Coarser Equivalences for Causal Concurrency

AZADEH FARZAN, University of Toronto, Canada

UMANG MATHUR, National University of Singapore, Singapore

Trace theory (formulated by Mazurkiewicz in 1987) is a principled framework for defining equivalence relations for concurrent program runs based on a commutativity relation over the set of atomic steps taken by individual program threads. Its simplicity, elegance, and algorithmic efficiency makes it useful in many different contexts including program verification and testing. It is well-understood that the larger the equivalence classes are, the more benefits they would bring to the algorithms and applications that use them. In this paper, we study relaxations of trace equivalence with the goal of maintaining its algorithmic advantages.

We first prove that the largest appropriate relaxation of trace equivalence, an equivalence relation that preserves the order of steps taken by each thread *and* what write operation each read operation observes, does not yield efficient algorithms. Specifically, we prove a *linear space lower bound* for the problem of checking, in a streaming setting, if two arbitrary steps of a concurrent program run are *causally concurrent* (i.e. they can be reordered in an equivalent run) or *causally ordered* (i.e. they always appear in the same order in all equivalent runs). The same problem can be decided in *constant space* for trace equivalence. Next, we propose a new commutativity-based notion of equivalence called *grain equivalence* that is strictly more relaxed than trace equivalence, and yet yields a constant space algorithm for the same problem. This notion of equivalence uses commutativity of *grains*, which are sequences of atomic steps, in addition to the standard commutativity from trace theory. We study the two distinct cases when the grains are contiguous subwords of the input program run and when they are not, formulate the precise definition of causal concurrency in each case, and show that they can be decided in *constant space*, despite being strict relaxations of the notion of causal concurrency based on trace equivalence.

CCS Concepts: • **Theory of computation → Concurrency**; **Formal languages and automata theory**; *Program analysis*; • **Software and its engineering** → *Software verification and validation.*

Additional Key Words and Phrases: concurrency, equivalence, reads-from, reduction, predictive analysis

ACM Reference Format:

Azadeh Farzan and Umang Mathur. 2024. Coarser Equivalences for Causal Concurrency. *Proc. ACM Program. Lang.* 8, POPL, Article 31 (January 2024), 31 pages. https://doi.org/10.1145/3632873

1 INTRODUCTION

In the last 50 years, several models have been introduced for concurrency and parallelism, of which Petri nets [Hack 1976], Hoare's CSP [Hoare 1978], Milner's CCS [Milner 1980], and event structures [Winskel 1987] are prominent examples. Trace theory [Diekert and Rozenberg 1995] is a paradigm

Authors' addresses: Azadeh Farzan, University of Toronto, Toronto, Canada, azadeh@cs.toronto.edu; Umang Mathur, National University of Singapore, Singapore, Singapore, umathur@comp.nus.edu.sg.

ACM 2475-1421/2024/1-ART31
https://doi.org/10.1145/3632873

in the same spirit which enriches words (or sequences) by a very restricted yet widely applicable mechanism to model parallelism: some pairs of *events* (atomic steps performed by individual threads) are determined statically to be *independent* (or commutative), and any two sequences that can be transformed to each other through swaps of consecutive independent events are identified as *trace equivalent*. In other words, it constructs a notion of equivalence based on *commutativity* of individual events. The simplicity of trace theory, first formulated by Mazurkiewicz in 1987 [Mazurkiewicz 1987], has made it highly popular in a number of areas in computer science, including programming languages, distributed computing, computer systems, and software engineering. The brilliance of trace theory lies in its simplicity, both conceptually and in yielding simple and efficient algorithms for several core problems in the context of concurrent and distributed programs. It has been widely used in both dynamic program analysis and in construction of program proofs. In dynamic program analysis, it has applications in predictive testing, for instance in data race prediction [Elmas et al. 2007; Flanagan and Freund 2009; Huang et al. 2014; Itzkovitz et al. 1999; Kini et al. 2017; Pavlogiannis 2019; Smaragdakis et al. 2012] and in prediction of atomicity violations [Farzan and Madhusudan 2006; Farzan et al. 2009; Mathur and Viswanathan 2020; Sorrentino et al. 2010] among others. In verification, it is used for the simplification of verification of concurrent and distributed programs [Desai et al. 2014; Drăgoi et al. 2016; Farzan 2023; Farzan et al. 2022; Farzan and Vandikas 2019, 2020; Genest et al. 2007].

The philosophy behind most applications of trace theory is that a single representative replaces an entire set of equivalent runs. Therefore, these applications would clearly benefit if larger sets of concurrent runs could soundly be considered equivalent. This motivates the key question in this paper: Can we retain all the benefits of classical trace theory while soundly enlarging the equivalence classes to improve the algorithms that use them? It is not difficult to come up with sound equivalence relations with larger classes. Abdulla et al. [2019] describe the sound equivalence relation *reads-from equivalence* which has the largest classes that remain sound and is defined in a way to preserve two essential properties of concurrent program runs: (1) the total order of *events* in each thread must remain the same in every equivalent run, and (2) each read event must observe the value written from the same write event in every equivalent run. The former is commonly known as the preservation of *program order*, and the latter as the preservation of the *reads-from* relation. These conditions guarantee that any equivalent run is also a valid run of the same concurrent program, and all the observed values (by read events) remain the same, which implies that the outcome of the program run must remain the same. That is, both control flow and data flow are preserved.

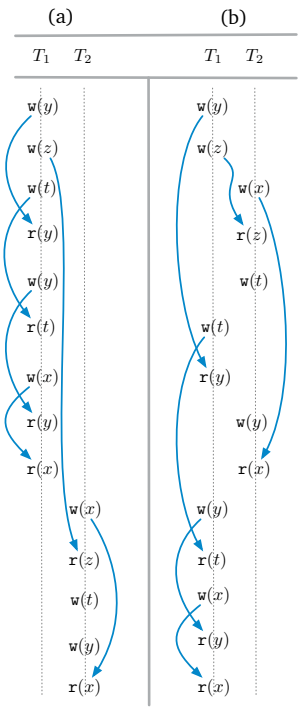


Fig. 1. Read-From Equivalence

Consider the concurrent program run illustrated in Fig. 1(a), and let us focus on the order of the read event $\mathtt{r}(x)$ from thread T_1 and the write event $\mathtt{w}(x)$ from thread T_2. In classical trace theory, these events are dependent (or non-commutatitive) and they must be appear in the same order in every member of the equivalence class of the run. This implies that the illustrated run belongs in an equivalence class of size 1. On the other hand, there exist other reads-from equivalent runs; one such run is illustrated in Fig. 1(b), in which the two aforementioned events have been reordered. The arrows connect each write event to the read event that reads its value, which remain unchanged between the two runs.

We give a formal argument for why (the more relaxed) reads-from equivalence is not as useful as trace equivalence from an algorithmic standpoint. One of the most fundamental algorithmic questions in this context is: Given two events e and e' in a run ρ, do they always appear in the same order in every member of the equivalence class of ρ or can they be reordered in an equivalent run? In the former case, we call e and e' *causally ordered*, and otherwise *causally concurrent*. For example, the events $\mathtt{r}(x)$ from thread T_1 and $\mathtt{w}(x)$ from thread T_2 are causally ordered under trace equivalence but causally concurrent under reads-from equivalence. On the other hand, $\mathtt{w}(z)$ event of thread T_1 and the $\mathtt{r}(z)$ event of thread T_2 are causally ordered under both equivalence relations.

For trace equivalence, it is possible to decide if two events are causally ordered or concurrent, using a *single pass constant space* algorithm; if the length of the run is assumed to be the size of the input, and other program measures such as the number of threads and the number of shared variables are considered to be constants. In Section 3, we prove that if *equivalence* is defined as broadly as reads-from equivalence, then this check can no longer be done in *constant space* by proving a *linear space* lower bound (Theorem 3.1). In particular, we prove this for two closely related variants of this problem: (1) the decision about the ordering of two specific events (as discussed in the example of Fig. 1), and (2) the decision about the ordering of any two occurrences of a specific (atomic) action (e.g. are any two $\mathtt{w}(x)$ actions unordered?). Both problems are closely related to predictive testing of violations of generic correctness properties for concurrent programs, such as data race freedom [Kini et al. 2017; Smaragdakis et al. 2012], deadlock freedom [Kalhauge and Palsberg 2018; Tunç et al. 2023] and atomicity [Farzan et al. 2009], and have applications in dynamic partial order reduction techniques [Abdulla et al. 2019; Flanagan and Godefroid 2005; Kokologiannakis et al. 2022] for model checking of concurrent programs. In all such contexts, having a monitor whose state space does not depend on the length of the input program run, which may include billions of events, is highly desirable. Thus far, the only existing instance of such a monitor has been based on trace equivalence.

We propose a new notion of equivalence for concurrent program runs, which in terms of expressivity lies in between trace and reads-from equivalences, and retains the highly desirable algorithmic simplicity of traces. The idea is based on enriching the classical commutativity relation of trace theory to additionally account for commutativity of certain *sequences of events*, called *grains*. A grain can be an arbitrarily long sequence of operations, which can belong to multiple threads. What motivates this definition is that in places where swapping a pair of individual events may not be possible, groups of operations (as grains) may still commute *soundly*, meaning without disturbing the program order or the reads-from relation. For two grains to be swappable, they must be adjacent and contiguous, at the time they are swapped.

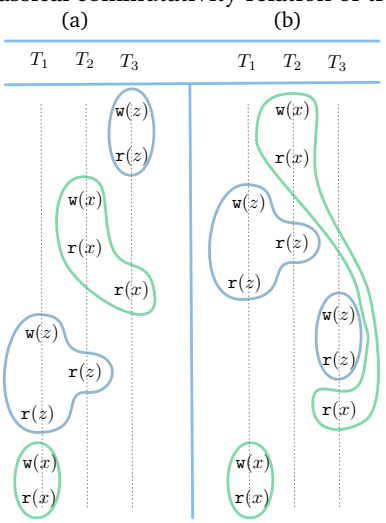


Fig. 2. Commuting Grains

As an example, consider the concurrent program run in Fig. 2(a). Four grains are marked. Observe that the grains of the same colour soundly commute. Grains of different colour always share a thread and therefore commuting them would break program order. First, assume that only the two blue grains exist. One can then use a sequence of swaps that are either standard swaps permitted by trace equivalence, or a swap of the two blue grains, and

transform the run in (a) to the one illustrated in Fig. 2(b), and hence reorder the two $\mathtt{w}(z)$ events. We call the run in (b) to be *grain-equivalent* to the one in (a). Observe that this is not possible under trace equivalence. A similar observation can be made if we consider only the green grains, and the goal of reordering the two $\mathtt{w}(x)$ operations. However, if all four grains are considered together, since the grains of different colour do not commute, nothing can be swapped in Fig. 2(a). Once something is decided to be part of a grain, it must always move together with the rest of the grain. This turns out to be a key to algorithmic simplicity.

In Section 4, we formally define the set of runs that are soundly equivalent to a program run based on a choice of grains that, as in Fig. 2(a), appear as contiguous subwords of the program run. We observe that different choices of grains can imply the concurrency of different pairs of events. We define *grain concurrency* of two events formally as the existence of a choice of grains and a sound commutativity relation (over the grains) such that the two events can be (soundly) reordered in a *grain-equivalent run* up to those choices.

The construction of a monitor based on trace equivalence vitally relies on the fact that the alphabet of concurrent programs is finite and consequently there are finitely many choices of commutativity relations over this alphabet. Note that, once any subword can become a new entity that participates in commutativity-based reasoning, the number of these subwords is no longer bounded. Neither is the set of possible commutativity relations over them. In Section 5, we prove by construction that *grain-equivalence* can be monitored in constant-space. This construction relies on a key insight that the monitor can maintain *summaries* for these grains that belong to a finite universe, and correctly check the concurrency status of two events.

Let us revisit the example runs in Fig. 2. The run in Fig. 2(b) is the result of commuting two blue grains in the run in Fig. 2(b). The first green grain, however, is no longer a contiguous subword and therefore cannot be seen as a potential grain in the solution we have outlined so far. In Section 6, we formally expand the definition of grains so that these so-called *scattered grains* can be considered as candidates as well. This requires a leap in the definition of the set of words that are soundly equivalent to the input run. In the case of contiguous grains, the set of equivalent runs maintains the characteristic of classic trace theory that one can transform the input run to any inferred equivalent run through a sequence of valid swaps. With *scattered grains*, we establish soundness by deferring to the more general definition of *reads-from equivalence*, and consequently forfeit the characteristic of transformation through swaps.

Surprisingly, however, this weaker definition still maintains the property that it can be monitored in constant-space. First, there is the additional complication that unlike contiguous subwords where at most one grain is *active* (open) at any given time, there may be an unbounded number of *active* scattered grains in a concurrent program run at any given time. Nevertheless, we prove that if an outcome can be decided based on an arbitrary choice of scattered grains, then it can also be decided based on a choice of scattered grains in which the number of active grains is bounded. With a bounded number of active grains (in contrast to just a single one), there is another complication where two active grains, which belong to the middle of a chain witnessing that e and e' are ordered, are only discovered to be ordered long after e and e' have been visited. In Section 7, we construct a monitor that resolves these problems and soundly checks the concurrency of a pair of events based on all possible choices of scattered grains, which we call *scattered grain concurrency*.

Even though the *scattered grain concurrency* monitor subsumes the *grain concurrency* monitor in expressivity, the paper presents these monitors separately, since the former admits a characterization

based on swaps and the latter does not. Moreover, this permits us to introduce the relevant ideas in tandem with the problems they help solve. In summary, the paper presents the following results:

- We prove that there is no constant-space algorithm that checks if two arbitrary events are causally ordered or concurrent under the reads-from equivalence relation by establishing a linear space lower bound, a quadratic time-space tradeoff bound, a conditional quadratic time lower bound as well as the non context-freeness of this problem. We complement these lower bounds by showing that the problem can nevertheless be solved in polynomial time as well as in deterministic linear space. (Section 3).
- We propose *grain equivalence* as the means of defining a set of runs that are soundly equivalent to a given program run and form a strictly larger set than the trace equivalence class of the run, and a strictly smaller set than the reads-from equivalence class of it. This notion of equivalence is constructed based on commutativity of certain pairs of words over the alphabet of concurrent program operations that appear as contiguous subwords of the input program run (Section 4).
- We introduce the notion of *grain concurrency* that attempts to find a witness for causal concurrency of a pair of events based on all possible choices of grains. We further weaken the definition of *grain concurrency* by permitting *scattered grains* to be soundly used for reasoning in equivalence and call this *scattered grain concurrency* (Section 6).
- We give a space efficient algorithm that soundly checks *grain concurrency* for a pair of events, i.e. whether the two events are causally ordered or concurrent up to *grain equivalence* (Section 5), and a space efficient algorithm that soundly checks *scattered grain concurrency* based on all possible choices of scattered grains (Section 7).

2 PRELIMINARIES

A string over an alphabet Σ is a finite sequence of symbols from Σ. We use $|w|$ to denote the length of the string w and $w[i]$ to denote the i^{th} symbol in w. The concatenation of two strings w, w' will be denoted by ww'.

2.1 Trace Equivalence

Antoni Mazurkiewicz popularized the use of partially commutative monoids for modelling executions of concurrent systems[Mazurkiewicz 1987]. We discuss this formalism here. An independence relation over Σ is a symmetric irreflexive binary relation $\mathbb{I} \subseteq \Sigma \times \Sigma$. The Mazurkiewicz equivalence (or trace equivalence) relation induced by $\mathbb{I}$, denoted[1] $\equiv_{\mathcal{M}}$ is then the smallest equivalence over Σ^* such that for any two strings $w, w' \in \Sigma^*$ and for any two letters $a, b \in \Sigma$ with $(a, b) \in \mathbb{I}$, we have $wabw' \equiv_{\mathcal{M}} wbaw'$. A Mazurkiewicz trace is then an equivalence class of $\equiv_{\mathcal{M}}$.

Mazurkiewicz partial order. An equivalence class of $\equiv_{\mathcal{M}}$ can be succinctly represented using a partial order on the set of *events* in a given string. Events are unique identifiers for the different occurrences of symbols in a string. Formally, for a string w, the set of events of w, denoted Events_w is the set of pairs of the form $e = (a, i)$ such that $a \in \Sigma$ and there are at least i occurrences of a in w. Thus, an event uniquely identifies a position in the string — the pair (a, i) corresponds to the unique position $j \leq |w|$ such that $w[j] = a$ and there are exactly $i - 1$ occurrences of a before the index j in w. Observe that if w and w' are permutations of each other, then $\mathsf{Events}_w = \mathsf{Events}_{w'}$. For an event $e = (a, i)$, we use the shorthand $w[e]$ to denote the label a. Often, we will use the position j itself to denote the event (a, i) when the distinction is immaterial. Fix $w \in \Sigma^*$. The

[1]The equivalence is parametric on the independence relation $\mathbb{I}$. In this view, a notation like $\equiv^{\mathbb{I}}_{\mathcal{M}}$ would be more precise. In favor of readability, we skip this parametrization; the independence relation $\mathbb{I}$ will always be clear.

Mazurkiewicz (or trace) partial order for w, denoted $\prec_{\mathcal{M}}$ is then, the transitive closure of the relation $\{(e, f) \mid e, f \in \mathsf{Events}_w,\ e \text{ occurs before } f \text{ in } w\ \wedge\ (w[e], w[f]) \notin \mathbb{I}\}$.

For Mazurkiewicz traces, the corresponding partial order is a sound and complete representation of an equivalence class [Mazurkiewicz 1987].

2.2 Concurrent alphabet and dependence

For modeling runs or executions of shared memory multi-threaded concurrent programs, we will consider the alphabet consisting of reads and writes. Let us fix *finite* sets $\mathcal{T}$ and $\mathcal{X}$ of thread identifiers and memory location identifiers respectively. The concurrent alphabet Σ_{Conc} we consider in the rest of the paper is:

$$\Sigma_{\mathsf{Conc}} = \{\langle t, o, x\rangle \mid t \in \mathcal{T}, o \in \{\mathsf{r}, \mathsf{w}\}, x \in \mathcal{X}\}$$

For a symbol $a = \langle t, o, x\rangle \in \Sigma_{\mathsf{Conc}}$, we say $\mathsf{thr}(a) = t$, $\mathsf{op}(a) = o$ and $\mathsf{var}(a) = x$. A concurrent program *run* or *execution* is a string over Σ_{Conc}. For a run $w \in \Sigma^*_{\mathsf{Conc}}$ and event $e \in \mathsf{Events}_w$, we overload the notation and use $\mathsf{thr}(e)$, $\mathsf{op}(e)$ and $\mathsf{var}(e)$ in place of $\mathsf{thr}(w[e])$, $\mathsf{op}(w[e])$ and $\mathsf{var}(w[e])$ respectively. Since the focus of the rest of the article will be on concurrent program runs, we will omit the subscript Conc, and unless Σ is not explicitly defined, we will assume it is Σ_{Conc}.

We use the following independence (commutativity) relation:

$$\mathbb{I}_{\mathcal{M}} \quad = \quad \Sigma \times \Sigma - \{(a, b) \mid \mathsf{thr}(a) = \mathsf{thr}(b) \vee (\mathsf{var}(a) = \mathsf{var}(b) \wedge \mathsf{w} \in \{\mathsf{op}(a), \mathsf{op}(b)\})\} \tag{1}$$

This defines an appropriate trace monoid for the alphabet of concurrent program actions. We refer to the equivalence class of a concurrent program run w in this monoid as $[w]_{\mathcal{M}}$.

Next, for ease of notation, we will often denote labels as $\langle t, o(x)\rangle$ (for example $\langle t, \mathsf{w}(x)\rangle$) in place of the expanded version $\langle t, o, x\rangle$. We will also, at times, omit the thread identifier and use the shorthand $e = o(x)$ to denote that $\mathsf{op}(e) = o$ and $\mathsf{var}(e) = x$.

2.3 Reads-From Equivalence

A natural notion of equivalence of program runs in the context of shared memory multi-threaded program is *reads-from* equivalence. We formalize this notion here.

Program Order and Reads-from mapping. The program order, or thread order induced by a concurrent program run $w \in \Sigma^*$ orders any two events belonging to the same thread. Formally, $\mathsf{po}_w = \{(e, f) \mid e, f \in \mathsf{Events}_w, \mathsf{thr}(e) = \mathsf{thr}(f), e \text{ occurs before } f \text{ in } w\}$. The reads-from mapping induced by w maps each read event to the write event it observes. In our context, this corresponds to the last conflicting write event before the read event. Formally, the reads-from mapping is a partial function $\mathsf{rf}_w : \mathsf{Events}_w \hookrightarrow \mathsf{Events}_w$ such that $\mathsf{rf}_w(e)$ is defined iff $\mathsf{op}(e) = \mathsf{r}$. Further, for a read event e occurring at the i^{th} index in w, $\mathsf{rf}_w(e)$, is the unique event f occurring at index $j < i$ for which $\mathsf{op}(f) = \mathsf{w}$, $\mathsf{var}(f) = \mathsf{var}(e)$ and there is no other write event on $\mathsf{var}(e)$ (occurring at index j') such that $j < j' < i$. Here, and for the rest of the paper, we assume that every read event is preceded by some write event on the same variable.

Reads-from equivalence. Reads-from equivalence is a semantic notion of equivalence on concurrent program runs, that distinguishes between two runs based on whether or not they might produce different outcomes. We say two runs $w, w' \in \Sigma^*$ are *reads-from equivalent*, denoted $w \equiv_{\mathsf{rf}} w'$ if $\mathsf{Events}_w = \mathsf{Events}_{w'}$, $\mathsf{po}_w = \mathsf{po}_{w'}$ and $\mathsf{rf}_w = \mathsf{rf}_{w'}$. That is, for w and w' to be reads-from equivalent, they should be permutations of each other and must follow the same program order, and further,

every read event e must read from the same write event in both w and w'. Reads-from equivalence is a strictly coarser equivalence than trace equivalence for concurrent program runs. That is, whenever $w \equiv_{\mathcal{M}} w'$, we must have $w \equiv_{\mathsf{rf}} w'$; but the converse is not true.

Example 2.1. Consider the two runs (denoted σ and σ') shown in Fig. 1(a) and Fig. 1(b) respectively. Observe that σ and σ' have the same set of events, and program order ($\mathsf{po}_\sigma = \mathsf{po}_{\sigma'}$). Also, for each read event e, $\mathsf{rf}_\sigma(e) = \mathsf{rf}_{\sigma'}(e)$. This means $\sigma \equiv_{\mathsf{rf}} \sigma'$. Consider now the permutation of σ corresponding to the sequence $\sigma'' = e_1 \dots e_8 e_{10} e_9 e_{11} \dots e_{14}$, where e_i denotes the i^{th} event of σ from the top. σ'' does not have the same reads-from mapping as σ since $\mathsf{rf}_{\sigma''}(e_9) = e_{10} \neq e_7 = \mathsf{rf}_\sigma(e_9)$, and thus $\sigma'' \not\equiv_{\mathsf{rf}} \sigma$.

Soundness of Equivalence. The focus of this work is to develop equivalences that are coarser than Mazurkiewicz equivalence and are also *sound*, where soundness will be with respect to reads-from equivalence. Given a run $w \in \Sigma^*$ and a set $S_w \subseteq \Sigma^*$, we say that S_w is sound for w if $S_w \subseteq \{w' \mid w \equiv_{\mathsf{rf}} w\}$. Likewise, an equivalence relation $\sim$ over Σ^* is said to be sound if for every $w \in \Sigma^*$, the equivalence class $[w]_\sim$ is sound for w. We remark that that trace equivalence defined based on the independence relation $\mathbb{I}_{\mathcal{M}}$ is sound in this sense:

Remark 1. For a given concurrent program run $w \in \Sigma$, every member of $[w]_{\mathcal{M}}$ preserves the *program order* and *reads-from* relations induced by w.

3 CAUSAL CONCURRENCY UNDER READS-FROM EQUIVALENCE

In scenarios like dynamic partial order reduction in stateless model checking, or runtime predictive monitoring, one is often interested in the *causal relationship* between actions. Understanding causality at the level of program runs often amounts to answering whether there is an equivalent run that witnesses the inversion of order between two events.

The efficiency of determining causal concurrency is then key in designing efficient techniques in the aforementioned contexts. When deploying such techniques for monitoring large scale software artifacts exhibiting executions with billions of events, a desirable goal is to design monitoring algorithms that can be efficiently implemented in an online 'incremental' fashion that store only a constant amount of information, independent of the size of the execution being monitored [Roşu and Viswanathan 2003]. In other words, an ideal algorithm would observe events in an execution in a single forward pass, using a small amount of memory. This motivates our investigation of the efficiency of checking causal ordering.

We first formally define the relevant algorithmic questions in the context of any equivalence relation on concurrent program runs, and then study their complexity under reads-from equivalence.

3.1 Causal Concurrency and Ordering

Formally, let $\sim$ be an equivalence over Σ^* such that when two runs are equivalent under $\sim$, they are also permutations of each other. Let $w \in \Sigma^*$ be a concurrent program run, and let $e, f \in \mathsf{Events}_w$ be two events occurring at indices i and j (with $i < j$). We say that e and f in w are *causally ordered* under $\sim$ if for every $w' \sim w$, e and f appear in the same order as they do in w. If e and f are not causally ordered under an equivalence $\sim$, we say that they are causally concurrent under $\sim$.

The following are the key algorithmic questions we investigate in this paper.

Problem 3.1 (Checking Causal Concurrency Between Events)**.** Let $\sim$ be an equivalence relation over Σ^*. Given a program run $w \in \Sigma^*$, and two events $e, f \in \mathsf{Events}_w$, the problem of checking causal concurrency between events asks if e and f are causally concurrent under $\sim$.

In the context of many applications in testing and verification of concurrent programs, one often asks the following more coarse grained question.

Problem 3.2 (Checking Causal Concurrency Between Symbols)**.** Let $\sim$ be an equivalence relation over Σ^*. Given a program run $w \in \Sigma^*$, and two symbols (or letters) $c, d \in \Sigma$, the problem of checking causal concurrency between symbols (or letters) asks to determine if there are events $e, f \in \mathsf{Events}_w$ such that $w[e] = c$, $w[f] = d$ and e and f are causally concurrent under $\sim$.

If one has an oracle for deciding causal concurrency between symbols, then one can use it to check causal concurrency between events. In particular, assume $c = w[e]$ and $d = w[f]$ and consider the alphabet $\Delta = \Sigma \uplus \{c^{\diamond_1}, d^{\diamond_2}\}$, where $c^{\diamond_1}$ and $d^{\diamond_2}$ are distinct marked copies of the letters $c, d \in \Sigma$. Consider the string $w' \in \Delta^*$ with $|w'| = |w|$ such that $w'[g] = w[g]$ for every $g \notin \{e, f\}$, $w[e] = c^{\diamond_1}$ and $w[f] = d^{\diamond_2}$. Then, causal concurrency (respectively orderedness) of symbols $c^{\diamond_1}$ and $d^{\diamond_2}$ under $\sim'$ implies the causal concurrency (respectively orderedness) of events e and f under $\sim$, where the equivalence $\sim'$ is the same as $\sim$, modulo renaming letters $c^{\diamond_1}$ and $d^{\diamond_2}$ to c and d respectively.

3.2 Computational Hardness in Checking Concurrency

For the case of trace equivalence, more specifically under $\equiv_{\mathcal{M}}$, causal concurrency can be determined in a constant space streaming fashion. This result is somewhat known amongst the experts in the field, but it does not appear in the following specific way anywhere in the literature.

Proposition 3.1 (Causal concurrency for trace equivalence)**.** Given an input $w \in \Sigma^*$ and symbols $c, d \in \Sigma$, the causal concurrency between c and d under $\equiv_{\mathcal{M}}$ can be determined using a single pass constant space streaming algorithm.

In Section 5, we give a constructive proof to this lemma by presenting a constant space monitoring algorithm. Next, we show that the same is not achievable for the case of reads-from equivalence — we show that any algorithm for checking causal ordering (for the semantic notion of equivalence $\equiv_{\mathsf{rf}}$) must use linear space in a streaming setting.

Theorem 3.1 (Linear space hardness)**.** Any streaming algorithm that checks the causal concurrency of a pair of symbols under $\equiv_{\mathsf{rf}}$ in a streaming fashion uses linear space, even for program runs containing just 2 threads and 6 variables.

The key idea behind the proof of Theorem 3.1 is to exploit and generalize the intricate pattern in the runs in Fig. 1. The idea is that determining if two specific events are causally concurrent in such a pattern, relies on successively inferring the concurrency status of linearly many pairs of events, placed arbitrarily far away in the past and/or in the future, which is impossible for a one pass streaming algorithm that only uses sub-linear space.

In fact, we use similar ideas as in the proof of the above statement to also establish a lower bound on the time-space tradeoff for the problem of determining causal concurrency, even when we do not bound the number of passes:

Theorem 3.2 (Quadratic time space lower bound)**.** For any algorithm (streaming or not) that checks if a pair of symbols are causally concurrent under $\equiv_{\mathsf{rf}}$ in $S(n)$ space and $T(n)$ time on an input run of length n, we must have $S(n) \cdot T(n) \in \Omega(n^2)$.

The linear lower bound in Theorem 3.1 establishes non-regularity of any monitor for causal concurrency of symbols. We can further refine this result and show that the problem of determining causal concurrency of symbols is also not context-free. We establish the following result by invoking

a pumping lemma argument for context free languages on sets of runs that observe intricate patterns akin to the one in Fig. 1.

Theorem 3.3 (Non Context-freeness)**.** Let $c, d \in \Sigma$. There exists no nondeterministic pushdown automaton that accepts exactly the runs w such that both c and d appear in w and are causally concurrent in it under $\equiv_{\mathsf{rf}}$.

Finally, conditioned on the widely believed Strong Exponential Time Hypothesis (SETH) [Impagliazzo and Paturi 2001], we establish a quadratic time lower bound for Problems 3.1 and 3.2.

Theorem 3.4. Assume SETH. The problem of determining the causal concurrency of a pair of events in w under $\equiv_{\mathsf{rf}}$ cannot be solved in time $O(|w|^{2-\epsilon})$ for all $\epsilon > 0$, even when w has 2 threads.

The proof is established via a fine-grained reduction from the Orthogonal Vector Conjecture, which also admits a quadratic lower bound under SETH [Williams 2005]. The input to the Orthogonal Vectors (OV) problem is a pair of sequences $A, B \subseteq \{0,1\}^d$ of d-dimensional vectors, each of length n (i.e., $|A| = |B| = n$), and the output is YES iff there are two vectors $a \in A, b \in B$ such that their inner product is 0 (i.e., $a \cdot b = 0$). The OV hypothesis states that for every $\epsilon > 0$, there is no algorithm that solves the OV problem (with input instances of length n and dimension d, with $d \in \Omega(\log n)$) in $O(n^{2-\epsilon}\,\mathrm{poly}(d))$ time.

3.3 Upper Bounds for Checking Concurrency

In this section, we study the precise complexity of reasoning about concurrency under $\equiv_{\mathsf{rf}}$ and establish time and space upper bounds. First, we observe that both Problems 3.1 and 3.2 can be solved using an algorithm whose running time is a polynomial expression whose degree varies with the number of threads:

Theorem 3.5 (Polynomial time algorithm)**.** Let $w \in \Sigma$ be a run with $|w| = n$ and let e and f be two events in w. The problem of determining if e and f are causally concurrent under $\equiv_{\mathsf{rf}}$ can be solved in time $O(|\mathcal{T}| \cdot n^{|\mathcal{T}|+1})$. Similarly, given $c, d \in \Sigma$, the problem of determining if c and d are causally concurrent under $\equiv_{\mathsf{rf}}$ can be solved in time $O(|\mathcal{T}| \cdot n^{|\mathcal{T}|+2})$.

The proof is based on constructing a 'frontier graph' [Gibbons and Korach 1994], whose vertices represent subsets of Events_w which are downward closed with respect to po_w and rf_w, while edges represent valid extensions obtained by adding single events to the subsets.

The problem can also be solved using a linearly bounded Turing machine, which also implies that the language of runs that exhibit concurrency of two given events is a context sensitive language.

Theorem 3.6 (Linear Space Upper Bound)**.** Let $w \in \Sigma$ be a run with $|w| = n$ and let e and f be two events in w. The problem of determining if e and f are causally concurrent under $\equiv_{\mathsf{rf}}$ can be solved in deterministic space $O(n)$. Similarly, given $c, d \in \Sigma$, the problem of determining if c and d are causally concurrent under $\equiv_{\mathsf{rf}}$ can be solved in time deterministic space $O(n)$.

The proof relies on successively generating permutations of the input run, checking if they are equivalent to it and also invert the order of the given events, all in deterministic linear space.

4 GRAIN COMMUTATIVITY

In this section, we present a new stronger and more syntactic definition of equivalence for concurrent runs that can overcome the hardness results previously discussed. We build on the theory of traces, where equivalence is defined based on a commutativity relation on the alphabet of program actions. The new equivalence relation is defined based on an extended commutativity relation

that additionally allows commutating some pairs of *words* over the same underlying alphabet, and strictly weakens trace equivalence. First, let us briefly discuss that unchecked generalization in this direction can very quickly result in hardness.

Theorem 4.1. Let $\Sigma = \{a, b, c\}$ and let $\mathbb{I} = \{(a, bc), (bc, a), (b, c), (c, b)\}$. There exists no constant space monitor that given an input word $w \in \Sigma^*$ can decide whether the first occurrence of a and the last occurrence of c in w are ordered.

The idea of the proof is to focus on words of the form $ab^n c^m$, and a causal concurrency query that involves the a and the last occurrence of c. It can be argued that the two are causally concurrent if and only if $n \geq m$.

Note that here, $\mathbb{I}$ is a generalization of commutativity relation in trace equivalence ($\mathbb{I}_{\mathcal{M}}$) by what seems to be the smallest increment to a classic commutativity relation over letters: the commutativity of one word of length 2 (the shortest possible word that is not a letter) against one single letter. Yet, we immediately loose the constant-space checkability of concurrency/orderedness of pairs of events. Therefore, to maintain the property of constant-space checkability, one has to be careful with the generalization of $\mathbb{I}_{\mathcal{M}}$.

4.1 Partially-Commutative Grain Monoids

We define an equivalence relation based on commutativity of words.

Definition 4.1 (Grains)**.** A *grain* is simply a non-empty word in Σ^*.

For a grain g, let $\mathit{letters}(g)$ be the set of letters (from Σ) that appear in g. We can define a partially commutative monoid in the same style as *trace monoids* [Mazurkiewicz 1987] as follows:

Definition 4.2 (Grain Monoids)**.** Given a trace monoid $(\Sigma, \mathbb{I})$, a set of grains G induces the (partially commutative) *grain* monoid $(\Sigma_G \cup \Sigma, \widehat{\mathbb{I}}_G)$ where $\Sigma_G = \{a_g |\ g \in G\}$ and

$$\widehat{\mathbb{I}}_G = \mathbb{I} \cup \mathbb{I}_G \cup \{(a, a_g), (a_g, a) |\ a \in \Sigma \wedge a_g \in \Sigma_G \wedge \{a\} \times \mathit{letters}(g) \subseteq \mathbb{I}\}$$
$$\cup \{(a_g, a_{g'}), (a_{g'}, a_g) |\ a_g, a_{g'} \in \Sigma_G \wedge \mathit{letters}(g) \times \mathit{letters}(g') \subseteq \mathbb{I}\}$$

where $\mathbb{I}_G \subseteq \Sigma_G \times \Sigma_G$, the *grain commutativity* relation, is an arbitrary symmetric independence relation defined on the grains.

If $G \subseteq \Sigma$ and $\mathbb{I}_G = \emptyset$, then the induced grain monoid coincides with the trace monoid $(\Sigma, \mathbb{I})$. Otherwise, it can be viewed as a classic trace monoid on a new alphabet $\Sigma_G \cup \Sigma$. For this reason, it induces an equivalence relation $\equiv_{\widehat{\mathbb{I}}_G}$ on the set of words in $(\Sigma_G \cup \Sigma)^*$.

Recall that $\mathbb{I}_{\mathcal{M}}$ is defined in the context of the alphabet Σ to be *sound*, precisely in the sense that the induced $\equiv_{\mathcal{M}}$ preserves rf-equivalence. We need to determine when $\equiv_G$ is considered sound.

Definition 4.3 (Strict Soundness)**.** We call a grain commutativity relation $\mathbb{I}_G$ *strictly sound* if, $(g, g') \in \mathbb{I}_G$ iff for all $\alpha, \beta \in \Sigma^*$, we have $\alpha g g' \beta \equiv_{\mathrm{rf}} \alpha g' g \beta$.

It is straightforward to see that if we let $G = \Sigma$, then the independence relation that defines trace equivalence is strictly sound according to this definition. The less straightforward fact is that trace equivalence defines the largest such sound relation. To be precise,

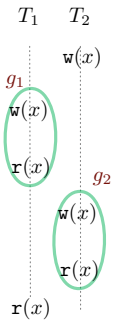
Proposition 4.1. If g and g' strictly soundly commute, then $gg' \equiv_{\mathcal{M}} g'g$.

On the one hand, strict soundness seems reasonable because it decouples commutativity from its *context*. On the other hand, it makes the grains seem pointless in the sense that they do not offer

any additional commutativity compared to classical trace monoids. The motivation behind forcing the soundness to be independent of the *context* (i.e. the choice of α and β in Definition 4.3) is that as one swaps two commutating grains, the *context* may change, and it would complicate reasoning substantially if the commutativity status of two other grains were to change as a result. In [Sassone et al. 1993], a generalized version of trace monoids are formulated that account for context. These have a sophisticated set of coherence and consistency conditions and have not been studied in any algorithmic contexts beyond being defined.

Fundamentally, we would like a commutativity relation that is sound in the sense that it maintains $\mathtt{rf}$-equivalence and defines an equivalence class in which the commutativity relation does not change from one member to another to keep things simple for formulating algorithms.

Example 4.1. Consider the program run on the right, with 2 threads. Two grains g_1 and g_2 have been marked. In isolation (as in if everything else from this run is ignored), these two grains soundly commute. But, the run illustrates that in the presence of the last $\mathtt{r}(x)$, the two grains do not soundly commute, and therefore they do not strictly soundly commute. Observe that the left context is irrelevant to the commutativity status of these two grains. Only the right context can violate soundness. If unsoundness is the result of the program order being broken, one would observe it by looking only at the two grains. Thus, any violations to soundness related to the context have to be related to the reads-from relation. Specifically, a write event $\mathtt{w}$ belongs to a grain, but there is at least one read event $\mathtt{r}$, which *reads* from it (i.e., $\mathtt{w} = \mathtt{rf}(\mathtt{r})$), that does not belong to the same grain. By formally disallowing any such bad right contexts for a pair of grains, one can define a more permissive commutativity relation. □



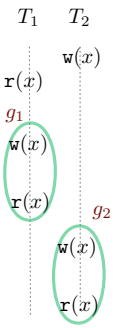
Example 4.1 illustrates why we put the fault in the definition of strict soundness mainly with the *right* context. To rule these scenarios out, one can restrict the right context from all possible contexts to those that cannot adversarially affect the commutativity status of g and g'.

Definition 4.4 (Sound Grain Commutativity)**.** We call a grain commutativity relation $\mathbb{I}_G$ *sound* if for all $g, g' \in G$, for all $x \in \mathsf{var}(g) \cap \mathsf{var}(g')$ where at least one $\mathtt{w}(x)$ appears in gg', and for all $\alpha, \beta \in \Sigma^*$ such that $\beta|_x \in L(\mathtt{w}(x)\Sigma^*)$, we have: $(g, g') \in \mathbb{I}_G \iff \alpha g g' \beta \equiv_{\mathtt{rf}} \alpha g' g \beta$

Definition 4.4 strictly weakens Definition 4.3 by limiting the (right) contexts in which commuting the actions must be sound. That is, every strictly sound grain commutativity relation is sound, but not all sound grain commutativity relations are strictly sound. Indeed, if g and g' do not strictly soundly commute, it is easy to construct a right context β in which they do not soundly commute.

Remark 2. Given a run w and another run u that can be acquired from w through a sequence of swaps defined by a *strictly* sound commutativity relation $\widehat{\mathbb{I}}_G$ (Definition 4.3), we have $w \equiv_{\mathtt{rf}} v$. This is not true for a sound commutativity relation (Definition 4.4).

Example 4.2. Recall the run illustrated in Example 4.1. According to Definition 4.4 and as discussed in Example 4.1, the two grains g_1 and g_2 soundly commute. But, clearly, the equivalence class of the run from Example 4.1 up to this commutativity relation is not sound. In contrast, the same two grains appear in the run illustrated on the right and the equivalence runs inferred by their commutativity are sound.



The difference is that the run on the right does not violate the condition about right contexts (from Definition 4.4) but the run from Example 4.1 does. □

It feels like we took one step forward by weakening the definition of soundness and then one step backward since it is not guaranteed to provide soundness in all contexts. There are, however, two key observations that make this definition of soundness a good fit in the context of our main goal, that is checking the status of concurrency of two given events. First, we are solely interested in the set of words equivalent to a single reference program run. Second, we may not have control over the choice of right contexts, but we do have control over the choice of grains and the commutativity relation $\mathbb{I}_G$. We can limit these choices based on the input run so that the equivalence class of the input run induced by the corresponding grain monoid is indeed sound. Therefore, we next focus on the equivalence class $[w]_G$ of a given word w and the choices for G (the set of grains) and the grain commutativity relation $\mathbb{I}_G$ that make $[w]_G$ sound.

4.2 Sound Grain Equivalence

First, observe that the same exact grain (which is a word) can appear several times as a subword in a particular word. In our formalism so far, we had no need of distinguishing the multiple instances, but since their right contexts may differ, we must do so now to attach different commutativity attributes to them.

Definition 4.5. A *valid* set of grains for a run $w \in \Sigma^*$ is set of indexed words of the form $g@i$ where g is a contiguous subword of w that appears at position i and no two grains overlap in w.

Consider a valid set of (indexed) grains G in a program run w. Since the indexed set G may only be a valid set for the run w in consideration, we cannot cleanly define the equivalence induced by G on the set of all runs Σ^*. However, we can still precisely define the class of runs that can be inferred by successively swapping adjacent grains from G. For ease of presentation, we will abuse the notation $[w]_G$ to denote this set, and will take the liberty to call it the *equivalence class of* w, when G is known from the context.

We can formalize $[w]_G$ as follows. Define $h_G : \Sigma^* \to (\Sigma_G \cup \Sigma)^*$ as the homomorphism that maps each indexed grain $g@i$ to the corresponding letter $a_g \in \Sigma_G$ and each letter in Σ, that does not belong to a grain, to itself. Let h_G^{-1} be the inverse homomorphism that replaces letters in Σ_G with the corresponding grain words. Then,

$$u \in [w]_G \iff \exists u' \in (\Sigma_G \cup \Sigma)^* : \left(u = h_G^{-1}(u') \wedge u' \equiv_{\widetilde{\mathbb{I}}_G} h_G(w)\right)$$

In short, the set of words that are considered equivalent to w are determined by those that are equivalent to its corresponding word in the grain monoid, for the specific choice of valid grains G. In prose, we say u is equivalent to w when $u \in [w]_G$ for some valid choice of grains G.

We lift the commutativity relation $\mathbb{I}_G$ to relate indexed words of the form $g@i$ as well. This will enable us to say that $(g@i, g'@j) \in \mathbb{I}_G$ while $(g@k, g'@j) \notin \mathbb{I}_G$. Note that corresponding grain monoid is defined as before, each new grain $g@i$ is mapped to a designated letter $a_{g@i}$.

Definition 4.6. For a word w and a set of valid grains G in w, $\mathbb{I}_G$ is sound if $[w]_G$ is sound (i.e. $[w]_G \subseteq [w]_{\mathrm{rf}}$).

Next we give necessary and sufficient conditions for soundness of $\mathbb{I}_G$. For an event $e = \mathtt{w}(x)$, let $\mathit{reads}(e)$ denote all events e' of the form $\mathtt{r}(x)$ that read from e. To lighten the notation whenever possible, we may refer to a grain only by its word g whenever the position is not of importance, or

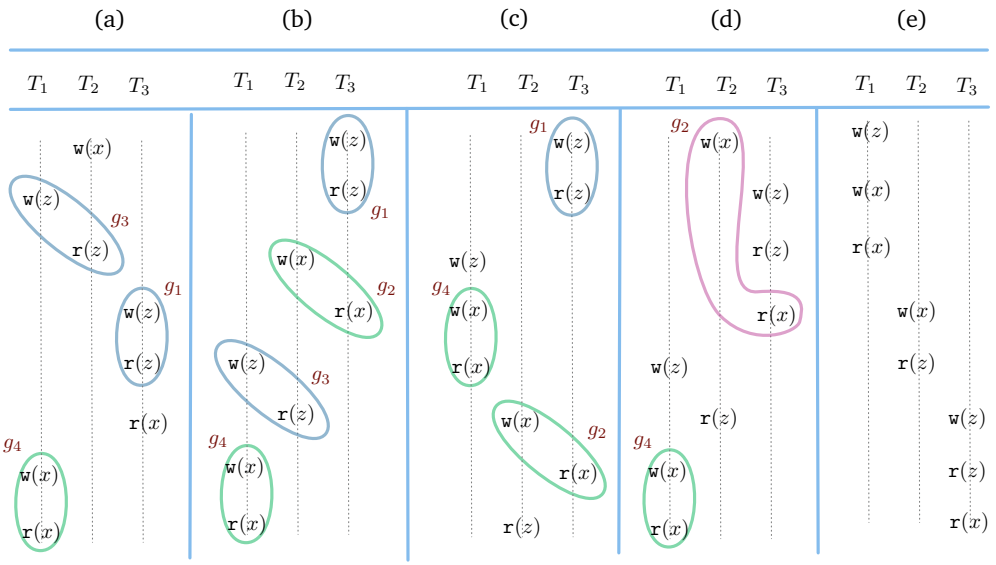


Fig. 3. Examples 4.3, 6.1, and 6.2.

implied from the context. We only specifically mention the position $g@i$ when it really matters. Define $\mathsf{op}(g, x)$, for a grain g to be the set of operations $(\{\mathtt{r}, \mathtt{w}\})$ of variable x that appear in g.

Theorem 4.2. In the context of a word w and a *valid* set of grains G, a commutativity relation $\mathbb{I}_G$ is sound iff for all $(g, g') \in \mathbb{I}_G$, $gg' \equiv_{\mathsf{rf}} g'g$ and for all $x \in \mathsf{var}(g) \cap \mathsf{var}(g')$ the following holds:

$$\big(\mathsf{op}(g,x) \cup \mathsf{op}(g',x) = \{\mathtt{r}\}\big) \vee \big(\forall e, f \cdot (e = \mathtt{w}(x), f = \mathtt{r}(x), f = \mathsf{rf}_w(e)) \implies (e \in g \iff f \in g)\big)$$

It is clear that if $gg' \equiv_{\mathsf{rf}} g'g$ does not hold, the resulting commutativity relation is unsound. Example 4.1 captures the idea of why the violation of the additional conditions also leads to unsoundness. Intuitively, these conditions express that if the grains share a variable and at least one writes to that shared variable, then once a write action is included in one grain then all reads that read from it also must be included in that grain. The proof of the other direction is a tedious case analysis.

Recall the program runs in Examples 4.1 and 4.2. Observe that even though the pairs of grains are identical, if we assume the commutativity of the pair of grains, then the choice in Example 4.2 satisfies the conditions of the above theorem, and the one in Example 4.1 does not; in particular, they violate the part that demands every read that reads from the same $\mathtt{w}(x)$ operation must belong to the grain.

4.3 The Expressive Power of $[w]_G$

We use an extended example to highlight how $[w]_G$ soundly enlarges the trace equivalence class of w, induced by the trace equivalence relation $\equiv_{\mathcal{M}}$.

Example 4.3. Consider the program runs illustrated in Fig. 3. They are all rf-equivalent. We observe that different choices of grains witness the equivalence of different pairs of the run from the figure. Independent of which grain is present in which subfigure, the only sound commutativity between grains, in addition to classic trace theory commutativity, is $\mathbb{I}_G = \{(g_1, g_3), (g_3, g_1), (g_2, g_4), (g_4, g_2)\}$.

First, focus on Fig. 3(a), and observe that g_1 and g_3 soundly commute (in the sense of Theorem 4.2). Yet g_4 does not commute with anything that it would not otherwise under the classic trace monoid through the commutativity of its individual events. For example, taking the single $\mathtt{w}(x)$ of thread T_2 or the single $\mathtt{r}(x)$ of thread T_3 as grains, one would violate the conditions of Theorem 4.2) if one were to declare either event commutative against g_4. With the grains marked in Fig. 3(a), the run illustrated in Fig. 3(a) is equivalent to the one in Fig. 3(b) — starting from (a), we can swap g_1 and g_3 first, and then g_1 against the $\mathtt{w}(x)$ of thread T_2 and g_3 against the $\mathtt{r}(x)$ of thread T_3.

Now consider the set of grains in Fig. 3(b). It is a superset of grains marked in Fig. 3(a), and yet, in this configuration we have less freedom of movement. $\mathbb{I}_G$ is still sound, but since (g_1, g_2), (g_2, g_3), and (g_3, g_4) do not commute, this run effectively belongs to an equivalence class of size 1. Specifically, we cannot conclude that it is equivalent to the run illustrated in Fig. 3(a). If we remove grain g_2, then the equivalence class gets larger, and includes the run illustrated in Fig. 3(a). Observe that, therefore:

Having more grains does not necessarily *imply having a larger equivalence classes.*

Similarly, if we take the set of grains in Fig. 3(c), then the run in Fig. 3(c) is equivalent to the one in Fig. 3(b). But, note that there is no possible choice of grains in Fig. 3(c) that would make it equivalent to the run in Fig. 3(a), and vice versa. The two runs are clearly rf-equivalent. They are grain-equivalent to the run in Fig. 3(b). Yet, for no choice of grains they can be made grain-equivalent to each other.

If $v \in [u]_G$ and $w \in [v]_{G'}$, there may not exist a sound G'' such that $w \in [u]_{G''}$.

Example 4.3 illustrates that depending on the input run w, or the choice of events for a query of concurrency, a different choice of grains G may be suitable to define the appropriate $[w]_G$.

Definition 4.7 (Grain Concurrency)**.** Consider a program run w and two events e and e' that appear in w. We call the pair of events e and e' to be *grain concurrent* iff there exists a sound set of grains G and a commutativity relation $\mathbb{I}_G$ in the context of w such that, there exists $u \in [w]_G$ in which e and e' are reordered with respect to their order of appearance in w.

Note that for any given program run w and any choice of valid grains G and a sound commutativity relation $\mathbb{I}_G$, $[w]_G$ is always, by definition, a superset of the trace equivalence class of w. Moreover, if we let $G = \Sigma$, then $[w]_G$ coincides with the trace equivalence class of w, since the $\mathbb{I}_{\mathcal{M}}$ is by definition sound. As such, any two events that are concurrent according to trace equivalent are also grain concurrent.

In Section 5, we formally argue that grain concurrency can be checked in constant space by giving a construction for a monitor that is strictly more expressive than a constant-space monitor based on $\equiv_{\mathcal{M}}$. For example, our monitor would declare that both the pair of $\mathtt{w}(x)$ and the pair of $\mathtt{w}(z)$ operations in the run illustrated in Fig. 3(b) can be soundly reordered, while they are strictly ordered according to $\equiv_{\mathcal{M}}$.

5 GRAIN CONCURRENCY MONITOR

Grain monoids are closely related to trace monoids. Therefore, we begin by defining a monitor that in constant space checks whether two events in an input run are concurrent according to $\equiv_{\mathcal{M}}$, and then present the grain concurrency monitor as an extension of this monitor.

To have a simple setup, we augment our alphabet Σ with two new symbols $\diamond_1$ and $\diamond_2$ that are assumed to appear precisely once each in any input word, marking the two events that are meant

State	Event	State Update
$\langle -, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle -, \mathrm{C}\rangle$
$\langle -, \mathrm{C}\rangle$	$\diamond_1$	$\langle \diamond_1, \mathrm{C}\rangle$
$\langle \diamond_1, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \diamond_1 -, \{e\}\rangle$
$\langle \diamond_1 -, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \diamond_1 -, \mathrm{C} \odot e\rangle$
$\langle \diamond_1 -, \mathrm{C}\rangle$	$\diamond_2$	$\langle \diamond_1 - \diamond_2, \mathrm{C}\rangle$
$\langle \diamond_1 - \diamond_2, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \mathit{Ord}(\mathrm{C}, e), \mathrm{C}\rangle$
$\langle \mathit{false}, \mathrm{C}\rangle$	$e \in \Sigma$	$\langle \mathit{false}, \mathrm{C}\rangle$

$$\mathrm{C} \odot e = \begin{cases} \mathrm{C} \cup \{e\} & \text{if } \exists e' \in \mathrm{C} : \ (e, e') \in \mathbb{D}_{\mathcal{M}} \\ \mathrm{C} & \text{owise} \end{cases}$$

$$\mathit{Ord}(\mathrm{C}, e) \iff \exists e' \in \mathrm{C} : \ (e, e') \in \mathbb{D}_{\mathcal{M}}$$

Fig. 4. Trace Concurrency Monitor: The monitor starts in state $\langle -, \varnothing\rangle$ and accepts if in a state $\langle \mathit{false}, \mathrm{C}\rangle$ (for any C) once the input is read. The operation $\odot$ updates C based on a new event.

to be checked for concurrency; these would be the events that immediately succeed $\diamond_1$ and $\diamond_2$. The regular expression $\Sigma^* \diamond_1 \Sigma^+ \diamond_2 \Sigma^+$ captures that the the two $\diamond$'s are properly placed in an input run. Therefore, in the description of our main monitor, we assume that the input run is already well-formed in this sense.

The high level idea behind the monitor is simple. The monitor idles until it sees the first marker $\diamond_1$ and the event e_1 that it marks. Afterwards, it maintains a summary of the set of operations, reads or writes to specific variables by specific threads, seen so far that are *ordered* wrt to e_1. When the monitor comes across $\diamond_2$ and therefore identifies the second event e_2, it can use the summary to determine if e_1 and e_2 are causally concurrent or ordered.

The key to constant-space implementability is that the monitor does not have to remember all individual events, but rather what variables and threads are involved. This is based on the observation that to determine if two events commute, it suffices to know what variables are being accessed, what the nature of the access is, and to what threads the two events belong. The summary can be maintained in the most compact manner if the list of operations $\mathsf{op}(x)$ and the list of thread identifies are kept separately. But, we present the less compact version that maintains the summary as a set of events, since it is easier to generalize this version of the monitor to *grains*.

Formally, the monitor's state is pair $\langle d, \mathrm{C}\rangle$ where $\mathrm{C} \subseteq \Sigma$ is a set of labels. The first element of the pair d is used to track whether the monitor has seen events e_1 and e_2 yet. It encodes the six distinct stages: $-$: before the first diamond, $\diamond_1$: right after the first diamond, $\diamond -$: after e_1 has been recorded, $\diamond_1 - \diamond_2$: right after the second diamond is seen, and *true*/*false*: depending on the monitors decision to accept/reject after reading e_2. Fig. 4 lists the transitions of the monitor and the functions used. Since the result is folklore, we forgo giving a proof for the correctness of this monitor.

5.1 A Monitor for a Fixed Set of Grains G

We introduce our monitor based on grains in two stages, for the simplicity of presentation. First, we assume a set of grains is pre-decided and pre-marked in an input word, and present the core idea behind monitoring causal concurrency in this setup. Then, in Section 5.2, we build the final grain concurrency monitor as a generalization of this one.

Assume that Σ is further extended with a pair of symbols $\triangleright$ and $\triangleleft$, which are used as delimiters to mark grain boundaries. Any letter that appears outside the range of these delimiters is treated as a standalone event. For example, the program run form Fig. 3(a) with the marked grains becomes:

$$\langle T_2, \mathtt{w}(x)\rangle \triangleright \langle T_1, \mathtt{w}(z)\rangle\langle T_2, \mathtt{r}(z)\rangle \triangleleft \triangleright\langle T_3, \mathtt{w}(z)\rangle\langle T_3, \mathtt{r}(z)\rangle \triangleleft \langle T_3, \mathtt{r}(x)\rangle \triangleright \langle T_1, \mathtt{w}(x)\rangle\langle T_1, \mathtt{r}(x)\rangle\triangleleft$$

The two events of interest are marked with $\diamond$'s, as before. Except that if either event belongs to a grain, then the diamond must mark the entire grain. For example, if we want to determine whether the two $\mathtt{w}(x)$ events are ordered in the program run above, the diamonds would mark the first one as before, and the second one behind the left grain delimiter like this:

$$\diamond_1\langle T_2, \mathtt{w}(x)\rangle \rhd \langle T_1, \mathtt{w}(z)\rangle\langle T_2, \mathtt{r}(z)\rangle \lhd \rhd\langle T_3, \mathtt{w}(z)\rangle\langle T_3, \mathtt{r}(z)\rangle \lhd \langle T_3, \mathtt{r}(x)\rangle \diamond_2 \rhd\langle T_1, \mathtt{w}(x)\rangle\langle T_1, \mathtt{r}(x)\rangle\lhd$$

Note that the events in a grain always move together. Therefore, one cannot have a verdict that an event e (e.g. the first $\mathtt{w}(x)$ above) is concurrent with some arbitrary event f of a grain (e.g. the second $\mathtt{w}(x)$ above), while it is ordered wrt another event f' of the same grain (e.g. the second $\mathtt{r}(x)$ above). The following (revised) regular expression captures all the input runs in which grains and $\diamond$'s are marked properly, and therefore, we do not make it part of the design of the monitor:

$$\left((\rhd\Sigma^+\lhd) + \Sigma\right)^* \diamond_1 \left((\rhd\Sigma^+\lhd) + \Sigma\right)^+ \diamond_2 \left((\rhd\Sigma^+\lhd) + \Sigma\right)^+ \tag{WF}$$

To generalize the monitor in Fig. 4 to work with grains, we face two sources of complications. First, grains are arbitrary words in Σ^+, and even though Σ is finite, Σ^+ is infinite in size. Second, there are (potentially) infinitely many sound commutativity relations that can be inferred over the unbounded set of potential grains.

A simple observation helps overcome the first problem. Fundamentally, we are interested in grains because we are interested in the commutativity properties of these grains. Theorem 4.2 captures what information is relevant to make a sound decision about the commutativity of two grains. According to Theorem 4.2, g and g' soundly commute if $gg' \equiv_{\mathsf{rf}} g'g$ **and** for every variable x accessed in g (respectively g'), where x is written in at least one of the two grains, the following predicate is true:

$$\mathsf{complete}(g, x) \stackrel{\text{def}}{=} \left(\forall e, f \cdot (e = \mathtt{w}(x), f = \mathtt{r}(x), e = \mathsf{rf}_w(f)) \implies (e \in g \iff f \in g)\right) \tag{2}$$

In other words, two grains commute, if they commute according to $\equiv_{\mathsf{rf}}$ in isolation and if the share a variable that is written by at least one of them, then both grains must be *complete* wrt to that variable. We now introduce the *signature* of a grain as a pair of a set of letters that appear in the grain and a set of variables wrt which the grain is complete:

$$\forall w \in \Sigma^+ : \ \mathit{grain}(w) \stackrel{\text{def}}{=} \langle \mathit{letters}(w), \{x \in \mathcal{X} \mid \mathsf{complete}(w, x)\}\rangle$$

which contains all the information required for deciding commutativity of two given grains. More importantly, observe that there are boundedly many different grain signatures, $2^{|\Sigma|+|\mathcal{X}|}$ to be exact. Therefore, in the summary (C in Fig. 4), rather than keep track of the grain as an arbitrary size word, we maintain a set of grain signatures, which is very much bounded.

Let us turn our attention to the second problem. For any pair of grains in a word, one can look at their signatures and soundly decide whether they commute or not. Yet, there are unboundedly many such grains, and therefore, unboundedly many such commutativity relations to enumerate for the definition of *grain concurrency* (Definition 4.7).

Observe that commutativity and causal concurrency are monotonically related: the larger the grain commutativity relation, the larger the set of pairs of grain concurrent events. Therefore, rather than enumerate all possible commutativity relations, one can conservatively choose the largest one. In this largest relation, any two grains, that can soundly commute, are assumed to be commuting. This is determined based on their signatures alone, and therefore, the number of possible choices for the *largest* commutativity relation is bounded because the number of distinct signatures is bounded.

The Monitor. One can conceptually think about the monitor combining the following two passes into a single pass through nondeterminism:

- Pass 1: From right to left, analyze the grains and replace each with a fresh letter (corresponding to its signature), and learn the bounded maximal commutativity relations for these new letters. Intuitively, this pass finds out which grains are complete wrt which variables, constructs their signature, and replaces the grains with a new letter that encodes the information from the signature. Constructing signatures is straightforward in a left-to-right or right-to-left pass, but it is more straightforward to see why completeness (condition 2) can be checked and encoded for each grain in a right-to-left pass: a violation of this condition manifests as a pending read's matching write appearing in a grain.
- Pass 2: In the style of the trace concurrency monitor (Fig. 4)), in a left to right pass, decide the causal concurrency of the two entities marked by $\diamond$'s based on the original letters and the new letters computed during pass 1.

Classic ideas from automata theory provide the recipe to combine these passes, through nondeterminism, into a single-pass (left to right) constant space monitor that decides causal concurrency between any two events based on the largest sound grain commutativity relation:

Theorem 5.1. For a fixed set of valid grains G and a largest sound commutativity relation $\mathbb{I}_G$, the monitor sketched above accepts a program run $u \diamond_1 e\, v \diamond_2 e' w$ iff e and e' are reordered in some member of $[u \diamond_1 e\, v \diamond_2 e' w]_G$.

The proof together with the details of the monitor are presented in [Farzan and Mathur 2023].

5.2 Grain Concurrency Monitor

We are now ready to construct the monitor that precisely captures Definition 4.7. This monitor nondeterministically guesses the $\triangleright$ and $\triangleleft$ symbols and therefore the grain boundaries, and for each guess runs the monitor outlined in the previous section. It has to maintain a state to make sure that the grains are well-formed (non-overlapping) and non-empty. Therefore it effectively makes a guess and checks that its guess belongs to the language of the regular expression WF. Note that this guessing must account for the $\diamond$'s. The monitor makes a guess that an event of interest will be in a grain that it just nondeterministically opened, and therefore marks it with a diamond. Naturally, all wrong guesses are refuted later when the grain closes without seeing an event of interest.

Theorem 5.2. There exists a monitor that can decide, in constant space, the (causal) grain concurrency (respectively orderedness) of two events in a given program run.

6 SCATTERED GRAINS

So far, we have formally defined grains as subwords of a concurrent program run. Let us revisit our examples from Fig. 3 to motivate expanding the definition to include grains that do not appear as contiguous subwords; we call these *scattered grains*.

Example 6.1. To argue for the equivalence of the run illustrated in Fig. 3(d) to the one in Fig. 3(c), we need the $\mathtt{w}(x)$ of T_2 to first commute as an individual event (from (d) to (b)), and then move as part of the grain that is marked in Fig. 3(c). If we are permitted to consider the *scattered grain* g_2 as a grain in Fig. 3(d) (marked in pink), then we can argue that $\mathtt{w}(x)$ of thread T_2 can be reordered against $\mathtt{w}(x)$ of thread T_1 by (eventually) swapping the corresponding grains g_2 and g_4.

The grain monitor we present in Section 5 cannot keep track of these dual roles. Starting from the run illustrated in Fig. 3(d), it cannot see the potential of the grain including the $\mathtt{w}(x)$ of T_2 and $\mathtt{r}(x)$

of T_3 forming after a few sound swaps, and therefore cannot reason that $\mathtt{w}(x)$ of T_2 can ultimately be soundly reordered against $\mathtt{w}(x)$ of T_1. □

Formally, we say $\mathbf{i}$ is subsequence of the sequence of the range $[1..n]$ if it is strictly increasing, and all its elements belong to $[1..n]$. For convenience, we treat subsequences as sets of their elements (without order) when appropriate.

Definition 6.1 (Scattered Grains)**.** A *scattered grain* of program run w is a subsequence of w. To distinguish identical scattered grains from each other, we denote them as $g@\mathbf{i}$, where $\mathbf{i}$ is a subsequence of $[1..|w|]$ and identifies the position of g. A set of scattered grains G for a word $w \in \Sigma^*$ is said to be *valid* if no two distinct grains overlap, that is, $g_1@\mathbf{i}_1 \neq g_2@\mathbf{i}_2 \in G \implies \mathbf{i}_1 \cap \mathbf{i}_2 = \emptyset$.

Observe that scattered grains generalize the definition of grains when the subsequences happen to be contiguous. We may refer to a scattered grain simply as g rather than $g@\mathbf{i}$ whenever the position is unimportant or clear from the context. Sound commutativity relations over scattered grains are defined identically to contiguous grains, therefore all definitions and theorems from Section 4 hold. Moreover, we assume that single events form grains of size one and therefore every event belongs to some grain in what follows.

Definition 6.2 (Grain Graph of a Run)**.** Let G be a valid set of scattered grains for a program run w and $\mathbb{I}_G$ be the largest sound commutativity relation over G in the context of w. The grain graph $\mathcal{G}_{w,G} = (V, E)$ is a directed graph defined with the set of nodes $V = G$ and the set of edges

$$E = V \times V - \{(v_1, v_2) \mid v_1 = v_2 \vee (v_1, v_2) \in \widehat{\mathbb{I}}_G \vee w|_{v_1,v_2} \equiv_{\mathcal{M}} v_2 v_1\}$$

where $w_{v_1,v2}$ is the projection of w to the content of the grains v_1 and v_2.

The first two sets of excluded edges correspond to the classic notions of anti-reflexivity and independence. The third condition above determines when there is an edge between two scattered grains that are *entangled*. We want a directed edge only if the second grain cannot be safely commuted to before the first grain.

Grain graphs can be used to define a notion of concurrency based on a set of scattered grains in the following sense:

Definition 6.3 (Grain Graph Concurrency)**.** Let w be a run, G be a valid set of scattered grains in w and $\mathcal{G}_{w,G}$ be the corresponding grain graph. Let e_1 and e_2 be events in w such that e_1 appears before e_2 in w. We say that the events e_1 and e_2 are *grain graph concurrent* under G if there is no path in $\mathcal{G}_{w,G}$ from the node containing e_1 to the node containing e_2.

We call a valid set of scattered grains G and the corresponding commutativity relation $\mathbb{I}_G$ *sound* in the context of a run w if the same conditions listed in Definition 4.4 hold.

Observe that for contiguous grains, soundness of grain concurrency was baked into the definition that $[w]_G$ is sound. With scattered grains, this is no longer the case, and hence we need the following theorem:

Theorem 6.1. (Soundness of Grain Graph Concurrency) Let w be a run, G be a valid set of scattered grains in w. If a pair of events e_1 and e_2 are grain graph concurrent under G, then they appear in a different order in some run u such that $u \equiv_{\mathsf{rf}} w$.

The proof relies on the construction of the *condensation* of $\mathcal{G}_{w,G}$; that is, the directed *acyclic* graph acquired from $\mathcal{G}_{w,G}$ by contracting all its maximal strongly connected components. The proof argues that any linearization of this *condensed* graph is rf-equivalent to w. This, in turn, means

that the condensed graph is analogous to a partial order representing a an equivalence class of $\equiv_{\mathcal{M}}$ induced by the trace commutativity relation $\mathbb{I}_{\mathcal{M}}$. However, it differs from it in two important ways: (1) even though every linearization is rf-equivalent to w, it is not guaranteed to be equivalent to w up to a sequence of valid grain and letter swaps, and (2) the set of linearizations does not necessarily include everything that is (grain) equivalent to w, even possibly w itself.

Consider the grains illustrated in Fig. 5. They can be used to argue that the first $\mathtt{w}(x)$ and the last $\mathtt{w}(y)$ are grain graph concurrent. The grain graph only has one edge between grains g_2 and g_3, since all other pairs commute. However, observe that because g_2 is somewhat *entangled* with g_1, there exists no swap sequence to witness this concurrency. Moreover, the illustrated run itself does not belong to any linearization of the (condensed) grain graph, since no such linearization can reproduce the *entanglement* of the grains of the illustrated run.

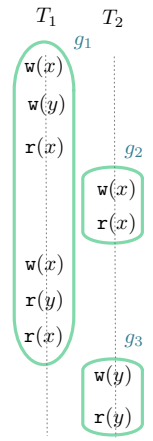


Fig. 5. Entangled Grains

Even though contiguous grains are a special case of scattered grains, the way we define *concurrency* in the two cases are fundamentally different in Definitions 4.7 and 6.3. Yet, for a set of contiguous grains, *grain graph concurrency* coincides with *grain concurrency*.

Theorem 6.2. For a program run w and a sound valid set of contiguous grains G, a pair of events e and e' are *grain concurrent* under G iff they are *grain graph concurrent* under G.

This is the consequence of the fact that $\mathcal{G}_{w,G}$ is an acyclic graph for a valid set of *contiguous* grains G, and as such the condensed graph and $\mathcal{G}_{w,G}$ coincide, and are identical to the partial order describing the same equivalence class in the corresponding grain monoid, for which a valid swap sequence can be constructed.

As with Definition 4.7, two events may be graph grain concurrent under one choice of scattered grains G but not under another choice G'. Consequently we define the following more permissive notion of concurrency under scattered grains.

Definition 6.4 (Scattered Grain Concurrency)**.** Consider a program run w and two events e and e' that appear in w. We call the pair of events e and e' to be *scattered grain concurrent* if there exists a valid set of scattered grains G for w such that e and e' are grain graph concurrent under G.

The Theorem 6.2 also implies that *scattered grain concurrency* properly subsumes *grain concurrency*. In Section 7, we demonstrate how *scattered grain concurrency* can be monitored in constant space. With the following example, we make the observation that even though *scattered grain concurrency* is strictly weaker than *grain concurrency*, it strictly under-approximates *sound concurrency* defined based on rf-equivalence.

Example 6.2. There remains a fundamental gap between the notion of concurrency defined based on rf-equivalence and scattered grain concurrency: in the run in Fig. 3(b), no choice of grains would witness the fact that the $\mathtt{w}(z)$ operation of thread T_3 can be soundly reordered against the $\mathtt{w}(x)$ operation of thread T_1. If all 4 grains are present, then the events are ordered. If we take either g_1 or g_3 out, then they become ordered through the conflict dependencies between the x operations. If we take either g_2 or g_4 out, then they become ordered through the conflict dependencies between the z variables. Yet, the rf-equivalent run in Fig. 3(e) witnesses that they are soundly concurrent.

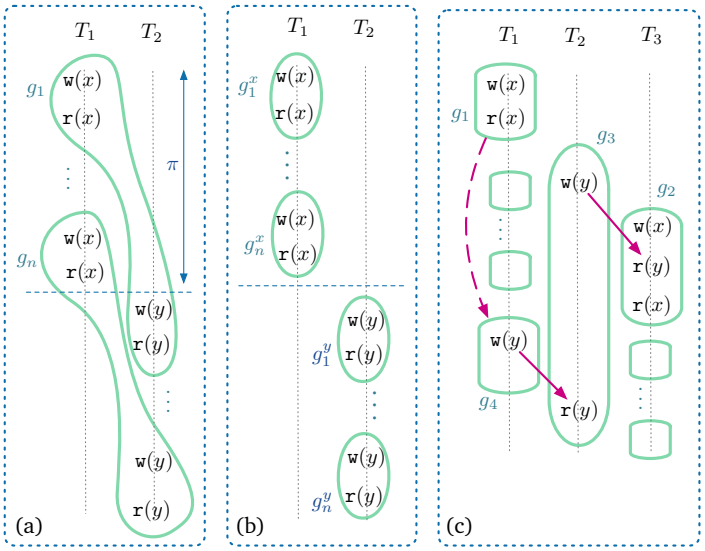


Fig. 6. The challenges of monitoring with scattered grains: (a) Unboundedly many active grains, (b) Minimal grains, and (c) Retroactive paths.

Interestingly, if we focus on the run in Fig. 3(e), and assume all grains g_1, g_2, g_3, and g_4 are present in it as scattered grains, then we can reason using the induced grain graph that the run in Fig. 3(b) is linearization of its (condensed) grain graph and as such $\mathtt{w}(x)$ of thread T_1 is (scattered) grain concurrent with $\mathtt{w}(z)$ of T_3. Therefore, in the non-swap-based notion of *scattered grain concurrency*, one can reason about the implied equivalence and the corresponding notion of the runs in Figures 3(b,e) in one way but the inverse. □

7 MONITORING WITH SCATTERED GRAINS

In this section, we develop a monitor for checking concurrency of two events in the presence of scattered grains. When grains are scattered, they can be interleaved in the run w, and this poses fundamental challenges towards the design of a constant space monitor.

We call a grain *active* in a prefix of a concurrent run, if part of the grain has appeared in the prefix, but it is has not appeared in its entirety. With contiguous grains, at most one grain can be active at any given time. In sharp contrast, the number of scattered grains that may be *active* simultaneously, can be unbounded (i.e., not constant). Consider the run and the scattered grains $G = \{g_1, \ldots, g_n\}$ marked in Fig. 6(a). Observe that all grains $g_1, g_2, \ldots g_n$ are active in the prefix π. The unboundedness of the number of active grains is problematic because one expects that a monitoring algorithm for checking concurrency would need to at least keep track of all active grains. This is the first challenge that has to be overcome in designing a constant space monitor for scattered grain concurrency.

The second challenge arises because a single active grain may overlap with many other active grains through its lifetime, even if at a given point only boundedly many grains are active. This means that two grains can be observed as ordered witnessed by a path in the grain graph (Definition 6.2), but this path is completed by a grain that appears long after the lifespan of both the grains have ended. Consider, for example, the run in Fig. 6(c). Before the grain g_4 appears, there is a path, namely the direct edge, from g_3 to g_2, but no path from g_1 to g_2. In fact, right before g_4 appears, g_1 and g_2 have both become inactive. When g_4 appears, a path is formed from g_1 to g_4. Once the $\mathtt{r}(y)$ event in g_3

appears, an edge is formed between grain g_4 and grain g_3, which now completes the path from g_1 to g_2 *retroactively*. Accounting for such retroactive paths is necessary for soundness.

We present solutions to these two challenges in Sections 7.1 and 7.2. We introduce the notion of a *minimal* grain to deal with the problem of unboundedly meany active grains and then demonstrate how a monitor can track retroactive paths by *summarizing* the paths between active grains when an intermediate grain has finished.

7.1 Bounding the number of Active Scattered Grains

We start by defining a class of grains that are *minimal*, with the idea that if all (scattered) grains are minimal, then the number of active grains in any prefix of the program run is bounded.

Definition 7.1 (Minimal Grain)**.** A (scattered) grain g is said to be minimal in program run w if for all $\rho, \sigma \neq \epsilon$ such that $g = \rho\sigma$, there is a read event $\mathtt{r} \in \sigma$ where $\mathrm{rf}_w(\mathtt{r}) \in \rho$. We call a set of grains minimal in w if all grains in it are minimal in w.

Intuitively, a grain is minimal if it cannot be broken into simpler grains, without worsening its commutativity status with respect to other grains. In Fig. 6(a), none of the grains $g_1, \ldots, g_n$ are minimal, because each of them have a prefix (of size 2) where there is no read event whose corresponding write event is not in the prefix grain. In contrast, the following regular expression can produce arbitrarily long minimal grains:

$$\mathtt{w}(x)\mathtt{w}(y)\mathtt{r}(x)\Big(\mathtt{w}(x)\mathtt{r}(y)\mathtt{w}(y)\mathtt{r}(x)\Big)^*$$

For a valid set G of grains in w and a prefix π of w, we use the notation $\mathrm{Active}_{\pi,G}$ to denote the set of grains in G that are active in π. If all grains are minimal, then the set of active grains is bounded in size by the total number of variables:

Lemma 7.1. Let w be a run and let G be a set of minimal grains in w. For a prefix π of w, we have $|\mathrm{Active}_{\pi,G}| \leq |\mathcal{X}|$.

This is implied by the following crucial observation. Assume two minimal grains g and g' are active at given prefix π of w. The minimality g (respectively g') implies that there is a variable x (respectively x') that is written in π and has a corresponding read in the remainder of w. The variables x and x' cannot coincide. Therefore one needs a distinct variable that only belongs to one of the many active grains at any given point, which puts a bound of $|\mathcal{X}|$ on the maximum number of grains that can be active at any given time.

Ideally, we want all grains to be minimal, since this resolves the problem of having unboundedly many active grains. Let us argue why this can be achieved without any compromises to the result of checking causal concurrency. Consider Fig. 6(b) which depicts the same run as in Fig. 6(a) but this time with a different set of grains which are all minimal. Observe that the set of minimal grains in Fig. 6(b) witness more causal concurrency than the set of non-minimal grains in Fig. 6(a). In general, one can argue that for any set of (non-minimal) grains that witnesses the causal concurrency of two events e and f, there exists a set of minimal events that does the same. We give a constructive argument for this claim.

One can argue that any grain g is the concatenation of a sequence of minimal grains. Recall Definition 7.1. For any ρ and σ that witness non-minimality of $g = \rho\sigma$ according to Definition 7.1, add a split point between ρ and σ. Let $g = g_1 \ldots g_n$ where g_i's are precisely marked by these split points. By definition, g_i's are all minimal. Define $\mathrm{split}(G) = \{g_1, \ldots, g_n\}$.

Given a valid set of grains G, we use $\mathsf{split}(G) = \bigcup_{g \in G} \mathsf{split}(g)$ to denote the set of grains obtained by splitting individual grains in G. We need to argue that splitting all grains into minimal grains would result in declaring at least as many pairs of events causally concurrent as before.

Given a commutativity relation $\mathbb{I}_G$ on G, define $\mathsf{split}(\mathbb{I}_G) \subseteq \mathsf{split}(G) \times \mathsf{split}(G)$ as

$$\mathsf{split}(\mathbb{I}_G) = \{(g_1', g_2') \mid \exists g_1, g_2 \in G, (g_1, g_2) \in \mathbb{I}_G \text{ and } g_1' \in \mathsf{split}(g_1), g_2' \in \mathsf{split}(g_2)\}$$

One can prove that $\mathsf{split}(\mathbb{I}_G)$ is sound. This in turn implies that splitting grains does not add spurious paths in the new grain graph, which did not exist in the original one.

Lemma 7.2. Let w be a run, G be a valid set of scattered grains, and $\mathbb{I}_G$ be a sound independence relation (Definition 4.6). $\mathsf{split}(\mathbb{I}_G)$ is sound, and for every pair of events (e_1, e_2), if e_1 and e_2 are grain graph concurrent under G (using commutativity relation $\mathbb{I}_G$), then they are grain graph concurrent under the grains $\mathsf{split}(G)$ (using commutativity relation $\mathsf{split}(\mathbb{I}_G)$).

Therefore, when checking scattered grain concurrency, it is safe to ignore all sets of grains that include any non-minimal grains, since the same causal concurrency verdicts can be declared by other sets of minimal grains.

7.2 Tracking Retroactive Paths

Let us address our second challenge, that is how to keep track of *retroactive* paths in the grain graph. We fix the set of minimal grains. Since grains containing only a single event are by definition minimal, we can assume, without loss of generality, that every event is part of a grain; standalone events are grains of size one. Like Section 5, the causal concurrency question between events e_1 and e_2 is posed as the causal concurrency between the grains $g^{\diamond_1}$ and $g^{\diamond_2}$ containing these events.

It is clear that to design a constant space monitor, we cannot store the entire grain graph in the memory of the monitor. The idea is to forget all grains that are no longer active and *summarize* their effect instead. Under the assumption that all grains are minimal, the number of active grains are bounded by $|\mathcal{X}|$. This guarantees that the monitor keeps track of constantly many grains. We can annotate events with a finite set of grain identifiers $\{1, 2, \ldots, |\mathcal{X}|\}$. Identifiers are reused for grains that do not overlap.

The monitor maintains a graph where the nodes are precisely the set of active grains. We call this graph the *summarized grain graph*. Recall that the key information a monitor wants from a grain graph is whether two grains are connected by a directed path in the grain graph. Paths from the grain graph are represented as edges in the summarized grain graph. In particular, the edges in the summarized grain graph capture paths in the original grain graph whose intermediate grains have become inactive.

More formally, let π be a prefix of run w and let G be a valid set of grains. For grains (active or otherwise) g, g' that overlap with π, use the notation $g \rightsquigarrow_{\pi,G} g'$ to say that there is a path *through* inactive grains from g to g', i.e., there are grains $g_1, g_2 \ldots, g_k \in G$ (with $k > 1$, $g = g_1$ and $g' = g_k$), such that $g_2, \ldots, g_{k-1} \subseteq \pi$ are inactive (i.e., have started and completed) in π, and (g_i, g_{i+1}) is an edge of the original grain graph, for each $1 \leq i \leq k-1$. In essence, a path in the grain graph can be split into successive paths (through inactive grains) between the active grains.

The summarized grain graph is maintained by the monitor for answering a specific causal concurrency query between two grains $g^{\diamond_1}$ and $g^{\diamond_2}$. Formally:

Definition 7.2 (Summarized Grain Graph)**.** Let w be a run, G be a valid set of scattered grains and let π be a prefix of w. The summarized conflict graph of π is $\mathcal{SG}_{\pi,G} = (V_\pi, E_\pi)$, where

(1) $V_\pi = \mathsf{Active}_{\pi,G} \cup \{g^{\diamond_1}, g^{\diamond_2}\}$ is the set containing the active grains at the end of π as well as the focal grains,

(2) $(g, g') \in E_\pi$ if $g \rightsquigarrow_{\pi,G} g'$,

Summarized grain graphs sufficiently capture the reachability information between the focal grains:

Proposition 7.1. Let w be a run and let G be a valid set of scattered grains in w. There is a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{G}_{w,G}$ iff there is a prefix π of w such that there is a path from $g^{\diamond_1}$ to $g^{\diamond_2}$ in $\mathcal{SG}_{\pi,G}$.

It remains to argue that a constant-space streaming algorithm (that reads an input run in one pass from left to right) can successfully construct the summarized grain graph for the run. Intuitively, every time an active grain g is about to end, and as such become inactive and disappear, its summary is added to all its predecessors g'; that is, all nodes in the summarized grain graph that have a incoming edge to g. This way, a future active grain g'', that would be a successor of g in the grain graph, will now become the successor of all g''s, since the summary information stored in them will trigger the formation of an edge from them to g''. We make this idea formal in the detailed description of the monitors in the next sections.

For example, recall Fig. 6(c) and assume we want to query causal concurrency between g_1 and g_2. As such, both g_1 and g_2 have dedicated nodes in the summarized grain graph independent of their activeness status. The dashed edge appears in the summarized grain graph in place of the path from g_1 to g_4, while g_4 is active. Once g_4 is finished, as a predecessor of g_4, g_1 would remember the $\mathtt{w}(y)$ access so that it can add an edge to g_3 when its $\mathtt{r}(y)$ appears.

7.3 Monitoring for a Fixed set of Minimal Scattered Grains

The description of the monitor that checks grain graph concurrency (under a given set of minimal grains) is now straightforward as it essentially tracks and updates the summarized grain graph after each prefix of the run that is seen. We assume that the run contains symbols $\triangleright$ and $\triangleleft$ denoting start and end of grains. For ease of presentation, let us assume that the run labels the *focal grains* $g^{\diamond_1}$ and $g^{\diamond_2}$ with fresh identifiers identifiers $\diamond_1$ and $\diamond_2$. Thus, the set of grain identifiers is thus $\mathsf{gIDs} = \{1, 2, \ldots, |\mathcal{X}|\} \uplus \{\diamond_1, \diamond_2\}$.

The commutativity status of a grain depends on the *pending* variables of a grain, i.e., for each grain g, we identify the set of variables x such that x is read at some event $e \in g$ but not written to in g (i.e., $\mathsf{rf}_w(e) \notin g$), or x is written at some event $e \in g$ to but is read outside of g i.e., $\exists e', \mathsf{rf}_w(e') = e \wedge e' \notin g$). While this information can be guessed non-deterministically and checked later on, for simplifying the presentation of our monitor, we will also assume that the alphabet encodes this information as part of the $\triangleright$ marker of grain. Thus, the alphabet of runs can be assumed to be

$$\widehat{\Sigma} = \mathsf{gIDs} \times (\Sigma \uplus \{\triangleright\} \times \mathcal{P}(\mathcal{X}) \uplus \{\triangleleft\}).$$

For the purpose of this section, we will assume that the runs we consider are valid strings over the alphabet $\widehat{\Sigma}$ that are minimal, and use the language $L_{\mathsf{VMG}} = \{w \in \widehat{\Sigma}^* \mid w$ represents a **v**alid **m**inimal g*rain* annotation with focal grains$\}$ to denote the set of all such annotated runs. Below, we present the annotated version of the run in Fig. 3(a) with all grains g_1, g_2, g_3, g_4, assuming the two focal grains are g_2 and g_4 is presented below.

$$\begin{aligned} w \quad = \quad & (\triangleright, \varnothing)^{\diamond_1} \langle T_2, \mathtt{w}(x)\rangle^{\diamond_1} (\triangleright, \varnothing)^1 \langle T_1, \mathtt{w}(z)\rangle^1 \langle T_2, \mathtt{r}(z)\rangle^1 \triangleleft^1 \ (\triangleright, \varnothing)^1 \langle T_3, \mathtt{w}(z)\rangle^1 \langle T_3, \mathtt{r}(z)\rangle^1 \triangleleft^1 \\ & \langle T_3, \mathtt{r}(x)\rangle^{\diamond_1} \ \triangleleft^{\diamond_1} \ (\triangleright, \varnothing)^{\diamond_2} \langle T_1, \mathtt{w}(x)\rangle^{\diamond_2} \langle T_1, \mathtt{r}(x)\rangle^{\diamond_2} \triangleleft^{\diamond_2} \end{aligned}$$

In the above, we use the notation a^i as shorthand for $(i, a) \in \widehat{\Sigma}$. Observe that the (unique) grains with grain identifier $\diamond_1$ (namely grain g_2) overlaps with both the grains with identifier 1 (i.e., grains g_1 and g_3). Also observe that L_{VMG} is a regular language. Thus, a monitor that works correctly for runs in this language also works for non-annotated runs because the language of a constant space monitor is regular and thus closed under projection.

As discussed in Section 5, it suffices to consider the largest sound commutativity relation on a given set of grains. In fact, the largest commutativity relation also has a succinct representation — the dependence between any two scattered grains in this relation can be checked only using the *signatures* $g_1 = \langle E_1, V_1 \rangle$ and $g_2 = \langle E_2, V_2 \rangle$ of these grains:

$$\mathit{depend}(g_1, g_2) \iff \exists e_1 \in E_1, e_2 \in E_2 \cdot \begin{pmatrix} & \mathsf{thr}(e_1) = \mathsf{thr}(e_2) \\ \vee & \mathsf{var}(e_1) = \mathsf{var}(e_2) \notin V_1 \cap V_2 \wedge \mathtt{w} \in \{\mathsf{op}(e_1), \mathsf{op}(e_2)\} \end{pmatrix}$$

Recall that the signatures are bounded sized, and so is their dependence relationship. Thus, in order to maintain the summarized graph inductively, states stores additional information to correctly infer commutativity with other grains, including those that have finished.

Our monitor essentially tracks grain signatures of interest to infer edges between relevant grains. Let π be a prefix of w and let g be some grain that is active in π. Then, we use the notation $\mathsf{C}_\pi(g) = \{w[e] \mid e \in g \cap [1..|\pi|]\}$ to denote the *Contents* of G in π. Likewise, we denote the set of *Pending* variables of g by $\mathsf{P}(g) = \{x \in \mathcal{X} \mid \exists e \in g, \mathsf{var}(e) = x, \neg\mathsf{complete}(g, x)\}$. For each active grain g that we track in our monitor, we will also maintain summarized information about those grains that are no longer active but can be reached from g, to accurately infer edges in the summarized graph (alternatively *retroactive* paths in the grain graph). The first such information is the *Summarized Contents* of the dead but reachable grains: $\mathsf{SC}_\pi(g) = \bigcup\{\mathsf{C}_\pi(g') \mid \exists g' \subseteq \pi, g \rightsquigarrow_{\pi,G} g'\}$. Likewise, we use $\mathsf{SP}_\pi(g) = \bigcup\{\mathsf{P}_\pi(g') \mid \exists g' \subseteq \pi, g \rightsquigarrow_{\pi,G} g'\}$ to denote the *Summarized Pending* variables that g must keep track of.

We now formally describe our **G**rain **G**raph concurrency monitor $\mathcal{A}_{\mathsf{GG}} = (Q_{\mathsf{GG}}, q_0, \delta_{\mathsf{GG}}, F_{\mathsf{GG}})$. The states Q_{GG} of $\mathcal{A}_{\mathsf{GG}}$ are tuples of the form $\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$ where

- $V \subseteq \mathsf{gIDs}$ represents the set of active and focal grains of the summarized graph.
- $E \subseteq V \times V$ represents the edges of the summarized graph.
- $\mathsf{C} : V \to \mathcal{P}(\Sigma)$ maintains the *contents* of active grains. For each grain g tracked as a vertex u, we will have $\mathsf{C}(u) = \mathsf{C}_\pi(g)$ after having processed the prefix π.
- $\mathsf{P} : V \to \mathcal{P}(\mathcal{X})$ to track the set $\mathsf{P}_\pi(g)$ for each grain g (at the end of prefix π) tracked as some vertex in V.
- $\mathsf{SC} : V \to \mathcal{P}(\Sigma)$ is such that $\mathsf{SC}_\pi(u)$ tracks the set $\mathsf{SC}_\pi(g)$ at the end of prefix π, where u represents the grain g.
- $\mathsf{SP} : V \to \mathcal{P}(\mathcal{X})$ to track the set $\mathsf{SP}_\pi(g)$ for each grain g (at te end of prefix π) tracked as some vertex in V.

The start state of the monitor is $q_0 = \langle \emptyset, \emptyset, \lambda i \cdot \emptyset, \lambda i \cdot \emptyset, \lambda i \cdot \emptyset, \lambda i \cdot \emptyset\rangle$. All states $\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$ in which $\diamond_1, \diamond_2 \in V$ and further $\diamond_2$ is reachable from $\diamond_1$ via a path using edges in E are marked rejecting, and others are accepting (i.e., belong to F_{GG}). The transitions of the monitor are described in Fig. 7. When we see an event $e = (i, (\triangleright, Y))$ that demarcates the beginning of a grain with identifier i, we also know upfront the set of pending variables in the grain. At this point, we create

State	Event	State Update
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, (\triangleright, Y))$	$\langle V \uplus \{i\}, E, \mathsf{C}, \mathsf{P}[i \mapsto Y], \mathsf{SC}, \mathsf{SP}\rangle$
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, \triangleleft), i \in \{\diamond_1, \diamond_2\}$	$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, \triangleleft), i \notin \{\diamond_1, \diamond_2\}$	$\langle V', E', \mathsf{C}', \mathsf{P}', \mathsf{SC}', \mathsf{SP}'\rangle$, where, $V' = V - \{i\}, E' = mrg(E, i)$ $\mathsf{C}' = \mathsf{C}[i \mapsto \varnothing], \mathsf{P}' = \mathsf{P}[i \mapsto \varnothing]$, $\mathsf{SC}' = mrgSm(\mathsf{SC}, \mathsf{C}, E, i)$, $\mathsf{SP}' = mrgSm(\mathsf{SP}, \mathsf{P}, E, i)$
$\langle V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP}\rangle$	$e = (i, a), a \in \Sigma$	$\langle V, E', \mathsf{C} \cup \{a\}, \mathsf{SC}, \mathsf{SP}\rangle$, where, $E' = E \cup \{(j, i) \mid j \neq i$ and $Dep(\mathsf{C}(j) \cup \mathsf{SC}(j),\ a,\ \mathsf{P}(j) \cup \mathsf{SP}(j) \cup \mathsf{P}(i))\ \}$

$$mrg(E, i) = (E - \{(i, j), (j, i) \mid j \neq i\}) \cup \{(j, k) \mid (j, i) \in E, (i, k) \in E\}$$

$$mrgSm(SM, M, E, i) = \lambda j \cdot \begin{cases} \varnothing & \text{if } j = i \\ SM(j) \cup M(i) \cup SM(i) & \text{if } j \neq i, (j, i) \in E \\ SM(j) & \text{owise} \end{cases}$$

$$Dep(S, a, Z) \iff \exists b \in S \cdot \Big(\mathsf{thr}(a) = \mathsf{thr}(b) \vee \big(\mathsf{var}(a) = \mathsf{var}(b) \in Z \wedge \mathtt{w} \in \{\mathsf{op}(a), \mathsf{op}(b)\}\big)\Big)$$

Fig. 7. Grain Graph Concurrency Monitor for Annotated Runs $\mathcal{A}_{\mathrm{GG}}$: The monitor rejects if it is in a state $(V, E, \mathsf{C}, \mathsf{P}, \mathsf{SC}, \mathsf{SP})$ such that $(\diamond_1, \diamond_2) \in E^*$ at the end of the run, and accepts otherwise.

.

a new node labeled i in the graph and also track this set Y. When we see the end of a grain (marked $\triangleleft$) with identifier i, then we *garbage collect* the node i from the graph. As part of this garbage collection, we add an edge from the immediate predecessors of i to its immediate successors; this is captured by the operation $mrg(\cdot, \cdot)$. Further, the maps C and P reset the entry corresponding to i, and SC and SP entries of the predecessors of i are updated to include $\mathsf{C}(i)$, $\mathsf{SC}(i)$, $\mathsf{P}(i)$ and $\mathsf{SP}(i)$; this is captured using $mrgSm(\cdot, \cdot, \cdot, \cdot)$. When we see a read or a write event $a = \langle T, o, x\rangle$ in grain corresponding to the node i, we add edges from all nodes j to i such that j conflicts with a, i.e., either the contents or the summary of j contains a letter that conflicts with a, or one of j or a dead grain reachable from j has x as a pending variable.

One can prove that the monitor in Fig. 7 correctly maintains $\rightsquigarrow_{\pi,G}$ between each pair of active/focal grains after every prefix π of w, and as such, it can correctly decide whether the two focal grains are scattered grain concurrent or not:

Theorem 7.1. Given a run $w \in L_{\mathsf{VMG}}$ annotated with grains G, w is accepted by $\mathcal{A}_{\mathrm{GG}}$ iff the two focal grains are grain graph concurrent under G.

7.4 Monitoring Scattered Grain Concurrency

Similar to the case for grain concurrency, the monitor for scattered grain concurrency (Definition 6.4) can non-deterministically guess the choice of scattered grains and check, using the monitor in Fig. 7, if any of these guesses is valid and declares the two focal grains to causally concurrent.

Theorem 7.2. There exists a monitor $\mathcal{A}$ that uses $2^{O(|X| \cdot |\Sigma|)}$ space and accepts the word $w \in \Sigma^* \diamond_1 \Sigma^+ \diamond_2 \Sigma^+$ iff events that appear immediately after $\diamond_1$ and $\diamond_2$ are scattered grain concurrent in w. Consequently, scattered grain concurrency can be checked in constant space.

The proof of the above theorem relies on Theorem 7.1 and the observations that given a run $w \in \Sigma$ (with $\diamond_1$ and $\diamond_2$), we can guess the grains on w in constant space, validate whether the guessed grains are minimal and valid, and finally if the resulting annotated run is accepted by $\mathcal{A}_{\text{GG}}$. The number of states in $\mathcal{A}_{\text{GG}}$ is $2^{|\mathcal{X}|^2} \cdot (2^{|\Sigma|})^{|\mathcal{X}|} \cdot (2^{|\mathcal{X}|})^{|\mathcal{X}|} \cdot (2^{|\Sigma|})^{|\mathcal{X}|} \in 2^{O(|\mathcal{X}| \cdot |\Sigma|)}$ can can be obtained by counting the different ways to construct graphs on $|\mathcal{X}|$ vertices and deciding on the contents, pending variables and summaries and summarized pending variables for these vertices. Since the scattered grain concurrency monitors essentially adds non-determinism on top of this DFA, its deterministic version has exponentially many states, and each state thus has size $2^{O(|\mathcal{X}| \cdot |\Sigma|)}$.

8 RELATED WORK

There is little work in the literature in the general area of generalizing commutativity-based analysis of concurrent programs. On the theoretical side, some generalizations of Mazurkiewicz traces have been studied before [Bauget and Gastin 1995; Kleijn et al. 1998; Maarand and Uustalu 2019; Sassone et al. 1993]. The focus has mostly been on incorporating the concept of *contextual* commutativity, that is, when two events can be considered commutative in some contexts but not in all. This is motivated, among other things, by send/receive or producer/consumer models in distributed systems where send and receive actions commute in contexts with non-empty message buffers.

On the practical side, similar ideas were used in [Desai et al. 2014; Farzan et al. 2022; Farzan and Vandikas 2020; Genest et al. 2007] to reason about equivalence classes of concurrent and distributed programs under contextual commutativity. There is a close connection between this notion of contextual commutativity and the concept of *conditional independence* in partial order reduction [Godefroid and Pirottin 1993; Katz and Peled 1992] which is used as a weakening of the independence (commutativity) relation by making it parametric on the current *state* to increase the potential for reduction.

This work points out that, in general, the coarsest notion of equivalence may not yield monitoring-style algorithms for analyzing runs of concurrent programs, and puts forward an alternative notion of equivalence, coarser than trace equivalence, that can be efficiently used in monitoring causal concurrency. Below, we briefly survey some application domains relevant to programming languages research where our proposed equivalences can have an immediate positive impact.

Concurrency Bug Prediction. Dynamic analysis techniques for detecting concurrency bugs such as data races [Flanagan and Freund 2009], deadlocks [Samak and Ramanathan 2014] and atomicity violations [Farzan and Madhusudan 2008; Flanagan et al. 2008; Sorrentino et al. 2010] suffer from poor coverage since their bug detection capability is determined by the precise thread scheduling observed during testing. Predictive techniques [Said et al. 2011; Sen et al. 2005; Wang et al. 2009] such as those for detecting data races [Flanagan and Freund 2009; Huang et al. 2014; Kini et al. 2017, 2018; Mathur et al. 2018, 2021; Pavlogiannis 2019; Roemer et al. 2018; Smaragdakis et al. 2012; Sulzmann and Stadtmüller 2020], deadlocks [Kalhauge and Palsberg 2018; Tunç et al. 2023] or violations of high level properties [Ang and Mathur 2024; Farzan et al. 2012; Huang 2018; Huang et al. 2015] enhance coverage by exploring equivalent runs that might yield a bug. The core algorithmic problems involved in such approaches are akin to checking causal concurrency and causal orderedness. Coarser yet tractable equivalence relations can yield better analysis techniques with more accurate predictions. Recent work [Kulkarni et al. 2021] explores hardness results for data race prediction, including Mazurkiewicz-style reasoning, when the alphabet is not assumed to be of constant size.

Dynamic Partial Order Reduction for Stateless Model Checking. There has been a rising interest in developing dynamic partial order based [Flanagan and Godefroid 2005] stateless model checking techniques [Godefroid 1997] that are *optimal*, in that they explore as few program runs as possible from the underlying program. Coming up with increasingly coarser equivalences is the go-to approach for this. The notion of reads-from equivalence we study has received a lot of attention in this context [Abdulla et al. 2019; Chalupa et al. 2017; Kokologiannakis et al. 2022, 2019]. Recent works also consider an even coarser reads-value-from [Agarwal et al. 2021; Chatterjee et al. 2019] equivalence. Events encode variable values and two runs are equivalent if every read observes the same value across the two runs. The problem of verifying sequential consistency, which is one of the key algorithmic questions underlying such approaches, was proved to be intractable in general by Gibbons and Korach [Gibbons and Korach 1997].

9 CONCLUSION AND FUTURE WORK

This paper demonstrates that reads-from equivalence, the most relaxed sound notion of equivalence on concurrent program runs, does not share the nice algorithmic properties of (Mazurkiewicz) trace equivalence. This poses the following research questions: "Are there other notions of equivalence, which remain sound, relax trace equivalence, and yet maintain its key desirable algorithmic properties? And, what design principles bring about algorithmic simplicity?". We propose two new notions of equivalence in this paper under which causal concurrency can be decided by a streaming algorithm in constant space. Equivalence based on contiguous grains shares the characteristic with trace equivalence, in that it is definable purely in terms of commutativity. Remarkably, while this characteristic is lost with scattered grains, the algorithmic simplicity remains. The point of commonality between the two notions is that each individual event can only *move* in one role: either individually, or as part of a grain. As we demonstrate in Theorem 4.1, algorithmic hardness kicks in when this rule is broken in the simplest of syntactic settings. It would be interesting to investigate whether one can further relax the notion of *grain equivalence* by breaking this barrier.

As we note in Section 7, the straightforward determinization of the (scattered) grain concurrency monitor can result in an exponential blowup on the number of threads and shared variables. We treat these parameters as constants, in a manner similar to trace theory's reliance on a finite (constant-sized) alphabet of actions. There seems to be a tradeoff between how coarse the equivalence relation is and how efficiently causal concurrency can be monitored. The research question of how to devise a practical monitor when the number of threads or variables is not very small remains an interesting direction for future research.

Finally, this paper studies coarser equivalences in the context of a *causal concurrency query*. In several application domains, the standard oracle for causal concurrency based on trace equivalence can be replaced with the (scattered) grain concurrency from this paper. Yet, there remain other application domains in which trace equivalence is used in completely different ways: for example, proof simplification [Farzan 2023] by verifying a commutativity-based reduction of a concurrent program. The notion of soundness used in this paper suffices when the object of study is a single program run. In proof simplification, however, a set of program runs must be considered together, and, as we argued, different grain commutativity relations may be sound in different program runs. Moreover, the alphabet of actions for proof simplification typically includes atomic program statements rather than shared variable reads and writes. As such, the theoretical results presented in this paper do not immediately offer a solution in such domains. It will be interesting to explore how similar coarse equivalences, based on commutativity of words (rather than symbols), can be designed to be exploited for proof simplification.

ACKNOWLEDGMENTS

The initial ideas of this work stemmed from discussions when the authors co-attended a research program titled Theoretical Foundations of Computer Systems (TFCS) organized by the Simons Institute for the Theory of Computing. Azadeh Farzan's research is supported by a Discovery Grant from the National Science and Engineering Research Council of Canada (NSERC). Umang Mathur was partially supported by a Singapore Ministry of Education (MoE) Academic Research Fund (AcRF) Tier 1 grant.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 150 (oct 2019), 29 pages. https://doi.org/10.1145/3360576

Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. 2021. Stateless Model Checking Under a Reads-Value-From Equivalence. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 341–366. https://doi.org/10.1007/978-3-030-81685-8_16

Zhendong Ang and Umang Mathur. 2024. Predictive Monitoring against Pattern Regular Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 73 (jan 2024). https://doi.org/10.1145/3632915

Serge Bauget and Paul Gastin. 1995. On congruences and partial orders. In *Mathematical Foundations of Computer Science 1995*, Jiří Wiedermann and Petr Hájek (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 434–443.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (dec 2017), 30 pages. https://doi.org/10.1145/3158119

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 124:1–124:29. https://doi.org/10.1145/3360550

Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 709–725.

Volker Diekert and Grzegorz Rozenberg (Eds.). 1995. *The Book of Traces*. World Scientific.

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

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

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

Azadeh Farzan and P. Madhusudan. 2006. Causal Atomicity. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–328.

Azadeh Farzan and P. Madhusudan. 2008. Monitoring Atomicity in Concurrent Programs. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–65.

Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting Null-pointer Dereferences in Concurrent Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. ACM, New York, NY, USA, Article 47, 11 pages. https://doi.org/10.1145/2393596.2393651

Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. 2009. Meta-analysis for Atomicity Violations Under Nested Locking. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 248–262. https://doi.org/10.1007/978-3-642-02658-4_21

Azadeh Farzan and Umang Mathur. 2023. Coarser Equivalences for Causal Concurrency. *CoRR* abs/2208.12117 (2023). https://doi.org/10.48550/ARXIV.2208.12117 arXiv:2208.12117

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

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

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 293–303. https://doi.org/10.1145/1375581.1375618

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

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

Phillip B. Gibbons and Ephraim Korach. 1994. On Testing Cache-Coherent Shared Memories. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures* (Cape May, New Jersey, USA) *(SPAA '94)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/181014.181328

Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614 arXiv:https://doi.org/10.1137/S0097539794279614

Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

Patrice Godefroid and Didier Pirottin. 1993. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. 438–449.

Michel Hack. 1976. *Petri net language.* Massachusetts Institute of Technology.

Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic Predictive Concurrency Analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 847–857.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Russell Impagliazzo and Ramamohan Paturi. 2001. On the complexity of k-SAT. *J. Comput. System Sci.* 62, 2 (2001), 367–375.

Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. 1999. Toward Integration of Data Race Detection in DSM Systems. *J. Parallel Distrib. Comput.* 59, 2 (Nov. 1999), 180–203. https://doi.org/10.1006/jpdc.1999.1574

Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

Shmuel Katz and Doron A. Peled. 1992. Defining Conditional Independence Using Collapses. *Theor. Comput. Sci.* 101, 2 (1992), 337–359.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2018. Data Race Detection on Compressed Traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3236024.3236025

H.C.M. Kleijn, R. Morin, and B. Rozoy. 1998. Event Structures for Local Traces. *Electronic Notes in Theoretical Computer Science* 16, 2 (1998), 98–113. https://doi.org/10.1016/S1571-0661(04)00120-3 EXPRESS '98, Fifth International Workshop on Expressiveness in Concurrency (Satellite Workshop of CONCUR '98).

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly Stateless, Optimal Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 6, POPL, Article 49 (jan 2022), 28 pages. https://doi.org/10.1145/3498711

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *32nd International Conference on Concurrency Theory (CONCUR 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:23. https://doi.org/10.4230/LIPIcs.CONCUR.2021.16

Hendrik Maarand and Tarmo Uustalu. 2019. Certified Normalization of Generalized Traces. *Innov. Syst. Softw. Eng.* 15, 3–4 (sep 2019), 253–265. https://doi.org/10.1007/s11334-019-00347-1

Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 145 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276515

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434317

Umang Mathur and Mahesh Viswanathan. 2020. Atomicity Checking in Linear Time Using Vector Clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 183–199. https://doi.org/10.1145/3373376.3378475

A Mazurkiewicz. 1987. 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., 279–324.

Robin Milner. 1980. *A calculus of communicating systems*. Springer.

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

Grigore Roşu and Mahesh Viswanathan. 2003. Testing Extended Regular Language Membership Incrementally by Rewriting. In *Rewriting Techniques and Applications*, Robert Nieuwenhuis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 499–514.

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 313–327. http://dl.acm.org/citation.cfm?id=1986308.1986334

Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2555243.2555262

Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. 1993. Deterministic behavioural models for concurrency. In *Mathematical Foundations of Computer Science 1993*, Andrzej M. Borzyszkowski and Stefan Sokołowski (Eds.).

Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Formal Methods for Open Object-Based Distributed Systems*, Martin Steffen and Gianluigi Zavattaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–226.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

Martin Sulzmann and Kai Stadtmüller. 2020. Efficient, near Complete, and Often Sound Hybrid Dynamic Data Race Prediction. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes* (Virtual, UK) *(MPLR 2020)*. Association for Computing Machinery, New York, NY, USA, 30–51. https://doi.org/10.1145/3426182.3426185

Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1733–1758. https://doi.org/10.1145/3591291

Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (jun 2023), 26 pages. https://doi.org/10.1145/3591291

Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *Proceedings of the 2Nd World Congress on Formal Methods* (Eindhoven, The Netherlands) *(FM '09)*. Springer-Verlag, Berlin, Heidelberg, 256–272. https://doi.org/10.1007/978-3-642-05089-3_17

Ryan Williams. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science* 348, 2-3 (2005), 357–365.

Glynn Winskel. 1987. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–392.

Received 2023-07-11; accepted 2023-11-07