Commutativity in Automated Verification

Azadeh Farzan
Department of Computer Science
University of Toronto
Toronto, Canada
azadeh@cs.toronto.edu

***Abstract*—Trace theory (formulated by Mazurkiewicz in 1987) is a framework for formalizing equivalence relations for concurrent program runs based on a commutativity relation over the set of atomic steps taken by individual program threads. It has been implicitly or explicitly used in a broad set of program analysis techniques, such as predictive testing for atomicity or data race violations, static and dynamic partial order reduction in model checking (particularly stateless model checking), and reasoning about distributed programs. In this paper, we introduce a different line of work that uses traces for the purpose of proof simplification for a broad set of automated verification goals. The long term thesis of this line of work has been that by taking advantage of commutativity, one can discover a substantially simpler verification task to replace the original one, and succeed at it despite the inevitability of the failure of the original one. The idea is to verify a different program in place of the original one and use commutativity as a way of soundly carrying the verification results over to the original one. We discuss hypersafety verification of sequential and concurrent programs, and safety and liveness verification of concurrent and distributed programs. We show how commutativity can be incorporated into a new verification algorithm which enumerates infinitely many possibilities for alternative programs to be verified instead of the original one. We conclude with an overview of some open research questions in this area.**

***Index Terms*—Concurrency, Distributed Programs, Automata, Commutativity**

I. Introduction

In 2017, we started looking into the problem of verification of *hypersafety* properties [1]. Intuitively speaking, these are properties whose violations are witnessed by one *or more* finite program runs: Classic safety properties are 1-safety properties and a k-safety property is one whose violation is witnessed by precisely k program runs. The most common way of encoding and proving hypersafety properties is through *self-composition*: A program P is composed with an additional $k-1$ copies of itself and the k-safety property of P is stated and proven as an equivalent 1-safety property of the resulting program $P^k = P|| \dots ||P$.

There is, however, an accepted wisdom amongst researchers [2] that the secret to success at this task lies in *not* proving P^k correct, but rather a *subset* of its behaviours. The reason for this is straightforward. In P^k, all k copies of P are distinct. Therefore, $P|| \dots ||P$ is *equivalent to* $P; \dots ; P$ for the purposes of the verification task; that is, the former satisfies the 1-safety property if and only if the latter does. A Floyd-Hoare style proof of $P; \dots ; P$ may require exponentially (on k) fewer assertions than that of P^k, since the latter has exponentially many more program locations. Therefore, in some sense of the word, the proof of $P; \dots ; P$ is a *simpler* program proof. But, *assertion quantity* is not the only thing that dictates the *complexity* of a Floyd-Hoare proof. The complexity of individual assertions, in terms of the assertion language, is somewhat more important if one has any hope of automation in this process. For example, reasoning about *linear arithmetic* is easier than reasoning about *nonlinear arithmetic*, and this is confirmed by the decidability status of the relevant decision procedures for them. This is precisely why, one does not simply abandon $P|| \dots ||P$ for $P; \dots ; P$ as a straightforward way of simplifying proofs; a proof for the latter requires a subset of the assertions of a proof for the former, but one has no control over which subset. Instead, one can opt for another program P', which is equivalent to P^k in the same sense, but its proof requires the simpler assertions from the proof of P^k.

Observe that since all the k copies of P in P^k are distinct, any two behaviours of P^k that are interleavings of the same local (per copy) behaviours are equivalent. P' is a *sound* choice if it includes one or more members from each such equivalence class. In the literature of hyperproperty verification and more generally relational program proofs, the choice of P' is called an *alignment* (of P^k).

In this paper, we call it a *reduction* of P^k, since our ultimate goal is to step out of this specific context and into a broader one that uses the same general reasoning principle. The equivalence classes induced by a commutativity relation are defined as *Mazurkiewicz Traces* [3]. For any program A, we call a program B to be a *reduction* of A if and only if B includes at least one representative from each (trace) equivalence class of behaviours in A. For example, $P; \dots ; P$ is a reduction of P^k.

In the case of hypersafety, we just happen to have full commutativity, since all copies are distinct. For a concurrent/distributed program, we typically have some (but not full) commutativity, but we can still follow the same principle that verifying a *reduction* of the program can be a substantially easier task than verifying 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

The author has been funded by an NSERC Discovery Grant.

979-8-3503-3587-3/23/$31.00 ©2023 IEEE

(or almost synchronous) programs in place of asynchronous programs with the rationale that the synchronous program admits a simpler proof [4], [5]. In the context of verification of concurrent programs, the same observation is exploited through the inference of large atomic blocks that would permit the prover to do much of the reasoning in a thread-local manner, and only reason about concurrency over a few interference points [6], [7].

The common thread in all three contexts of reasoning is that choosing the *right* reduction is the key to success in the verification task. Yet, in all cases, the right reduction is either fixed in advance, by choosing a specific domain of programs that share the choice of the right reduction, or the user is involved in specifying it somehow; for example through explicit *yield points in the code* [6], [7]. In some cases, this methodology may be reasonable. In general, two complications may arise: (1) It may not be an easy task for a user to envision an arbitrary reduction with a simple proof, and (2) even if the user has a high level understanding of what this reduction should look like, it is not always straightforward to communicate it to a tool; for example, *yield points* are easy to use for the users, but they cannot encode every reduction.

Since a verification algorithm is fundamentally a search for proof, we wondered if it can be turned into *a search for both a reduction and a proof of its correctness*. This paper briefly surveys some of the ideas that resulted from an attempt to actualize this goal.

II. A Suitable Algorithmic Verification Paradigm

The most common approach to reasoning about program correctness, among *programmers*, is based on operational reasoning, which consists of an informal analysis in terms of the executions of the given program. The classic approach to program correctness, among the *experts in this field*, is based on axiomatic reasoning. Such a proof will proceed in a syntax-directed manner by recursion on the structure of the program.

For concurrent and distributed programs, which consist of components executing simultaneously, Hoare's axiomatic approach was extended by Owicki/Gries [8] and Jones [9] to one that aims to construct a proof of correctness for the whole program out of proofs for individual components (threads or processes) and a modicum of glue that connects them together. This reasoning is based on the principle of non-interference: other threads/processes do not interfere with the functionality of the current one, and therefore do not have much impact on its proof of correctness. Not all concurrent programs are designed with this principle in mind. For cooperative concurrency, in which processes work together to achieve some common goal, the notion of non-interference is not natural. Axiomatic proof systems have clever workarounds (e.g., ghost state [10]) for these cases, but cleverness is an obstacle to automation: humans are clever, but automated verifiers are not. In contrast, informal operational arguments accommodate arbitrary global facts into proofs more naturally.

In a series of papers [11]–[15], we illustrated how a high level operational argument can be made rigorous like a classical axiomatic proof. The thesis of our work is that operational arguments can be formalized using axiomatic proofs of a program's behaviours, which are sequences of instructions that are executed in order (without conditional branching, looping, etc). The basic object of interest in this system remains a Hoare triple $\{P\}\ \rho\ \{Q\}$, which consists of two assertions P and Q (called the pre-condition and post-condition of the triple, respectively), and a program behaviour ρ. Hoare triples form the foundation of most axiomatic proof systems. In an operational argument, we think of a program as a set of its behaviours, corresponding to the possible paths of execution the program may take. We view an operational proof for a program as mechanism that, for any program behaviour ρ, can derive validity of the Hoare triple $\{\text{Pre}\}\ \rho\ \{\text{Post}\}$. The key is an implicit finite representation of a proof for a potentially infinite set of such behaviours. This raises two questions:

- What is the mechanism for deriving valid Hoare triples for program behaviours and representing the resulting argument as a coherent formal proof?
- How can we be assured that a correctness proof can be derived for every program behaviour?

In the context of algorithmic verification, this operational view can be implemented by a refinement loop illustrated in Figure 1. Roughly speaking, boxes (a) and (b) correspond to the first question, and boxes (c) and (d) correspond to the second question.

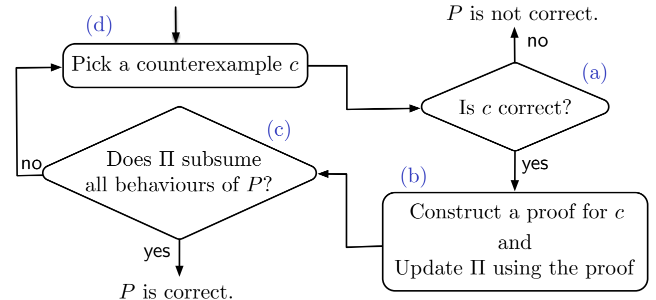

Fig. 1. A Refinement loop For operational-style reasoning about correctness of a program P.

The focus of this paper is not on this type of operational reasoning. Therefore, we refer the interested reader to [16], [17] for a few different ways in which the pair of questions can be consistently and effectively answered for different program models. This style of operational reasoning, however, has been essential in the success of incorporating commutativity into the verification algorithm. Think of it this way: if we do not know what reduction (which is a program in its own right) we will end up proving correct a priori, we do not have access to the syntax of this reduction (program). It is hard to imagine how a proof in the syntax-based axiomatic style might be gradually constructed without access to this syntax.

It suffices to have a high level understanding of how the refinement loop in Figure 1 works to read the rest of this paper. A program is always represented using an appropriate notion

of automaton, which simply captures its (syntactic) behaviours. The property of interest is already encoded in the program. For example, in place of having pre/post-conditions, we can let the program assume the precondition at the beginning, and state the violation of the postcondition at the end. Correctness of the original program would then become *infeasibility* of all (syntactic) behaviours of the new program.

The proof conjecture Π itself is represented by a compatible automaton, and always accepts a set of behaviours that are correct by construction. The check in box (c) becomes a straightforward language subsumption test. As long as it is decidable for the particular class of automata used, one has an algorithm for box (c). A counterexample, that is a witness to the violation of the subsumption check, will then be a program behaviour that is not covered by the proof. It can be viewed as a *simple* program; for example, in the context of safety, it is a finite run which can be viewed as a sequential program without any branching or looping structure. One can use one of the many existing verification technique (e.g., Craig interpolation) to verify it, and if it is correct produce a Hoare/Floyd style proof for it. Then, this proof can be generalized and combined (not necessarily in that order) with an existing proof conjecture Π. Therefore, Π may start from being the empty language in this refinement loop, and then iteratively grow larger based on counterexamples until the program is subsumed.

III. Dual Search for Reductions and Their Proofs

We start with a simple notion of commutativity: a and b (as atomic program steps) *commute* if and only if $a; b$ and $b; a$ have the same semantics (i.e. input-output relation). This leads to a simple observation that if a program run $\rho ab\rho'$ satisfies some specification (e.g. pre/post-condition) and a and b commute, then $\rho ba\rho'$ also satisfies the same specification. It is then convenient to call these and only these program runs *equivalent* under a commutativity relation I that includes the pair (a, b). The equivalence relation $\equiv_I$ induced by I is the reflexive and transitive closure of the relation $\sim_I$ defined as:

$$\sigma \sim_I \sigma' \iff \sigma = \rho ab\rho' \wedge \sigma' = \rho ba\rho' \wedge (a, b) \in I.$$

In other words, two program behaviours are equivalent if one can be acquired from the other through an arbitrary number of swaps of adjacent commutative actions. The equivalence classes defined by $\equiv_I$ are Mazurkiewicz traces [3].

In the refinement loop of Figure 1, this suggests a straightforward way of exploiting commutativity: for every behaviour σ in Π, include all other behaviours $\sigma' \equiv_I \sigma$ also in Π. This amounts to computing the commutativity closure of Π up to I and proposes the following proof rule for verification:

$$\frac{\Pi \models \varphi \quad P \subseteq [\Pi]^I}{P \models \varphi} \qquad \text{(CLOSURE)}$$

In a sense, we get more milage out of our efforts to construct proofs for individual counterexamples by using the commutativity closure. This proof rule is practically useless, however, since checking the premise $P \subset [\Pi]^I$ is undecidable [18] even for regular languages which are the simplest class of languages that can be used in our operational reasoning scheme. Therefore, we need a compromise. Consider the following rule:

$$\frac{\Pi \models \varphi \quad \exists \widehat{P} \in \mathit{Red}(P).\ \widehat{P} \subseteq \Pi}{P \models \varphi} \qquad \text{(REDUCTION)}$$

where $\mathit{Red}(P)$ is a set of programs that satisfies the following

$$\widehat{P} \in \mathit{Red}(P) \iff \forall \rho \in P\ \exists \rho' \in \widehat{P} \quad \rho \equiv_I \rho'$$

Note that if we let $\mathit{Red}(P)$ be the largest set of programs that are all subsets of behaviours of P and satisfy the above condition, then Rules CLOSURE and REDUCTION are equivalent, since for this $\mathit{Red}(P)$, we have:

$$(\exists \widehat{P} \in \mathit{Red}(P).\ \widehat{P} \subseteq \Pi) \iff P \subseteq [\Pi]^I$$

and therefore this new premise is as undecidable as the old one. However, it presents us with a new opportunity: settle for a subset of the reductions in $\mathit{Red}(P)$ so that the premise becomes algorithmically checkable.

In [11], we introduce one such expressive set of reductions inspired by *lexicographical* normal forms of Mazurkiewicz traces. The *lexicographical normal form* (LEX) of an equivalence class is simply lexicographically least member of the class, based on an order assumed on the alphabet of actions. Let Σ be an alphabet of program atomic actions. We define a new context-sensitive alphabet order $\prec$: $\Sigma^* \to \mathit{lin}(\Sigma)$ where $\mathit{lin}(\Sigma)$ is the set of all possible total orders on Σ. Intuitively, $\prec$ defines an ordering of actions sensitive to the context in which the letters appear in a program run.

Fix a choice of $\prec$ and define a reduction $\widehat{P}_\prec$ parametric on this choice as one satisfying the following condition for all $\rho \in P$:

$$\neg(\rho = \sigma_1 b \sigma_2 a \sigma_3 \wedge a \prec_{\sigma_1} b \wedge (a, b\sigma_2) \in I) \implies \rho \in \widehat{P}_\prec$$

The left hand side of the implication determines that the run ρ is the lexicographically least member of its equivalence class up to $\prec$. Therefore, it does belong to the reduction $\widehat{P}$, which only collects these normal forms. Let Red_{lex} be the set of all such reductions for all possible choices of $\prec$. Note that despite there being only finitely many total orders on Σ, there are infinitely many choices for $\prec$, and therefore, infinitely many reductions can be enumerated as long as the program has infinitely many runs. Moreover, each such reduction is *optimal* in the sense that it includes precisely one member of each equivalence class of program behaviours. Therefore, the proof does not need to cover any redundant program behaviours.

The technical breakthrough in [11] is that for a regular program language P, the set of reductions $\mathit{Red}_{lex}(P)$ can be effectively represented using a Looping Tree Automaton (LTA). Therefore, in the diagram in Figure 1, all instances of P can be replaced with $\mathit{Red}_{lex}(P)$. On the other hand, the set of all subsets of a regular language Π can also be represented by an LTA, which can then replace all instances of Π. Then, the premise check $\exists p \in \mathit{Red}_{lex}(P) \quad p \subseteq \Pi$

reduces to checking if the intersection of the two LTAs is nonempty. This is decidable in linear time. Remarkably, even though individual reductions represented by the LTA do not have to be all regular and the same is true for subsets of Π, whenever the intersection is nonempty, it always includes a regular language in it. Therefore, the algorithm can produce the regular reduction as a checkable certificate.

Moreover, we prove in [11] that whenever the intersection is empty, that is while the refinement loop is not yet done, *a finite set of counterexamples* witness the *insufficiency* of the proof. That is, there are finitely many counterexample program behaviour that, if proven correct in the next round and added to proof Π, guarantee us *progress* across all *infinitely many* reductions. Therefore, the instantiation of refinement loop in Figure 1 based on Rule REDUCTION with $Red_{lex}(P)$ as the set of reductions enjoys the same progress properties as the one that simply works with a single regular program and its proof. This is a highly nontrivial observation and the key to a proper dual algorithmic search for a reduction and a proof: one does not get stuck by committing to a *bad* reduction when it comes to the expansion of the proof.

In [11], we demonstrate that an implementation of this idea is successful in discovering different reductions that are required to verify different benchmarks from different contexts. This includes discovering the right alignments for hypersafety verification of sequential and concurrent programs, correct sequentialization for the verification of concurrent programs, and the right type of synchronization for the verification of distributed programs. The key focus of the empirical study of [11] is on problems where the invariants are beyond the reach of standard automated verification techniques, mainly because the language of assertions for a full program proof involves assertions that are nonlinear, use quantification over arrays, or reason about buffers with unlimited bounds, or are simply hard to *guess* by automated techniques (e.g. interpolation). In each case, there exist a reduction with a proof using *simple* assertions and the tool, WEAVER, is able to discover both.

IV. SEMI- AND CONTEXTUAL COMMUTATIVITY

The reader who is partially familiar with the literature on commutativity in concurrent program verification most likely knows about Lipton's reductions [19], where individual program actions can be *left*, *right*, *both*, or *non* movers. The technical ideas described in Section III can be extended, with a bit of effort, to work with a non-symmetric notion of commutativity, namely *semi-commutativity*, as discussed in [12]. This is important for taking advantage of the existing semi-commutativity between program actions where full commutativity does not exist. For example, a `send` and a `receive` operation on the same channel in a distributed program, where a `receive` operation can be commuted to the right of any `send` operation but not to its left. We discuss in [12] how Lipton's reductions and such *semi-reductions* are incomparable notions.

In [12], another notion of commutativity, namely *contextual commutativity* is also investigated. In message-passing programs, a `receive` can be commuted to the left of a `send` operation that is not its matching `send`, which is determined by the context. The most significant contribution of [12] is a proposed class of reductions, called *contextual reductions*, which take advantage of such *contextual commutativity*. Two statements may commute in one context and not in another. Standard commutativity would declare them always non-commutative. Contextual commutativity lets the two statements commute wherever they do soundly.

Inspired by a language-theoretic notion of context from generalized Mazurkiewicz traces [20], we define contextual reductions that are recognized by LTAs. The elegance of this definition is in that it does not commit to a particular contextual commutativity relation in advance. Similarly to the definition of $\prec$ in Section III, we can define a contextual commutativity relation $I : \Sigma^* \to \Sigma \times \Sigma$ which can encode an arbitrary commutativity relation at any given context. Note that not all such relations are sound, that is, not all pairs of actions may soundly commute in the corresponding contexts as specified. A key technical development in [12] is that we can defer the decision of the soundness of a particular choice to the proof checking step. That is, a proof candidate decides if a choice of contextual commutativity relation is provably sound.

An LTA then models (infinitely many) reductions of a program based on infinitely many choices of contextual commutativity relations and infinitely many choices of contextual orders $\prec$, in the same style as Red_{lex} from Section III. Another LTA captures all subsets of Π. Proof checking passes if we find a reduction that is subsumed by Π and the contextual commutativity choices that are made in the construction of it are proven sound by Π. When we use counterexamples to grow Π, we trick the refinement loop to also infer assertions that substantiate the soundness of a larger and larger contextual commutativity relation through refinement.

In some sense, there are three searches that happen simultaneously: a search for a provably sound contextual commutativity relation, a search for a reduction induced by some provably sound commutativity relation, and a search for a proof of the correctness of the reduction and the soundness of the contextual commutativity relation used in defining it.

In the context of hypersafety verification, since one always has access to a full (symmetric) commutativity relation, neither semi-commutativity nor contextual commutativity is of any use. But, in the context of verification of concurrent and distributed programs, as is demonstrated in [12], a tool equipped with the additional power, SIEVER, can manage to prove many more programs correct than does WEAVER which can only do non-contextual symmetric commutativity reasoning.

V. ABSTRACT COMMUTATIVITY

Contextual commutativity allows us to declare more program behaviour equivalent. There is another simple observation that can produce more commutativity in program reasoning. Two program actions may not commute if one considers their concrete semantics, but in the context of a

specific program and a specific property of that program, the concrete semantics may be more detailed than necessary. If one considers an abstraction of the program actions which still maintains the property of interest for the program, the two actions may commute under this *abstract* semantics. In [14], we explore how this can be formalized and effectively used in a verification algorithm.

The idea of abstract commutativity has been used in the literature [6], [7]. Our shared view is that abstraction can help increase commutativity. However, the increased commutativity there is chiefly used for the construction of larger atomic blocks to exploit local reasoning. Our thesis is different. We believe that abstract commutativity can play a significant role in proving properties of concurrent programs that concern only a comparatively small slice of the program. Examples of these are properties local to a thread, or safety of memory accesses, or lightweight properties like race detection. The idea is that one can infer near total commutativity of actions outside the program slice and as a result get substantially smaller representation for a sound reduction of the program based on the abstract commutativity relation.

A contribution of [14] is a framework in which we define general soundness conditions for abstract commutativity relations to ensure the correctness of the verification is preserved. We base definitions of abstract commutativity (as well as the notion of its soundness) on a given proof candidate (Π in the diagram in Figure 1). The proof relies on the reduction defined by the abstract commutativity relation, and in turn the proof guarantees soundness of the abstract commutativity relation. This circular dependency is vaguely reminiscent of rely-guarantee reasoning. We define several sound and generic abstract commutativity relations and show empirically that even coarse abstractions have significant benefit for the verification algorithm, specially for lightweight properties of concurrent programs.

Instantiating this framework with several abstract commutativity relations leads to a very important insight: Commutativity does not increase monotonically with an increase in the level of abstraction. For example, two `x++` statements commute *under concrete semantics*, but switching to an abstract semantics that loosely says that `x` can be any value after the statement, they would no longer commute. This lack of monotonicity brought us to the following question: Can we leverage the power of multiple commutativity relations, that are not consistent with each other, to construct a simple proof in an automated verification algorithm?

VI. Stratification of Commutativity Relations

Unlike the interactive setting (for example [6]), a verification algorithm cannot rely on the user to intervene and decide where to apply which abstraction to obtain an ideal abstract commutativity relation. Straightforward approaches to combining different commutativity relations, such as taking their union, turn out to be either unsound or unsuitable for verification algorithms.

In [14], we propose a general way of combining two or more commutativity relations *in strata*. To establish that one program behaviour is equivalent to another one, we perform an arbitrary number of commutations according to one relation, and then switch to the next one, and then the next one. More precisely if we have two commutativity relations I and I', in Rule Closure, the inclusion check $P \subseteq [\Pi]^I$ can be soundly replaced by

$$P \subseteq \left[[\Pi]^I\right]^{I'}$$

under the restriction that I is based on semantics that is strictly more abstract than that of I'. To have an algorithmic solution, however, we need the corresponding version of this for Rule Reduction. Switching gears from closures to reductions, we want the following check instead:

$$\forall \sigma \in P \; \exists \sigma' \in \Pi, \sigma'' \in \Sigma^* \quad \sigma \equiv_{I'} \sigma'' \equiv_I \sigma'$$

This is generally equivalent to, and as undecidable as, the variation of test with stratified closures. In the spirit of reductions inspired by lexicographical normal forms, we can switch to the following decidable strengthening of the check:

$$\forall \sigma \in P \; \exists \sigma' \in \Pi, \sigma'' \in \Sigma^* \quad \sigma \equiv_{I'} \sigma'' \equiv_I \sigma' \wedge \; \sigma' \prec \sigma'' \prec \sigma$$

This gives rise to the concept of *stratified reductions*. The above test is algorithmically implemented in the refinement loop of Figure 1 as the following inclusion check:

$$red^I_\prec\left(red^{I'}_\prec(P) \cup \Pi\right) \subseteq \Pi \tag{SR}$$

which is provably equisatisfiable [14]. This can be extended from two to any number of commutativity relations.

The algorithmic implementation of the SR check is nontrivial. There is an inherent nondeterminism engrained in the idea of stratified commutativity which is related to the decision of *when* to switch from swapping according to one relation to the next. We present an effective algorithm which resolves this nondeterminism in a provably optimal way, and therefore, precisely implements the SR check.

VII. Beyond Safety

So far, our focus has been on safety properties of concurrent/distributed programs. In the automata-theoretic framework, it is fairly straightforward [17] to reduce the check for an arbitrary liveness property to a termination check for a transformed program. Therefore, we turn to the problem of checking termination of concurrent/distributed programs as the foundation of reasoning about liveness. The question arises naturally whether the ideas developed for verification of safety properties could work, with little adjustment, for this verification task as well. The answer turns out to be NO.

Recall the refinement loop in Figure 1. Without considerations for commutativity, the same refinement loop can be used for checking termination of concurrent programs. The program and the proof are represented using (Büchi) automata, and module (c) is implemented as inclusion checks between the languages of these automata. Counterexamples are program *lassos* which correspond to *ultimately periodic* words.

For module (b), any known technique to reason about the termination of a lasso, which is a simple sequential program, can be used.

The same notion of commutativity equivalence, which we discussed for finite program behaviours, also exists for infinite program behaviours, whether they are ultimately periodic or not. In the same manner, it turns out that accounting for commutativity by computing the closure of the proof Π (in the stye of Rule CLOSURE) immediately leads to an undecidable proof rule [15].

However, unlike all previous sections in this paper, a notion of program reduction based on lexicographical normal forms does not provide us with a sound and decidable proof rule. This is due to the simple fact that equivalence classes of ω-words may include words that are themselves not ω-words, and consequently, the lexicographically least member of an equivalence class does not necessarily have to be a word in Σ^ω.

Let λ_1 and λ_2 stand for the loop bodies of the two loops in two different threads that fully commute. Let the alphabet ordering rank every letter of λ_1 before every letter of λ_2. The interleaved run $\lambda_1^\omega \lambda_2^\omega$ becomes the lexicographically least member of the equivalence class of $(\lambda_1\lambda_2)^\omega$. Since λ_1 and λ_2 commute, we can make *infinitely many* swaps to transform $(\lambda_1\lambda_2)^\omega$ to $\lambda_1^\omega \lambda_2^\omega$. Note that the former is an ω-word, but the latter is not and is called a *transfinite* word.

First, It appears that we are indefinitely postponing λ_2 in favour of λ_1. Second, a word with a length strictly larger than ω does not have an appropriate representation in language theory because it does not belong to Σ^ω. Both are viewed as unholy things in the context of software model checking.

Yet, the observation that $(\lambda_1\lambda_2)^\omega \equiv \lambda_1^\omega \lambda_2^\omega$ is the key to a powerful proof rule for termination of concurrent programs: If λ_1^ω is terminating and λ_1 commutes against λ_2, then we can conclude that $(\lambda_1\lambda_2)^\omega$ is terminating. Note that the converse is not true; termination of $\lambda_1^\omega \lambda_2^\omega$ does not necessarily imply the termination of λ_2^ω. Hence, for the termination of the entire program (and not just the run $(\lambda_1\lambda_2)^\omega$), one needs to argue about the termination of both λ_1^ω and λ_2^ω. This is perfectly aligned with a high level argument a human would do: If two loops fully commute, it should suffice to prove each terminating to have a proof for the termination of their concurrent composition. In [15], we formally state and prove this proof rule, called the *omega-prefix* proof rule, and show how it can be incorporated into an algorithmic verification framework. In contrast to all previous cases, the natural way to incorporate this rule is as a generalization of the proof Π rather than a reduction of the program P.

This proof rule is powerful in reasoning about independent loops. But, when some actions in λ_1 and λ_2 commute and some do not, it does not get us far in simplifying the termination proof of the parallel composition of the two loops. To also take advantage of such scenarios, we propose a way of soundly and efficiently finitizing the ω-regular languages of program and the proof lassos in [15] so that reductions for finite-word languages can be used instead. These reductions can be seamlessly combined with any proof generalization scheme used for Π, in particular, one that implements the *omega-prefix* proof rule. As a result, a coherent algorithmic solution emerges which performs substantially better than the baseline in checking termination of concurrent programs.

VIII. Future Directions

We believe that we have seen strong evidence that the use of commutativity can simplify proofs of concurrent and distributed programs, for safety, liveness, and hypersafety properties. There are however many open questions left in this domain.

First, we have a methodology for adding *contexts* to *concrete commutativity* relations. We do not yet know how to do the same thing for a generic *abstract commutativity* relation. To add to this mystery, there seems to be a loose connection between *contextual* and *abstract* commutativity. In our work so far, whenever we talk about *context*, we refer to a *left* context only. Whereas, abstractions, in some sense, provide the power to account for a full (both left and right) context to consider the relative order of two program actions. It will be very interesting to explore and formalize this relation further.

Second, our focus so far has been on programs with a fixed number of threads (or processes). All constructions rely on the *finite* alphabet of actions. For parameterized programs or distributed protocols, which contain unboundedly many threads, one needs an infinite alphabet of actions to model the program behaviour faithfully. It will be interesting to investigate if and how the verification of parametrized programs can be simplified using the concepts of commutativity mentioned in this article. In particular, it will be interesting to formulate what reductions of parameterized programs may look like and how they can be finitely represented and analyzed.

Lastly, and perhaps most importantly, the theory behind every work referenced in this article relies on classic results proven about Mazurkiewicz traces. They characterize a clean and elegant notion of commutativity that simplifies and streamlines reasoning about the equivalence classes. The same is true about much of the literature on the usage of commutativity in other areas, such as dynamic program analysis for predictive testing for atomicity [21], [22] and race freedom [23], [24], and dynamic partial order reduction [25] in stateless model checking [26], [27]. However, to anyone with some experience in concurrent and distributed programming, it is evident that commutativity at the level of atomic actions is somewhat constrained. For example, an atomic increment followed by an atomic decrement amounts to a `skip` action that commutes against any environment action, while the individual increment and decrement may not. It would be very interesting to explore *coarser* notions of equivalence for concurrent program runs that would share the key properties of Mazurkiewicz traces which have proven to be broadly useful in many areas of computer science like programming languages, software engineering, and distributed computing since 1987.

REFERENCES

[1] R. Pucella and F. B. Schneider, "Independence from obfuscation: A semantic framework for diversity," *J. Comput. Secur.*, vol. 18, no. 5, pp. 701–749, 2010.

[2] G. Barthe, P. R. D'argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[3] A. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer-Verlag New York, Inc., 1987, pp. 279–324.

[4] K. von Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala, "Pretend synchrony: synchronous verification of asynchronous distributed programs," *PACMPL*, vol. 3, no. POPL, pp. 59:1–59:30, 2019.

[5] B. Genest, D. Kuske, and A. Muscholl, "On communicating automata with bounded channels," *Fundam. Inform.*, vol. 80, no. 1-3, pp. 147–167, 2007.

[6] T. Elmas, S. Qadeer, and S. Tasiran, "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*, Z. Shao and B. C. Pierce, Eds. ACM, 2009, pp. 2–15.

[7] B. Kragl and S. Qadeer, "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*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 79–102.

[8] S. S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976. [Online]. Available: https://doi.org/10.1145/360051.360224

[9] C. B. Jones, "The early search for tractable ways of reasoning about programs," *IEEE Ann. Hist. Comput.*, vol. 25, no. 2, p. 26?49, apr 2003. [Online]. Available: https://doi.org/10.1109/MAHC.2003.1203057

[10] L. Lamport, "Control predicates are better than dummy variables for reasoning about program control," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, p. 267?281, apr 1988. [Online]. Available: https://doi.org/10.1145/42190.42348

[11] A. Farzan and A. Vandikas, "Automated hypersafety verification," in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer, 2019, pp. 200–218.

[12] ——, "Reductions for safety proofs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–28, 2020.

[13] A. Farzan, D. Klumpp, and A. Podelski, "Sound sequentialization for concurrent program verification," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 506–521.

[14] ——, "Stratified commutativity in verification algorithms for concurrent programs," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 1426–1453, 2023.

[15] A. Farzan and D. Lette, "Commutativity for concurrent program termination proofs," in *Computer Aided Verification: 31st International Conference, CAV 2023*. Springer, 2023.

[16] A. Farzan, Z. Kincaid, and A. Podelski, "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*, 2015, pp. 407–420.

[17] ——, "Proving liveness of parameterized programs," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 185–196. [Online]. Available: https://doi.org/10.1145/2933575.2935310

[18] V. Diekert and G. Rozenberg, *The book of traces*. World scientific, 1995.

[19] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975.

[20] V. Sassone, M. Nielsen, and G. Winskel, "Deterministic behavioural models for concurrency," in *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, 1993, pp. 682–692.

[21] A. Farzan and P. Madhusudan, "Monitoring atomicity in concurrent programs," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 52–65.

[22] A. Farzan, P. Madhusudan, and F. Sorrentino, "Meta-analysis for atomicity violations under nested locking," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 248–262.

[23] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490

[24] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 387–400. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103702

[25] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 110–121. [Online]. Available: https://doi.org/10.1145/1040305.1040315

[26] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 174–186. [Online]. Available: https://doi.org/10.1145/263699.263717

[27] M. Kokologiannakis, I. Marmanis, V. Gladstein, and V. Vafeiadis, "Truly stateless, optimal dynamic partial order reduction," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498711