, (\mathsf{sync}_{\bullet,T'}(\tau_2) \vee \tau_2 = \varepsilon) \tag{3.1}$$

$$\mathsf{sync}_{\bullet}(\tau) \iff \exists T \subseteq_{\mathsf{fin}} \mathbb{N} \,.\, \mathsf{sync}_{\bullet,T}(\tau) \tag{3.2}$$

The definition of $\mathsf{sync}_\bullet$ states that whenever some set of threads T are running, they can only pass a sync-point $\bullet$ together (as in the sequence $\langle \bullet : T \rangle$). At any point, some threads may terminate (i.e., not perform any further actions); from thereon only the remaining threads T' need to synchronize on sync-points.

As sync-points may be added to programs that already use other synchronization actions, we note that synchronization alphabets $(\mathbb{S}_1, \mathsf{sync}_1), (\mathbb{S}_2, \mathsf{sync}_2)$, with $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$, combine to a synchronization alphabet $(\mathbb{S}_1, \mathsf{sync}_1) \oplus (\mathbb{S}_2, \mathsf{sync}_2) := (\mathbb{S}_1 \cup \mathbb{S}_2, \mathsf{sync}_{12})$, where $\mathsf{sync}_{12}(\tau)$ holds if and only if $\mathsf{sync}_1(\tau|_{(\mathsf{Act} \cup \mathbb{S}_1)_{\mathsf{idx}}})$ and $\mathsf{sync}_2(\tau|_{(\mathsf{Act} \cup \mathbb{S}_2)_{\mathsf{idx}}})$ hold. We define sync-point instrumentation as follows:

Definition 3.4. *A* sync-point instrumentation *of P is a control flow graph G' over $\mathsf{Act} \cup \mathbb{S} \cup \mathbb{S}_\bullet$ with $\mathcal{T}(G) = \mathcal{T}(G')|_{\mathsf{Act} \cup \mathbb{S}}$, and for all $\tau_1, \tau_2 \in \mathcal{T}(G')$, we have that $\tau_1|_{\mathsf{Act}} = \tau_2|_{\mathsf{Act}}$ implies $\tau_1 = \tau_2$.*

A sync-point instrumentation is thus a modified thread template that inserts the sync-point symbol $\bullet$ at various locations. The injectivity condition for projection means that for every $\tau \in \mathcal{T}(G)$, there is a *unique* way to insert sync-points in it and derive a trace in $\mathcal{T}(G')$. One way to ensure this injectivity syntactically is to construct G' from $G = (\mathbb{L}, \Delta, \ell_{\mathsf{init}}, \ell_{\mathsf{exit}})$ by inserting sync-points at a set of locations $M \subseteq \mathbb{L}$. I.e., we define $G' = (\mathbb{L} \uplus \{\hat{\ell} \mid \ell \in M\}, \Delta', \ell_{\mathsf{init}}, \ell_{\mathsf{exit}})$ with

$$\Delta' = \{\, (\ell, a, \ell') \in \Delta \mid \ell \notin M \,\} \cup \{\, (\ell, \bullet, \hat{\ell}) \mid \ell \in M \,\} \\ \cup \{\, (\hat{\ell}, a, \ell') \mid \ell \in M \wedge (\ell, a, \ell') \in \Delta \,\} \,.$$

We add a copy $\hat{\ell}$ of every location $\ell \in M$, such that ℓ retains its incoming edges, ℓ and $\hat{\ell}$ are connected by a sync-point, and all outgoing edges of ℓ are moved to $\hat{\ell}$ (i.e., after the sync-point). This syntactic sync-point insertion at locations M is sufficient to specify reductions e.g. for distributed protocols based on *rounds*, as the point where a thread advances from one round to another can be identified syntactically in the code. Moreover, sync-points can faithfully implement lockstep execution for thread templates with a single loop. However, if the template contains multiple loops, the lockstep scheduler cannot be modeled precisely with sync-points, due to the injectivity condition.

Observation 3.5. *Any G' constructed as described above is a sync-point instrumentation of P.*

A given sync-point instrumentation G' induces a new program $P \lhd_\bullet G' := ((\mathbb{S}_{+\bullet}, \mathsf{sync}_{+\bullet}), G')$ with the combined synchronization alphabet $(\mathbb{S}_{+\bullet}, \mathsf{sync}_{+\bullet}) := (\mathbb{S}, \mathsf{sync}) \oplus (\mathbb{S}_\bullet, \mathsf{sync}_\bullet)$. The additional synchronization enforced by the sync-points yields a subset of interleavings: $\mathcal{L}(P \lhd_\bullet G') \subseteq \mathcal{L}(P)$.

Definition 3.6. *A sync-point instrumentation G' of P is* sound *(with respect to I) if $\mathcal{L}(P \lhd_\bullet G')$ is a Mazurkiewicz reduction (up to I) of $\mathcal{L}(P)$.*

Section 4.2 discusses the problem of *deciding* whether a given sync-point instrumentation is sound.

3.3 Combining Atomic Blocks and Synchronous Reductions

We combine atomic blocks and synchronous reductions in a straightforward manner in order to define the class of *natural reductions*.

Definition 3.7. *A* natural reduction specification *$(\mathcal{F}, G')$ for P consists of an atomic fusion $\mathcal{F}$ of P and a sync-point instrumentation G' of $P \lhd_{\mathsf{at}} \mathcal{F}$.*

Note that by this definition, atomic blocks may not contain sync-points. There would be no point for a thread to wait on all other threads at a sync-point, while simultaneously preventing other threads from making progress and reaching sync-points (as that would interrupt the atomic block).

The notion of soundness follows directly from soundness of atomic fusions and sync-point instrumentations:

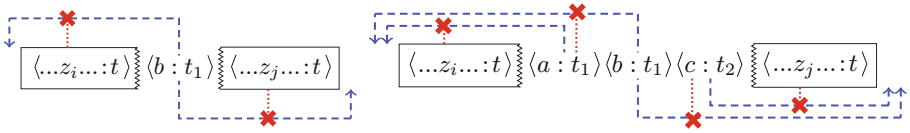
Definition 3.8. *A natural reduction specification $(\mathcal{F}, G')$ for P is* sound *(wrt. I) if $\widehat{\mu_{\mathcal{F}}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F} \lhd_\bullet G'))$ is a Mazurkiewicz reduction (up to I) of $\mathcal{L}(P)$.*

4 Deciding the Soundness Problem

We first consider the problems of deciding soundness for atomic fusions and sync-point instrumentations separately, in Sect. 4.1 and Sect. 4.2, respectively. Section 4.3 then proposes a combined algorithm for deciding soundness of natural reduction specifications.

4.1 Deciding Soundness of Atomic Blocks

We turn to the problem of deciding whether a given atomic fusion is sound. The complexity of this problem, like many problems over parameterized programs, depends crucially on the mechanisms allowed for synchronization between threads [9]. In fact, we link its complexity to the *coverability problem* for configurations of arbitrary width.



(a) $(z_i, b) \notin I, (b, z_j) \notin I$. (b) $(z_i, a) \notin I, (a, b) \in \mathsf{po}, (b, c) \notin I, (c, z_j) \notin I$

Fig. 4. Action b cannot escape to the left/right of the atomic block $z_1 \dots z_m$.

We begin by examining the notion of a *sound* atomic fusion $\mathcal{F}$ of P more closely. In essence, it states that every interleaving of the original program P is covered by some interleaving in which the actions inside an atomic block execute without interruption by other threads. To ensure this, we must check that each action that may be interleaved inside the atomic block can be swapped either before all actions of the atomic block (it *escapes to the left*) or after all actions of the atomic block (it *escapes to the right*).[2] There can be different obstacles that prevent an action from escaping an atomic block $z_1 \dots z_m$, thus causing the atomic fusion to be unsound. In the simplest case, illustrated in Fig. 4a, an action b does not commute to the left of some z_i nor to the right of some z_j, with $i < j$. Then, the kind of interleaving shown in Fig. 4a has no representative where $z_1 \dots z_m$ executes atomically, and the given atomic fusion is unsound.

However, the problem can be more complex. For instance, in Fig. 4b, b commutes to the left of all $z_1, \dots, z_m$, but a previous action a of the same thread does not commute with z_i. As we cannot swap $\langle b : t_1 \rangle$ with $\langle a : t_1 \rangle$, in this scenario, $\langle b : t_1 \rangle$ cannot escape to the left. On the other hand, $\langle b : t_1 \rangle$ cannot escape to the right either, due to the presence of an action $\langle c : t_2 \rangle$ such that

[2] This is a shift from the classical perspective [21], which reasons about *moving* the actions *of the atomic block*. We discuss limitations of that perspective in Sect. 5.

$(b, c) \notin I$. In order for $\langle b : t_1 \rangle$ to escape to the right, $\langle c : t_2 \rangle$ would also have to escape *to the right*, but we have $(c, z_j) \notin I$.

As shown here, even if individual actions interleaved in an atomic block can escape, in general we have to reason about complex chains of dependencies (e.g., if $\langle a : t_1 \rangle$ can only escape to the right, then $\langle b : t_1 \rangle$ must also escape to the right, and hence the same applies to $\langle c : t_2 \rangle$). Whether such a chain of dependencies causes unsoundness however crucially depends on whether the actions $\langle a : t_1 \rangle, \langle b : t_1 \rangle, \langle c : t_2 \rangle$ can actually be interleaved inside the block in this manner, or whether synchronization prevents this. For instance, if actions b and c are protected by the same lock, the issue above could not occur (t_1 and t_2 cannot both hold the lock at the same time). The soundness of atomic fusions is thus tightly linked to the question of *coverability* under synchronization:

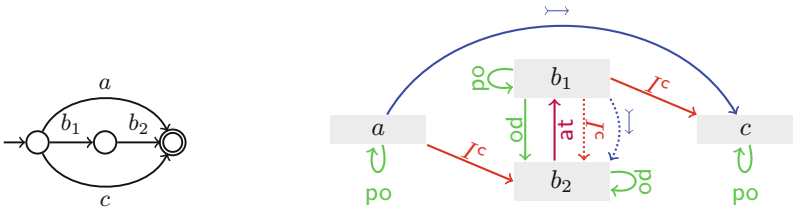
Definition 4.1. (Coverability). *Let $C = [\ell_1, \ldots, \ell_n]$ be a finite multiset of locations of P. The configuration C is* coverable *in P if there exist a trace $\hat{\tau} \in (\Sigma(G) \cup \mathbb{S})^*_{\mathsf{idx}}$ with $\mathsf{sync}(\hat{\tau})$, and (distinct) $t_1, \ldots, t_n \in \mathbb{N}$ with $\ell_{\mathsf{init}} \xrightarrow{\lfloor \hat{\tau} \rfloor_{t_i}}_{\Delta} \ell_i$.*

Theorem 4.2. *For every synchronization alphabet, the coverability problem over programs with this synchronization alphabet is polynomial-time reducible to the problem of deciding whether a given atomic fusion of a given program is unsound wrt. a given commutativity relation.*

Proof (sketch). Given configuration $C = [\ell_1, \ldots, \ell_n]$, we let n fresh actions $a_1, \ldots, a_n$ form a dependency chain with actions b_1, b_2 belonging to an atomic block (i.e., $(b_1, a_1) \notin I$, $(a_i, a_{i+1}) \notin I$, $(a_n, b_2) \notin I$); all other actions commute. We insert the actions a_i at the corresponding locations ℓ_i, and add the atomic block $b_1 b_2$ as a branch from the initial to the exit location. The atomic fusion is unsound if and only if a trace containing $\langle b_1 : i_0 \rangle \langle a_1 : i_1 \rangle \ldots \langle a_n : i_n \rangle \langle b_2 : i_0 \rangle$ is allowed by sync, which holds if and only if C is coverable.

In particular, the width of the configurations in the coverability problem corresponds to the size of simple cycles in the complement of I.

As a corollary of Theorem 4.2, deciding soundness of atomic fusions for programs with powerful synchronization mechanisms such as *broadcast* has non-elementary complexity [9]. Even for programs with simpler synchronization mechanisms, such as boolean variables, the problem is coNP-hard. This motivates us to consider an abstract view of the program, which ignores the semantics of such synchronization actions, and treats them like any other ordinary actions. This yields an over-approximation of the program, which contains some interleavings that would be ruled out by the synchronization mechanism. Reasoning on this abstract view of the program allows us to *certify* the soundness of atomic fusions (and later, of natural reductions generally): any atomic fusion that is sound wrt. the abstract view is also sound wrt. a more concrete view. A determination that an atomic fusion is *unsound* does not transfer to the concrete level: the perceived unsoundness may be an artifact of the abstraction. Yet, the abstract view provides for an efficient and predictable decision procedure.



(a) Example of a thread template. The actions $b_1 b_2$ should be fused into an atomic block.

(b) Relations between the actions. Solid lines for $I_1 = \{a, b_1, b_2, c\}^2 \setminus \{(a, b_2), (b_1, c)\}$; solid and dotted lines for $I_2 = I_1 \setminus \{(b_1, b_2)\}$.

Fig. 5. Illustration of Proposition 4.3 and relevant relations between actions.

Consequently, we assume for the remainder of this section that our program P uses the trivial synchronization alphabet $(\emptyset, \mathsf{sync}_\top)$. Furthermore, let $\mathcal{F} = (G', (\beta_1, G_{\beta_1}), \ldots, (\beta_n, G_{\beta_n}))$ be an atomic fusion of P. We present an algorithm to decide whether the given atomic fusion $\mathcal{F}$ is sound wrt. I.

The algorithm is based on detecting the different kinds of "obstacles" that might prevent an action interleaved inside an atomic block from escaping, as discussed above (and illustrated in Fig. 4). For $a, b \in \Sigma(G)$, we write $(a, b) \in \mathsf{po}$ (*program order*) if there exists a trace of the form $\rho\iota\sigma \in \mathcal{T}(G)$, where ι begins with a and ends with b (possibly $a = \iota = b$). $I^{\mathsf{c}} := \Sigma(G)^2 \setminus I$ denotes the complement of I. We write $(a, b) \in \mathsf{at}$ if there exists $\rho b\iota a\sigma \in \mathcal{T}(G_{\beta_k})$, for some β_k (the reverse program order within atomic sections). We define $\rightarrowtail$ as the relation $I^{\mathsf{c}} \circ \big((\mathsf{po} \cup \mathsf{at}) \circ I^{\mathsf{c}}\big)^+$, where $\circ$ denotes relational composition. The following result formalizes the idea that soundness of $\mathcal{F}$ corresponds to the non-existence of the kind of dependence chains $\langle z_i : t\rangle, \langle a : t_1\rangle, \langle b : t_1\rangle, \langle c : t_2\rangle, \langle z_j : t\rangle$ as in the example of Fig. 4b.

Proposition 4.3. *The atomic fusion $\mathcal{F}$ is sound if and only if there do not exist $z_1 \ldots z_m \in \mathcal{T}(G_{\beta_k})$ (for some k) and $i < j \in \{1, \ldots, m\}$ such that $z_i \rightarrowtail z_j$ holds.*

Proof (sketch). If $z_i \rightarrowtail z_j$ holds, we construct an interleaving in which $z_1 \ldots z_m$ cannot be made atomic, except possibly by breaking apart another atomic block. Conversely, under the assumption that no $z_1 \ldots z_m$ with $z_i \rightarrowtail z_j$ exists, we show that every interleaving can be transformed into one where all atomic blocks are represented by their respective block symbol β_k.

Example 4.4. Consider the thread template in Fig. 5a, and the atomic fusion that combines $b_1 b_2$ into an atomic block. By Proposition 4.3, this atomic fusion is sound if and only if $b_1 \not\rightarrowtail b_2$ holds. As illustrated in Fig. 5b, this is the case for $I_1 = \{a, b_1, b_2, c\} \setminus \{(a, b_2), (b_1, c)\}$.

However, if b_1 and b_2 do not commute, as for $I_2 = I_1 \setminus \{(b_1, b_2)\}$, we have $(b_1, b_2) \notin I_2$, $(b_2, b_1) \in \mathsf{at}$, and $(b_1, b_2) \notin I_2$. Hence, it holds that $b_1 \rightarrowtail b_2$. The proof of Proposition 4.3 constructs the interleaving $\langle b_1 : 1\rangle\langle b_1 : 2\rangle\langle b_2 : 2\rangle\langle b_2 : 1\rangle$.

This interleaving cannot be transformed (through commutations allowed by I_2) into an interleaving where both thread 1 and thread 2 execute $b_1 b_2$ atomically.

Using Proposition 4.3, the following algorithm decides soundness of $\mathcal{F}$. As the bodies G_{β_k} of atomic blocks may contain complex control flow including loops, we cannot simply enumerate all possible $z_1 \dots z_m \in \mathcal{T}(G_{\beta_k})$. Hence, the algorithm reasons on the level of strongly connected components (SCCs) of G_{β_k}.

Algorithm 4.5. *Execute the following steps:*

(1) Compute the graph $(\Sigma(G), \rightarrowtail)$. *Perform the subsequent steps for each* β_k.
(2) Compute the SCCs of the edges of G_{β_k}, *along with the reachability relation* $\preceq$ *between SCCs (a partial order).*
(3) Compute $\min(a)$ *for each* $a \in \Sigma(G)$, *the set of* $\preceq$*-minimal SCCs of* G_{β_k} *containing some edge labeled by an action* $b \in \Sigma(G_{\beta_k})$ *with* $(b, a) \notin I$.
(4) Compute $\max(a)$ *for each* $a \in \Sigma(G)$, *the set of* $\preceq$*-maximal SCCs of* G_{β_k} *containing some edge labeled by an action* $b \in \Sigma(G_{\beta_k})$ *with* $(a, b) \notin I$.
(5) Check if there exist SCCs S_1, S_2 *of* G_{β_k} *such that*

- $S_1 \preceq S_2$, *and if* $S_1 = S_2$ *then* S_1 *is a non-trivial SCC*[3], *and*
- *the graph* $(\Sigma(G), \rightarrowtail)$ *has an edge from some* a *to some* b *with* $S_1 \in \min(a)$ *and* $S_2 \in \max(b)$.

If such S_1, S_2 *exist (for some* β_k*), then* $\mathcal{F}$ *is unsound. Otherwise,* $\mathcal{F}$ *is sound.*

Theorem 4.6. *Algorithm 4.5 decides soundness of atomic fusions, and terminates in polynomial time.*

4.2 Deciding Soundness of Synchronous Reductions

Next, we consider the soundness problem for synchronous reductions. Hence, let $P = ((\emptyset, \mathsf{sync}_\top), G)$ once again be a program over the trivial synchronization alphabet, and let G' be a sync-point instrumentation of P. We are interested in deciding whether G' is sound, i.e., whether $\mathcal{L}(P \lhd_\bullet G')$ is a Mazurkiewicz reduction of $\mathcal{L}(P)$, up to the commutativity relation I. Recall that $\mathcal{L}(P \lhd_\bullet G')$ is given by

$$\mathcal{L}(P \lhd_\bullet G') = \{\tau \in \mathsf{Act}_{\mathsf{idx}}^* \mid \exists \hat{\tau} \in (\mathsf{Act} \cup \mathbb{S}_\bullet)_{\mathsf{idx}}^* . \tau = \hat{\tau}|_{\mathsf{Act}_{\mathsf{idx}}} \text{ and } \mathsf{sync}_\bullet(\hat{\tau}) \text{ and } \forall i \in \mathbb{N} . \lfloor \hat{\tau} \rfloor_i \in \mathcal{T}(G') \cup \{\varepsilon\}\} \tag{4.1}$$

By the definition of $\mathsf{sync}_\bullet$ (Eq. (3.1)), the underlying traces $\hat{\tau}$ with $\mathsf{sync}_\bullet(\hat{\tau})$ can be split into "phases" consisting of actions from Act, separated by occurrences of sequences $\langle \bullet : t_1 \rangle \dots \langle \bullet : t_n \rangle$, where $t_1, \dots, t_n$ are the threads that are still

[3] A non-trivial SCC must either contain at least two elements, or its single element must have a self-loop.

running. In order for the sync-point reduction to be sound, it must be possible to bring every interleaving of the original program P into such a "phase form" by swapping commuting actions and then inserting occurrences of $\langle \bullet : t_1 \rangle \dots \langle \bullet : t_n \rangle$ in the right places. Based on this idea, we define a "phase order" on actions:

Definition 4.7. *The* phase order *is the relation* $\mathsf{pho} \subseteq \Sigma(G)^2$*, with* $(a, b) \in \mathsf{pho}$ *if there exists a trace* $\hat{\tau} \in (\mathsf{Act} \cup \mathbb{S}_\bullet)^*_{\mathsf{idx}}$ *with* $\mathsf{sync}_\bullet(\hat{\tau})$ *and* $\hat{\tau}|_{\mathsf{Act}_{\mathsf{idx}}} \in \mathcal{L}(P \lhd_\bullet G')$ *such that for some* $i \neq j$*, we have* $\lfloor \hat{\tau} \rfloor_i = \tau_1 a \sigma_1$*,* $\lfloor \hat{\tau} \rfloor_j = \tau_2 b \sigma_2$*, and* $|\tau_1|_\bullet < |\tau_2|_\bullet$*.*

Intuitively, the phase order captures that a can occur in a strictly later phase than b, in some execution of the instrumented program. As we are considering programs without synchronization (other than the newly-introduced sync-point $\bullet$), we can break this condition down further:

Observation 4.8. *We have* $(a, b) \in \mathsf{pho}$ *if and only if there exist two traces* $\rho_1 a \sigma_1, \rho_2 b \sigma_2 \in \mathcal{T}(G')$ *with* $|\rho_1|_\bullet < |\rho_2|_\bullet$*.*

Given pho, it is straightforward to decide soundness of the reduction:

Proposition 4.9. *The sync-point instrumentation* G' *is sound if and only if* $\mathsf{pho} \subseteq I^{-1}$ *holds.*

Proof (sketch). If there exists a pair $(a, b) \in \mathsf{pho} \setminus I^{-1}$, we construct an interleaving (containing a and b), and use the injectivity condition in Definition 3.4 to show that no representative for this interleaving exists in $\mathcal{L}(P \lhd_{\mathsf{at}} G)$.

Conversely, if $\mathsf{pho} \subseteq I^{-1}$ holds, we show that every $\tau \in \mathcal{L}(P)$ does have a representative in $\mathcal{L}(P \lhd_{\mathsf{at}} G)$.

The fact that the above proof uses the injectivity condition of Definition 3.4 only for one direction of the equivalence implies that even for an extended class of sync-point instrumentations that do not satisfy injectivity, $\mathsf{pho} \subseteq I^{-1}$ implies soundness of the instrumentation. Our approach thus remains sound in this extended class, though not complete.

To check soundness of a sync-point instrumentation, the only remaining difficulty is to compute pho. In a program model without synchronization, this is possible in polynomial time, using the following approach:

Algorithm 4.10. *We compute* pho *as follows:*

(1) Compute $\min_\bullet(a) := \min\{\, |\tau|_\bullet \mid \exists \sigma \,.\, \tau a \sigma \in \mathcal{T}(G') \,\}$ *for each action* a*, by a shortest-path computation on* G'*, where the initial location of* G' *is the source, locations enabling* a *are the targets, and the weight of an edge is 1 if labeled by* $\bullet$ *and 0 otherwise.*

(2) Compute $\max_\bullet(a) := \max\{\, |\tau|_\bullet \mid \exists \sigma \,.\, \tau a \sigma \in \mathcal{T}(G') \,\} \in \mathbb{N} \cup \{\infty\}$ *for each action* a*.*

To do so, consider all loop-free paths from the initial location to a location enabling a*. Determine if any such path can be pumped with a loop that contains* $\bullet$ *(the loop may also include* a *itself). If so, then* $\max_\bullet(a)$ *is* ∞*. Otherwise,* $\max_\bullet(a)$ *is the maximum weight of the loop-free paths, where the weight of an edge is 1 if labeled by* $\bullet$ *and 0 otherwise.*

(3) Check for all pairs (a, b) *whether* $\min_\bullet(a) < \max_\bullet(b)$. *If so, then* $(a, b) \in \mathsf{pho}$ *holds. Otherwise, we have* $(a, b) \notin \mathsf{pho}$.

Theorem 4.11. *Soundness of sync-point instrumentations (in programs with the trivial synchronization alphabet) can be decided in polynomial time.*

4.3 Deciding Soundness of Natural Reductions

Let P be a program, and $(\mathcal{F}, G')$ be a natural reduction specification for P. When deciding whether $(\mathcal{F}, G')$ is sound, it is of course desirable to make use of the algorithms introduced in the previous subsections. And indeed, the following observation gives us some hope that soundness checking of complicated reductions can be decomposed into smaller steps.

Proposition 4.12. *Let* $L_1, L_2, L_3 \subseteq \mathsf{Act}^*_{\mathsf{idx}}$ *such that* $L_1 \subseteq L_2 \subseteq L_3$. *Then* L_1 *is a Mazurkiewicz reduction of* L_3 *if and only if* L_1 *is a Mazurkiewicz reduction of* L_2, *and* L_2 *is a Mazurkiewicz reduction of* L_3.

Recall that $(\mathcal{F}, G')$ is sound (wrt. I) if $\widehat{\mu_\mathcal{F}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F} \lhd_\bullet G'))$ is a Mazurkiewicz reduction (up to I) of $\mathcal{L}(P)$. As an instantiation of the above result, we get that $(\mathcal{F}, G')$ is sound if and only if

(1) $\widehat{\mu_\mathcal{F}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F} \lhd_\bullet G'))$ is a Mazurkiewicz reduction up to I of $\widehat{\mu_\mathcal{F}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F}))$,
(2) and $\widehat{\mu_\mathcal{F}}(\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F}))$ is a Mazurkiewicz reduction up to I of $\mathcal{L}(P)$.

Condition (2) directly corresponds to soundness of the atomic fusion $\mathcal{F}$, and can therefore be decided using Algorithm 4.5. The situation is more complex for condition (1), as (due to the presence of $\widehat{\mu_\mathcal{F}}$) it does not correspond directly to soundness of G'. Conceptually, it is not even immediately clear what soundness of G' would mean: $\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F})$ and $\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F} \lhd_\bullet G')$ are not languages over $\Sigma(G)$; they contain the block symbols introduced by $\mathcal{F}$ (in addition to a subset of actions from $\Sigma(G)$). However, we can define the following commutativity relation, where $\mathcal{F} = (G'', (\beta_1, G_{\beta_1}), \ldots, (\beta_n, G_{\beta_n}))$, and a, b range over $\Sigma(G)$:

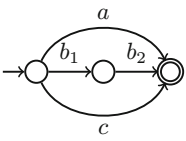
$$\begin{aligned}\tilde{I} := I \cap \Sigma(G')^2 \\ \cup \{\, (a, \beta_k) \mid \forall b \in \Sigma(G_{\beta_k}) \,.\, (a, b) \in I \,\} \\ \cup \{\, (\beta_k, b) \mid \forall a \in \Sigma(G_{\beta_k}) \,.\, (a, b) \in I \,\} \\ \cup \{\, (\beta_{k_1}, \beta_{k_2}) \mid \forall a \in \Sigma(G_{\beta_{k_1}})\ \forall b \in \Sigma(G_{\beta_{k_2}}) \,.\, (a, b) \in I \,\}\end{aligned}$$

For this commutativity relation, we show:

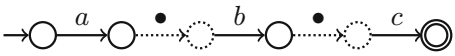
Lemma 4.13. *Condition (1) holds if and only if* $\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F} \lhd_\bullet G')$ *is a Mazurkiewicz reduction up to* $\tilde{I}$ *of* $\mathcal{L}(P \lhd_{\mathsf{at}} \mathcal{F})$.

This lemma allows us to decide condition (1) using the approach from Sect. 4.2. We conclude:

Theorem 4.14. *The soundness of a natural reduction specification for a program without synchronization is decidable in polynomial time.*



(a) The atomic fusion of $b_1 b_2$ is sound wrt. $I = \{a, b_1, b_2, c\}^2 \setminus \{(a, b_2), (b_1, c)\}$.



(b) The dotted sync-point instrumentation is sound wrt. $I = \{a, b, c\}^2 \setminus \{(b, b), (c, c)\}$.

Fig. 6. Examples for incompleteness of mover reasoning.

Observe that, in the case of programs with synchronization, our hardness result for atomic blocks carries over to the more general problem of soundness of natural reduction specifications.

Corollary 4.15. *For every synchronization alphabet, the coverability problem is polynomial-time reducible to the soundness of natural reduction specifications.*

Proof. Follows directly from Theorem 4.2.

5 On the Incompleteness of Mover Reasoning

There is a wealth of literature on sound reductions given by atomic blocks. This literature builds on the seminal work of Lipton [21] and the idea of *movers*. We show that, while sound, mover reasoning is incomplete (i.e., fails to admit some sound atomic fusions). Completeness requires moving from mostly local reasoning (as in Lipton's rule) to a global view (as in our algorithm). A similar observation also holds for sync-points.

We begin by introducing movers and Lipton's rule for atomic blocks, reformulated in our formal setting. As before, we assume a program $P = ((\emptyset, \mathsf{sync}_\top), G)$ and a commutativity relation $I \subseteq \Sigma(G)^2$.

Definition 5.1. ([21]). *An action $a \in \Sigma(G)$ is:*

- *a* left-mover *if $(b, a) \in I$ for all $b \in \Sigma(G)$,*
- *a* right-mover *if $(a, b) \in I$ for all $b \in \Sigma(G)$,*
- *a* both-mover *if it is both a left-mover and a right-mover,*
- *and a* non-mover *if it is neither a left-mover nor a right-mover.*

Lipton's rule is the following result:

Proposition 5.2. ([21]). *An atomic fusion $(G', (\beta_1, G_{\beta_1}), ..., (\beta_n, G_{\beta_n}))$ is sound if every $\tau \in \mathcal{T}(G_{\beta_k})$ (for every k) can be written as $\tau = \tau_r \, a \, \tau_l$, for a sequence of right-movers τ_r, some action a and a sequence of left-movers τ_l.*

While Lipton's rule is sufficient to ensure soundness, it is not a necessary condition, as demonstrated by the following example.

Example 5.3. Consider the thread template in Fig. 6a, over $\Sigma = \{a, b_1, b_2, c\}$. Assume the given commutativity relation is $I = \Sigma^2 \setminus \{(a, b_2), (b_1, c)\}$.

Then, b_1 is a left-mover (but not a right-mover), whereas b_2 is a right-mover (but not a left-mover). Hence, Lipton's rule is not sufficient to conclude that $b_1 b_2$ can be fused in an atomic block. Yet, it *is* sound to make $b_1 b_2$ atomic. In any interleaving, all occurrences of a can be commuted to the beginning, and all occurrences of c can be commuted to the end of the interleaving. The remaining interleaved occurrences of b_1, b_2 (from different threads) commute freely against each other, and can be reordered such that each $\langle b_1 b_2 : i \rangle$ executes atomically.

Going further, if we consider $I' = I \setminus \{(b_1, b_2)\}$, the mover properties of all actions remain unchanged. Yet, the atomic fusion of $b_1 b_2$ is unsound wrt. I': the interleaving $\langle b_1 : 1 \rangle \langle b_1 : 2 \rangle \langle b_2 : 1 \rangle \langle b_2 : 2 \rangle$ has no representative in which $\langle b_1 b_2 : 1 \rangle$ and $\langle b_1 b_2 : 2 \rangle$ execute atomically. Hence, mover properties are generally insufficient to characterize sound atomic fusions (even beyond Lipton's rule).

By contrast, our algorithm is sound and complete; but it requires a global (rather than local) view that takes e.g. control flow into account. In particular, Example 4.4 explains how our algorithm handles the above example.

As a notable aside, in the case where I is symmetric (every left- or right-mover is a both-mover), Lipton's rule is indeed complete for parameterized programs.

Proposition 5.4. *Let I be symmetric. If $\mathcal{F} = (G', (\beta_1, G_{\beta_1}), \ldots, (\beta_n, G_{\beta_n}))$ is sound wrt. I, then all $\tau \in \mathcal{T}(G_{\beta_k})$ (for every k) can be written as $\tau = \tau_r \, a \, \tau_l$, for a sequence of both-movers τ_r, some action a, and a sequence of both-movers τ_l.*

Proof (sketch). Assuming an atomic block that is not in Lipton form, we construct an interleaving of four threads (two of which execute the atomic block), such that no representative of this interleaving executes the block atomically. The construction makes use of the fact that the program is parameterized (in the number of threads) and that there is no synchronization between threads.

We conclude this section by showing that mover reasoning is similarly unable to provide a characterization of sound sync-point instrumentations.

Example 5.5. Consider the thread template and sync-point instrumentation in Fig. 6b, over actions $\Sigma = \{a, b, c\}$. This sync-point instrumentation is sound wrt. the commutativity relation $I = \Sigma^2 \setminus \{(b, b), (c, c)\}$. However, this fact cannot be justified purely with mover properties. In this example, we have that a is a both-mover, whereas b and c are non-movers. If we consider a different commutativity relation $I' = \Sigma^2 \setminus \{(b, c), (c, b)\}$, the mover properties remain unchanged, yet the sync-point instrumentation is unsound wrt. I'.

Unlike for atomic blocks, the inability of mover properties to characterize sound sync-point instrumentations thus persists even in the case of symmetric commutativity (both I and I' in the example above are symmetric).

6 Reduction Soundness in the Presence of Locks

Ignoring all synchronization between threads is a rather coarse abstraction. One may be tempted to allow for at least some light-weight synchronization mecha-

nisms to be considered during reduction checking. For instance, locks (mutexes) are such a light-weight mechanism, compared to more powerful synchronization via e.g. (global) boolean variables, broadcast, etc. Locks are also ubiquitously used in concurrent programs. However, we show in this section that allowing locks immediately leads to coNP-hardness for the soundness problem of natural reductions, even in the simplest cases.

Atomic Blocks. As shown in Theorem 4.2, coverability over programs with a certain synchronization alphabet is polynomial-time reducible to the *un*soundness of atomic fusions (with only a single atomic block) for the same class of programs. For the case of synchronization via locks, we show:

Theorem 6.1. *The coverability problem over programs with locks is NP-hard.*

Proof (sketch). We reduce 3-Sat to reachability in a bounded-thread program with locks, and then to coverability in a parameterized program with locks. For the latter step, we construct a thread template where each thread of the bounded-thread program is a separate branch, protected by a dedicated lock to ensure this branch can only be taken by one thread instance. For the reduction from 3-Sat, we introduce one lock m_i^l for each pair of a literal l over the propositional variables and a clause i. There is one thread for each clause i, which consists of choosing one of the literals l in the clause to satisfy, acquiring the lock m_i^l, checking that no thread for some other clause j has made a contradictory choice (i.e., $m_j^{\neg l}$ must be free), and reaching some location ℓ_i. The configuration of all ℓ_i is reachable if and only if the conjunction of all clauses is satisfiable.

Corollary 6.2. *Atomic fusion soundness in programs with locks is* coNP*-hard.*

Proof. Follows directly from Theorem 4.2 and Theorem 6.1.

The corresponding upper-bound is (to our knowledge) an open question.

Furthermore, we note that previous work [12] has shown coNP-hardness for a special case of atomic fusion soundness (*conflict serializability*) in programs with a bounded number of threads (even without synchronization). Interestingly, this previous result provides for an alternative proof of Corollary 6.2: We can reduce the soundness problem in parameterized programs to the bounded case, by using dedicated locks to limit the number of running threads. Hence, while the shift from a bounded number of threads to parameterized programs significantly simplifies the soundness problem in the absence of synchronization, lock synchronization yields back the hardness.

Synchronous Reductions. For synchronous reductions (resp. sync-point instrumentations), we have not shown a general result relating coverability (for configurations of arbitrary width) and reduction checking. In fact, the algorithm in Sect. 4.2 at first suggests that only pairwise checks are needed (for the computation of the phase-order). Yet, the difficulty lies in the fact that sync-points themselves are synchronization actions, and may interfere with other synchronization mechanisms such as locks. Using this observation, we show:

Theorem 6.3. *The coverability problem of programs with locks is polynomial-time reducible to the sync-point soundness problem (of programs with locks).*

Proof (sketch). Given a program P and a configuration $C = [\ell_1, \ldots, \ell_n]$ for which to decide coverability, we modify the thread template as follows. We take fresh locks $m_1, \ldots, m_n$, add a fresh location ℓ'_{exit}, and chose ℓ'_{exit} as the new exit location. From each location $\ell_i \in C$, we add a subgraph that first acquires m_i, then branches over all $j \neq i$ and checks that m_j is still free (by acquiring and releasing it), and then finally releases m_i, reaching ℓ'_{exit}. For the sync-point instrumentation, we add a single sync-point $\bullet$ immediately before the final step (releasing m_i). We note that C is coverable (in P) if and only if it is reachable. Any interleaving τ of P reaching a sub-configuration of C is an interleaving of the modified program (all threads can reach ℓ'_{exit}). However, in an interleaving reaching exactly C, the sync-point instrumentation forces at least one of the n threads to deadlock when trying to acquire some m_j (and hence, all other threads are blocked at the sync-point). Thus, the sync-point instrumentation is sound (wrt. any I) if and only if C is not coverable.

Using Partial Information About Locks. We conclude this section with a short outlook on how soundness checking for natural reductions can nevertheless benefit from (possibly incomplete, but sound) information about locks. Suppose that we have information available about a set of locks M_ℓ that a thread *must* hold whenever it is at location ℓ. Such information could for instance be derived from a static analysis, or a thread-modular proof given to a deductive verifier by a user. Then one can soundly declare that outgoing actions a_ℓ of ℓ and $a_{\ell'}$ of ℓ' commute whenever the sets M_ℓ and $M_{\ell'}$ overlap.

More generally, any kind of *thread-modular invariants* $\varphi_\ell, \varphi_{\ell'}$ that hold whenever a thread is at location ℓ resp. ℓ' can be used to increase commutativity, by determining if actions $a_\ell, a_{\ell'}$ commute *under the assumption* $\varphi_\ell \wedge \varphi_{\ell'}$ [11]. The idea for locks described above corresponds to the special case that $\varphi_\ell \wedge \varphi_{\ell'} \equiv \bot$ (it is impossible that both threads hold the same lock), under which the actions vacuously commute. The resulting extended commutativity relation $I' \supseteq I$ can increase the power of both mover reasoning [21] as well as our algorithms.

7 Related Work

Sound reductions proposed by atomic blocks have been studied and implemented in many deductive proof systems for concurrent and distributed systems [2,6,14,18,19,21,22]. These approaches are based on Lipton's reductions [21] and are thus inherently incomplete (see Sect. 5). In principle, our sound and complete algorithm for atomic blocks may serve as a replacement for the current implementations in these proof systems. Yet, some hurdles remain. Practical tools such as Civl [19] typically support specifications via `assert` statements, whereas our approach currently only targets verification of postconditions. While we do not expect the extension to `assert` statements to change the complexity picture significantly, it requires non-trivial conceptual adjustments, in particular

when the execution of a single atomic block is not guaranteed to terminate (due to loops or recursion, see e.g. [15]).

Similarly, sound but possibly incomplete techniques for proving soundness of synchronous reductions have been studied in [16,20]. These techniques are also based on the mover types from Lipton's reduction theory.

Checking *conflict serializability* for transactions can be seen as an instance of deciding soundness of atomic blocks, where the commutativity relation is fixed: read accesses commute against one another, and (read or write) accesses to different shared variables also commute. The complexity of checking serializability has been studied in [1,12], but these results do not extend to arbitrary commutativity relations as in our paper.

On the side of algorithmic verification (and more classically, finite-state model checking), *partial order reduction* algorithms [17] are employed to automatically compute reductions that are sound by construction. The works of Farzan et al. [10,11] define an algorithmic verification framework for concurrent programs that includes computing sound reductions. The syntactic representation of these reductions is somewhat complex, which is fine in the context of a fully automated, non-interactive proof. In our paper, we look at classes of reductions which are humanly readable and use simple syntactic constructs.

8 Conclusion

We have proposed *natural reductions* of parameterized concurrent programs, an approach for users to specify a subset of interleavings that may be easier to verify. Natural reductions can be specified with ease, by adding atomic blocks and global rendez-vous points to the thread template.

We have studied the problem of deciding whether a natural reduction given by a user is indeed *sound*, and can thus be soundly verified in place of the original program. We link the complexity of this problem to the corresponding coverability problem. As a consequence, the problem becomes intractable (coNP-hard) as soon as even very light-weight synchronization between threads is allowed (in particular, locks). This motivates us to take an abstract view of the program, i.e., to abstract away from the semantics of locks and other synchronization operators. We present the first complete decision procedure for checking soundness of natural reductions in this setting, and show that it runs in polynomial time. Hence, our approach overcomes the inherent incompleteness of previous work based on *mover reasoning*, while retaining polynomial complexity.

In the future, our decision procedure could replace resp. complement mover reasoning in reduction-based deductive verifiers such as Civl [19] or Anchor [14], broadening the range of specifiable reductions and providing a guarantee of completeness. As part of such an application, a relevant question for study is the extension of our polynomial-time algorithm to other settings. In particular, it is of interest to determine whether some form of synchronization (even more light-weight than locks), such as *structured concurrent programs*, *permissions*, or specific *locking disciplines* can be accommodated. Given that in such cases, it

is typically easy to determine which parts of the program may run in parallel, there is some hope that polynomial runtime can be retained.

At the same time, there also remain open questions in the cases where a precise, polynomial-time decision procedure does not exist. First, the matching upper bounds for our complexity results remains a question of theoretical interest, even if the practical impact for deductive proof systems may be limited. Second, we have shown that a precise solution requires a global view of the program, and the purely-local reasoning of Lipton movers is incomplete. This raises the question of whether there are (practically useful and feasible) intermediate points within the gap between fully-local, polynomial-time Lipton reasoning and a fully-global, super-polynomial algorithm. For the case of locks, we have sketched one such intermediate solution in Sect. 6.

Acknowledgments. Constantin Enea and Dominik Klumpp are partially supported by the French National Research Agency (projects "SCEPROOF" and "CENTEANES").

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying Concurrent Programs against Sequential Specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_17
2. Chajed, T., Kaashoek, M.F., Lampson, B.W., Zeldovich, N.: Verifying concurrent software using movers in CSPEC. In: OSDI, pp. 306–322. USENIX Association (2018)
3. Chou, C., Gafni, E.: Understanding and verifying distributed algorithms using stratified decomposition. In: PODC, pp. 44–65. ACM (1988). https://doi.org/10.1145/62546.62556
4. Clerbout, M., Latteux, M., Roos, Y.: Semi-commutations. In: The Book of Traces, pp. 487–552. World Scientific (1995). https://doi.org/10.1142/9789814261456_0012
5. Damian, A., Drăgoi, C., Militaru, A., Widder, J.: Communication-Closed Asynchronous Protocols. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 344–363. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_20
6. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL, pp. 2–15. ACM (2009). https://doi.org/10.1145/1480881.1480885
7. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. **2**(3), 155–173 (1982). https://doi.org/10.1016/0167-6423(83)90013-8
8. Enea, C., Farzan, A., Klumpp, D.: On the complexity of checking soundness of natural reductions (extended version). Tech. rep. (2026). https://doi.org/10.48550/arXiv.2605.13780

9. Esparza, J.: Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In: STACS, pp. 1–10. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2014). https://doi.org/10.4230/LIPICS.STACS.2014.1
10. Farzan, A., Klumpp, D., Podelski, A.: Sound sequentialization for concurrent program verification. In: PLDI, pp. 506–521. ACM (2022). https://doi.org/10.1145/3519939.3523727
11. Farzan, A., Klumpp, D., Podelski, A.: Commutativity simplifies proofs of parameterized programs. Proc. ACM Program. Lang. **8**(POPL), 2485–2513 (2024). https://doi.org/10.1145/3632925
12. Farzan, A., Madhusudan, P.: The Complexity of Predicting Atomicity Violations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 155–169. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_14
13. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc. ACM Program. Lang. **4**(POPL), 13:1–13:28 (2020). https://doi.org/10.1145/3371081
14. Flanagan, C., Freund, S.N.: The Anchor verifier for blocking and non-blocking concurrent software. Proc. ACM Program. Lang. **4**(OOPSLA), 156:1–156:29 (2020). https://doi.org/10.1145/3428224
15. Gangamreddypalli, N., Enea, C., Qadeer, S.: Reduction for structured concurrent programs. In: ESOP (1), pp. 252–282. LNCS, Springer, Cham (2026). https://doi.org/10.1007/978-3-032-22720-1_10
16. von Gleissenthall, K., Kici, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: synchronous verification of asynchronous distributed programs. Proc. ACM Program. Lang. **3**(POPL), 59:1–59:30 (2019). https://doi.org/10.1145/3290372
17. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60761-7
18. Hawblitzel, C., et al.: Ironfleet: proving practical distributed systems correct. In: SOSP, pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428
19. Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and Modular Refinement Reasoning for Concurrent Programs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 449–465. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_26
20. Kragl, B., Enea, C., Henzinger, T.A., Mutluergil, S.O., Qadeer, S.: Inductive sequentialization of asynchronous programs. In: PLDI, pp. 227–242. ACM (2020). https://doi.org/10.1145/3385412.3385980
21. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975). https://doi.org/10.1145/361227.361234
22. Lorch, J.R., et al.: Armada: low-effort verification of high-performance concurrent programs. In: PLDI, pp. 197–210. ACM (2020). https://doi.org/10.1145/3385412.3385971
23. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 255, pp. 278–324. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_30
24. Mutluergil, S.O., Tasiran, S.: A mechanized refinement proof of the Chase-Lev deque using a proof system. Computing **101**(1), 59–74 (2019). https://doi.org/10.1007/S00607-018-0635-4