Predictive Monitoring against Pattern Regular Languages

ZHENDONG ANG, National University of Singapore, Singapore
UMANG MATHUR, National University of Singapore, Singapore

While current bug detection techniques for concurrent software focus on unearthing low-level issues such as data races or deadlocks, they often fall short of discovering more intricate temporal behaviours that can arise even in the absence of such low-level issues. In this paper, we focus on the problem of dynamically analysing concurrent software against high-level temporal specifications such as LTL. Existing techniques for runtime monitoring against such specifications are primarily designed for sequential software and remain inadequate in the presence of concurrency — violations may be observed only in intricate thread interleavings, requiring many re-runs of the underlying software in conjunction with the analysis. Towards this, we study the problem of *predictive runtime monitoring*, inspired by the analogous problem of *predictive data race detection* studied extensively recently. The predictive runtime monitoring question asks, given an execution σ, if it can be soundly *reordered* to expose violations of a specification. In general, this problem may become easily intractable when either the specifications or the notion of reorderings used is complex.

In this paper, we focus on specifications that are given in regular languages. Our notion of reorderings is *trace equivalence*, where an execution is considered a reordering of another if it can be obtained from the latter by successively commuting adjacent independent actions. We first show that, even in this simplistic setting, the problem of predictive monitoring admits a super-linear lower bound of $O(n^\alpha)$, where n is the number of events in the execution, and α is a parameter describing the degree of commutativity, and typically corresponds to the number of threads in the execution. As a result, predictive runtime monitoring even in this setting is unlikely to be efficiently solvable, unlike in the non-predictive setting where the problem can be checked using a deterministic finite automaton (and thus, a constant-space streaming linear-time algorithm).

Towards this, we identify a sub-class of regular languages, called *pattern languages* (and their extension *generalized pattern languages*). Pattern languages can naturally express specific ordering of some number of (labelled) events, and have been inspired by popular empirical hypotheses underlying many concurrency bug detection approaches such as the “small bug depth” hypothesis. More importantly, we show that for pattern (and generalized pattern) languages, the predictive monitoring problem can be solved using a constant-space streaming linear-time algorithm. We implement and evaluate our algorithm PatternTrack on benchmarks from the literature and show that it is effective in monitoring large-scale applications.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → *Theory and algorithms for application domains*; *Program analysis*.

Additional Key Words and Phrases: concurrency, dynamic analysis, predictive monitoring, complexity

ACM Reference Format:
Zhendong Ang and Umang Mathur. 2024. Predictive Monitoring against Pattern Regular Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 73 (January 2024), 35 pages. https://doi.org/10.1145/3632915

1 INTRODUCTION

Writing reliable concurrent programs remains a challenge to date. Subtle bugs, arising due to intricate choreography of threads, often evade rigorous testing but appear in deployment under

Authors' addresses: Zhendong Ang, National University of Singapore, Singapore, Singapore, zhendong.ang@u.nus.edu; Umang Mathur, National University of Singapore, Singapore, Singapore, umathur@comp.nus.edu.sg.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/1-ART73
https://doi.org/10.1145/3632915

intense workloads. The first line of defence against such bugs is tools that enable developers to find concurrency errors automatically. Towards this, both static and dynamic analysis approaches have been proposed. Static approaches [Blackshear et al. 2018; Engler and Ashcraft 2003; Naik et al. 2006] are typically geared towards certifying program correctness and can overwhelm developers with excessive false positive reports [Sadowski et al. 2018]. Dynamic analysis techniques, on the other hand, remain the preferred approach for the task of automated bug detection. Here one executes the underlying program, observes its behaviour, and infers the presence or absence of bugs based on this observed behaviour. Even though such techniques are implicitly incomplete, they remain popular in practice, thanks to the inherent desirable properties such as low overhead and soundness of bug reports. Unsurprisingly, such techniques enjoy more widespread adoption [Serebryany and Iskhodzhanov 2009].

Traditional dynamic analysis approaches for detecting concurrency bugs often cater to implicit specifications such as data race freedom, deadlock freedom or atomicity of code blocks. However, such techniques do not attempt to expose undesirable behaviours due to faulty order of interactions of threads, that are nevertheless symptomatic of serious failures. In practice, developers often rely on specific properties such as class invariants or temporal behaviours such as "file `f` cannot be closed in between the creation of `f` and access to `f`" to reason about the overall safety of their software. Nevertheless, the validity of such invariants often relies on complex synchronizations to be put in place, a task that even the most expert developers struggle to get right. Tools such as data race detectors are not helpful either; they fail to expose such high-level problems.

Consider, for example, the simplistic Java program P in Figure 1a. This is a code snippet inspired from the GitHub project `antlrworks` [ant 2023]. The class **`DBPlayer`** contains two fields, `count` and `inputs`, and two methods, `play` and `reset`, both of which write to both fields. In order to keep `count` and `inputs` in a consistent state, executions of method `play` and `reset` should behave atomically. Observe that this is a high-level data race, instead of a classical data race property [Artho et al. 2003], which are prevented because of the use of thread-safe data structure and atomic integer. This type of violation is also studied in [Vaziri et al. 2006].

The problematic behaviour in the above example can, in fact, be expressed, in a temporal specification formalism like LTL, or as a regular language L_{fail}, that expresses the order in which the events $\texttt{add}_{\mathsf{Call}}$, $\texttt{clear}_{\mathsf{Call}}$, on the `inputs` field, and the two `set` operations on the `count` field are made. Such specification languages have been extensively studied in the context of runtime verification and efficient algorithms and tools have been developed to monitor such specifications. Such techniques, however, are not well-suited for monitoring concurrent programs. Thanks to non-determinism due to thread scheduling, even the most well-engineered LTL-monitoring technique may require many re-executions of the program under test to eventually observe a violation of a desirable property. Consider again the program P be from Figure 1a, and the execution σ_{safe} generated when monitoring P (Figure 1b). As such, $\sigma_{\mathsf{safe}} \notin L_{\mathsf{fail}}$ fails to expose the violation encoded as L_{fail}, even though the very similar execution σ_{fail} (Figure 1c) of the same program P can very well expose the problematic behaviour. This highlights the lack of robustness in traditional monitoring approaches that simply check whether an execution observed during dynamic analysis witnesses a specification.

To tackle this challenge, we borrow wisdom from *predictive* approaches [Huang et al. 2015, 2014; Kalhauge and Palsberg 2018; Kini et al. 2017; Mathur et al. 2021; Pavlogiannis 2019; Said et al. 2011; Smaragdakis et al. 2012], that are capable of exposing low-level bugs such as data races even starting from executions that do not explicitly observe the concurrency bug under question. Towards this, we consider the analogous *predictive runtime monitoring* question — given a specification, encoded as a language L and given an execution σ of some concurrent program P, can σ be *reordered*, in a sound manner, to execution σ' so that $\sigma' \in L$? Observe that an efficient

```java
public class DBPlayer {
  AtomicInteger count = new AtomicInteger(0);
  ConcurrentHashMap.KeySetView<Integer, Boolean> inputs = ConcurrentHashMap.newKeySet();

  public void play(int event, int lastPos) {
    inputs.add(event);
    count.set(lastPos);
  }

  public void reset() {
    inputs.clear();
    count.set(0);
  }

  public static void main(String[] args)
    throws InterruptedException{
    final DBPlayer player = new DBPlayer();

    Thread t1 = new Thread(new Runnable() {
      public void run() {
        player.reset();
    }});

    Thread t2 = new Thread(new Runnable() {
      public void run() {
        player.play(365, 5);
    }});

    t1.start();
    t2.start();
  }
}
```

(a) Example Java program P

$$
\begin{aligned}
e_1 &: [t_{\text{main}}, \texttt{fork}(t_1)] \\
e_2 &: [t_{\text{main}}, \texttt{fork}(t_2)] \\
e_3 &: [t_1, \texttt{reset}_{\text{Call}}] \\
e_4 &: \underline{[t_1, \texttt{clear}_{\text{Call}}(\texttt{inputs})]} \\
e_5 &: [t_1, \texttt{write}(\texttt{inputs})] \\
e_6 &: [t_1, \texttt{clear}_{\text{Return}}] \\
e_7 &: \underline{[t_1, \texttt{set}(\texttt{count})]} \\
e_8 &: [t_1, \texttt{reset}_{\text{Return}}] \\
e_9 &: [t_2, \texttt{play}_{\text{Call}}] \\
e_{10} &: \underline{[t_2, \texttt{add}_{\text{Call}}(\texttt{inputs})]} \\
e_{11} &: [t_2, \texttt{write}(\texttt{inputs})] \\
e_{12} &: [t_2, \texttt{add}_{\text{Return}}] \\
e_{13} &: \underline{[t_2, \texttt{set}(\texttt{count})]} \\
e_{14} &: [t_2, \texttt{play}_{\text{Return}}]
\end{aligned}
$$

(b) Execution σ_{safe} of P

$$
\begin{aligned}
e_1 &: [t_{\text{main}}, \texttt{fork}(t_1)] \\
e_2 &: [t_{\text{main}}, \texttt{fork}(t_2)] \\
e_9 &: [t_2, \texttt{play}_{\text{Call}}] \\
e_{10} &: \underline{[t_2, \texttt{add}_{\text{Call}}(\texttt{inputs})]} \\
e_3 &: [t_1, \texttt{reset}_{\text{Call}}] \\
e_4 &: \underline{[t_1, \texttt{clear}_{\text{Call}}(\texttt{inputs})]} \\
e_5 &: [t_1, \texttt{write}(\texttt{inputs})] \\
e_6 &: [t_1, \texttt{clear}_{\text{Return}}] \\
e_7 &: \underline{[t_1, \texttt{set}(\texttt{count})]} \\
e_8 &: [t_1, \texttt{reset}_{\text{Return}}] \\
e_{11} &: [t_2, \texttt{write}(\texttt{inputs})] \\
e_{12} &: [t_2, \texttt{add}_{\text{Return}}] \\
e_{13} &: \underline{[t_2, \texttt{set}(\texttt{count})]} \\
e_{14} &: [t_2, \texttt{play}_{\text{Return}}]
\end{aligned}
$$

(c) Execution σ_{fail} of P

Fig. 1. Example Java program P and its executions. Interleaving `play()` and `reset()` results in the values of `count` and `inputs` to be in an inconsistent state.

solution to the predictive runtime monitoring problem has the potential to enhance coverage of traditional runtime verification approaches that only focus on the observed execution, when used for concurrent software.

Even in the context of the simplest specification — data races — the predictive question is intractable, in general [Mathur et al. 2020]. Nevertheless, tractable and linear-time algorithms for predictive data race detection have been proposed recently [Kini et al. 2017; Mathur et al. 2021]. GPredict [Huang et al. 2015] also uses predictive analysis to detect high-level concurrency properties. However, it is SMT-based and not scalable in practice. In this work, we aim to develop efficient algorithms for predictive monitoring against richer specifications. In the context of data races, the key insight underlying efficient algorithms is to restrict the search space of reorderings, and is also central to our work presented here.

We consider reorderings described by *Mazurkiewicz trace equivalence* [Mazurkiewicz 1987]. Here, one fixes an *independence relation*, consisting of pairs of actions that can be commuted when present next to each other in any context. With such a commutativity specification, two executions are deemed equivalent if they can be reached from each other by successive commutations of independent actions. Indeed, the most popular approaches for race detection — those based on the *happens-before* partial order [Flanagan and Freund 2009; Itzkovitz et al. 1999] — essentially employ this reasoning principle and, as a result, can be implemented using a fast linear-time algorithm.

In this paper, we study the problem of predictive trace monitoring against a regular language L — given an execution σ, is σ Mazurkiewicz trace equivalent to an execution that belongs to L? The problem has previously been studied in a theoretical setting [Bertoni et al. 1989], where the authors proposed an algorithm that runs in time $O(n^\alpha)$ for executions with n events; α is a parameter given by the independence relation, and is typically equal to the number of threads in the execution. Such an algorithm is unlikely to be practical for large-scale software applications where executions typically contain millions of events. Unfortunately, as we show in our paper, in general, this problem cannot be solved using a faster algorithm — we show a matching conditional lower bound of $\Omega(n^\alpha)$ using techniques from fine-grained complexity [Williams 2018].

To this end, we identify a class of specifications that we call *pattern* regular languages for which predictive monitoring can be performed efficiently. This class of pattern languages has been inspired by systematic concurrency bug detection approaches that often rely on empirical hypotheses such as the *small bug depth hypothesis* — "*Empirically, though, many bugs in programs depend on the precise ordering of a small number of events . . . there is some constant d and a subset . . . of events such that some ordering of these d events already exposes the bug no matter how all other events are ordered.*" [Chistikov et al. 2016]. Pattern languages are the natural class of specifications under such a hypothesis — a language L is a pattern language if it is of the form $\texttt{Patt}_{a_1,\ldots,a_d} = \Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_d \Sigma^*$, where Σ is the alphabet labelling events, and $a_1, \ldots, a_d \in \Sigma$ are d symbols (not necessarily distinct from each other). We also propose an extension, the class of *generalized* pattern languages which are unions of pattern languages. We expect that this class of specifications, previously deployed in domains such as pattern mining [Agrawal and Srikant 1995] or root cause analysis [Murali et al. 2021], will be useful in advancing the state of the art in concurrency testing, given the success of past approaches based on similar empirical hypotheses including context bounding [Burckhardt et al. 2010; Emmi et al. 2011; Musuvathi and Qadeer 2006], small bug depth [Burckhardt et al. 2010; Chistikov et al. 2016], delay bounding [Emmi et al. 2011], coarse interleaving hypothesis [Kasikci et al. 2017] for testing Java-like multi-threaded software programs, event-driven and asynchronous programs, distributed systems, and more recently for testing and model checking of weak memory behaviours [Abdulla et al. 2019; Gao et al. 2023].

The main result of this work is that the class of (generalized) pattern languages admits a constant-space linear-time algorithm for the predictive monitoring problem under trace equivalence. This algorithm relies on several interesting insights specific to pattern languages that enable our efficient monitoring algorithm, and is of independent interest.

We show that, in fact, we can use *vector clocks* to further develop a fast and scalable predictive monitoring algorithm PatternTrack. We implement PatternTrack in Java and evaluate its performance over a set of benchmark Java applications derived from prior works. Our comparison of PatternTrack against the classical algorithm due to [Bertoni et al. 1989] reveals the effectiveness of both a specialized class of specifications and a linear-time algorithm for predictive monitoring.

The rest of the paper is organized as follows. In Section 2, we discuss background on modelling events, executions, and questions in runtime monitoring. In Section 3, we recall trace equivalence and present our hardness result for predictive trace monitoring against regular languages (Theorem 3.2). In Section 4, we formally define the class of pattern and generalized pattern languages, and provide language-theoretic characterizations of these languages. In Section 5, we present our main result — a constant-space streaming algorithm for predictive trace monitoring against generalized pattern regular languages, and a more practical algorithm PatternTrack that uses vector clocks (Section 5) for the same task. We present our experimental evaluation in Section 6, discuss related work in Section 7 and conclude in Section 8.

2 PRELIMINARIES

In this section, we present relevant background and notations useful for the rest of the paper.

Events and Executions. Our primary subject of study is concurrent programs and their executions with a focus on developing monitoring algorithms for them. In this regard, executions can be modelled as sequences of events. For this, we will consider an *event alphabet*, or simply, alphabet Σ, which will intuitively represent a set of relevant events observed during program execution. As an example, Σ can include information about instructions corresponding to method invocations/returns, or even loads or stores of memory locations. For the purpose of modelling concurrent programs, the symbols in our alphabet Σ also contain information about thread identifiers, belonging to a fixed set $\mathcal{T}$. An event is a unique occurrence of a symbol from Σ. Formally, an event is a tuple $e = \langle id, a \rangle$, where id is the unique identifier of e, while $a \in \Sigma$ is the label of e; we will denote the identifiers and labels of e using the notation $\mathsf{lab}(e) = a$ and $\mathsf{id}(e) = id$. In the context of concurrent executions, the labels of events will be of the form $a = [t, op] \in \Sigma$ to denote information about the thread $t \in \mathcal{T}$ that performs the corresponding event and the operation op performed in that event. We will often omit the identifier id of events when it is clear from the context. An execution can then be modelled as a sequence of events with labels from Σ. We use Events_σ to denote the set of events occurring in an execution σ. We will use $<^\sigma$ to denote the total order induced by σ, i.e., $e_1 <^\sigma e_2$ iff the event e_1 appears before e_2 in the sequence σ. The word constructed by concatenating the labels of σ will be denoted by $\mathsf{lab}(\sigma) \in \Sigma^*$, and will often confuse σ with $\mathsf{lab}(\sigma)$ when the identifiers of events are not important. We will use $|\sigma|$ to denote the number of events in σ.

Runtime Monitoring. Runtime verification has emerged as a powerful technique for enhancing reliability of software and hardware systems by augmenting traditional software testing methods with more expressive specifications that can be checked or *monitored* during the execution. One of the primary components in a traditional runtime monitoring workflow is the choice of specifications to be monitored at runtime. In general, many different classes of specifications have been considered in the literature and practice, including temporal logics like LTL [Pnueli 1977] and its variants [Giacomo and Vardi 2013] with quantitative [Maler and Nickovic 2004] and timing aspects [Koymans 1990]. More expressive variants such as extensions with calls and returns [Alur et al. 2004], extended regular expressions or vanilla regular languages have also been considered. A large class of these specifications can be described by a language (or a set of executions) over a chosen alphabet Σ of interest. Examples include LTL, FOLTL, extended-regular expressions [Rosu and Viswanathan 2003], etc, and the more general class of regular languages. For a language L over the alphabet Σ, the runtime monitoring problem then asks — given an execution $\sigma \in \Sigma^*$ coming from a program P, does σ satisfy the specification L, i.e., $\sigma \in L$? When L represents an undesired (resp. desired) behaviour, a positive (resp. negative) answer to the membership question $\sigma \in L$ can then be flagged to developers or system engineers who can fix the erroneous behaviour (or in some cases, refine the specification). In this paper, we study the monitoring problem in the context of concurrent programs, with a focus on developing more robust solutions to the monitoring problem. Next, we motivate the *predictive* monitoring problem and formally describe it subsequently.

Example 2.1. Consider again the Java program P, shown in Figure 1a, that implements class `DBPlayer` with fields, `count` and `inputs`, and member methods, `play` (for adding event indices to `inputs` and log position into `count`) and `reset` (for clearing `inputs` and set `count` as 0). The function `main` creates a `DBPlayer` object, and then calls `play` and `reset` concurrently in different threads. Suppose, we are interested in observing the behaviours of the program that lead to the inconsistent state of fields `count` and `inputs`. The events thus generated belong to the alphabet $\Sigma_{\mathsf{Ex}} = \{[t, op] \mid t \in \mathcal{T}, op \in \mathcal{O}\}$, where $\mathcal{T} = \{t_{\mathsf{main}}, t_1, t_2\}$ is the set of threads corresponding to the main function and two forked threads, while $\mathcal{O} = \{\texttt{fork}(t) \mid t \in \mathcal{T}\} \cup \{\texttt{write(inputs)}\} \cup \{\texttt{add}_{\text{Call}}(\texttt{inputs}), \texttt{add}_{\text{Return}}, \texttt{clear}_{\text{Call}}(\texttt{inputs}), \texttt{clear}_{\text{Return}}, \texttt{set}(\text{count})\}$. Observe that an execution σ of this program might leave the fields in an inconsistent state if it belongs to the following language L_{fail} that constrains the order of calls to `add`, `clear`, and two `set`s to `count`:

$$\Sigma_{\text{Ex}}^{*} \cdot [t_2, \texttt{add}_{\text{Call}}(\texttt{inputs})] \cdot \Sigma_{\text{Ex}}^{*} \cdot [t_1, \texttt{clear}_{\text{Call}}(\texttt{inputs})] \cdot \Sigma_{\text{Ex}}^{*} \cdot [t_1, \texttt{set}(\text{count})] \cdot \Sigma_{\text{Ex}}^{*} \cdot [t_2, \texttt{set}(\text{count})] \cdot \Sigma_{\text{Ex}}^{*}$$

In Figure 1b, we show an execution σ_{safe} of P, which is a sequence of 14 events $e_1, \ldots, e_{14}$, labelled with Σ_{Ex}. Observe that $\sigma_{\text{safe}} \notin L_{\text{fail}}$ fails to witness the buggy behaviour of P, and would deem the task of runtime monitoring against L_{fail} unsuccessful. On the other hand, the bug would be exposed successfully if the execution σ_{fail} (Figure 1c) was observed when monitoring P. More importantly, observe that, σ_{fail} can, in fact, be obtained by *reordering* the events of σ_{safe}, hinting at the possibility of enhancing the effectiveness of runtime monitoring via *prediction*! Notice that an execution belonging to L_{fail} may not immediately leave the data in an inconsistent state. Indeed, σ_{fail} here is actually benign, but it hints towards another execution that orders e_{11} before e_5, which is invalid.

Example 2.1 illustrates the challenge of naively adapting the traditional runtime monitoring workflow for the case when the software under test exhibits concurrency — even though the underlying software may violate a specification, exposing the precise execution that witnesses such a violation is like finding a needle in a haystack. This is because, even under the same input, not all thread interleavings may witness a violation of a temporal specification of interest, and the precise interleaving that indeed witnesses the violation may require careful orchestration from the thread scheduler. As a result, the bug may remain unexposed even after several attempts to execute the program. The emerging class of *predictive analysis* techniques [Kini et al. 2017; Mathur et al. 2021; Pavlogiannis 2019; Smaragdakis et al. 2012] attempts to address this problem in the context of concurrency bugs such as data races and deadlocks. Here, one observes a single execution of a program, and *infers* additional executions (from the observed one) that are guaranteed to be generated by the same underlying program, and also witness a violation of a property of interest. Such techniques thus partially resolve the dependence on the often demonic non-determinism due to thread scheduler, and proactively increase coverage when testing concurrent programs.

Correct Reorderings. In order to formalize the predictive analysis framework, we require a notion of the set of *feasible* or *correct* reorderings of a given execution [Serbanuta et al. 2012]. For a given execution σ over Σ, the set $\text{CReorderings}(\sigma)$ of correct reorderings of σ represents the set of executions that will be generated by any program P that generates σ. The formal notion of correct reorderings can often be obtained by formally modelling the programming language, together with the semantics of concurrent objects [Herlihy and Wing 1990] such as threads, shared memory and synchronization objects such as locks [Serbanuta et al. 2012]. In the case of multi-threaded programs, it is customary to include symbols corresponding to reads and writes to all shared objects and lock acquisition and release events in the alphabet Σ. Prior works have developed several different (yet related) definitions to precisely capture this notion; the most prominent notion is due to [Smaragdakis et al. 2012], which we describe next.

Given a well-formed[1] execution $\sigma \in \Sigma^*$, we have that an execution $\rho \in \text{CReorderings}(\sigma)$ iff the following conditions hold — (1) The set of events in ρ is a subset of those in σ, (2) ρ is well-formed; this means, for example, critical sections on the same lock do not overlap in ρ. (3) ρ preserves the *program-order* of σ, i.e., for every thread t, the sequence of events $\rho|_t$ obtained by projecting ρ to events of thread t is a prefix of the sequence $\sigma|_t$ obtained by projecting σ to t. (4) ρ preserves

[1]An execution is well-formed if it can be generated by a program. This means it obeys the semantics of the underlying programming language. As an example, when executions contain events corresponding to acquire and release operations of locks, well-formedness asks that critical sections on the same lock do not overlap.

control flow of σ, i.e., every read event e in ρ observes the same write event e' as in σ. Observe that, for programming languages such as Java or C/C++, $\rho \in \mathsf{CReorderings}(\sigma)$ implies that indeed any control flow undertaken by σ can also be undertaken by ρ, and thus, any program that generates σ can also generate ρ, possibly under a different thread interleaving. In general, the more permissive the set $\mathsf{CReorderings}(\sigma)$, the higher the computational complexity involved in reasoning with it. For example, for the precise definition due to [Smaragdakis et al. 2012], the race *prediction* question — given an execution σ, check if there is a $\rho \in \mathsf{CReorderings}(\sigma)$ that exhibits a data race — is an NP-hard problem [Mathur et al. 2020]. On the other hand, for simpler (and less permissive) notions such as trace equivalence, which we recall in Section 3, the corresponding race prediction question becomes linear-time (and also constant-space) checkable!

Predictive Monitoring. We are now ready to formalize the predictive monitoring framework, by generalizing related notions such as predictive data race detection [Kini et al. 2017; Kulkarni et al. 2021; Mathur et al. 2021; Said et al. 2011; Smaragdakis et al. 2012] or predictive deadlock detection [Tunç et al. 2023]. For a specification language L, the predictive monitoring question asks, given an execution $\sigma \in \Sigma^*$, is there an execution ρ such that $\rho \in \mathsf{CReorderings}(\sigma) \cap L$, i.e., ρ is a correct reordering of σ that also witnesses the specification L. In the context of Example 2.1, while the execution σ_{safe} (Figure 1b) is a negative instance of the monitoring problem against the language L_{fail}, it is indeed a positive instance of *predictive* monitoring problem against the same language, because the witnessing execution $\sigma_{\mathsf{fail}} \in L_{\mathsf{fail}}$ is a *correct reordering* of σ_{safe}. In fact, notions such as predictive data race detection can be easily formulated in this framework as follows. Let us fix the set of threads to be $\mathcal{T}$, and also a set of memory locations/objects $\mathcal{X}$. Consider the alphabet of read and write events: $\Sigma_{\mathsf{RW}} = \{[t, \mathsf{w}(x)], [t, \mathsf{r}(x)] \mid t \in \mathcal{T}, x \in \mathcal{X}\}$. The following language over Σ_{RW} then represents the set of all executions with a data race.

$$L_{\mathsf{race}} = \bigcup_{\substack{t \neq t' \in \mathcal{T} \\ x \in \mathcal{X}}} \Sigma^*_{\mathsf{RW}} \Big([t, \mathsf{w}(x)][t', \mathsf{w}(x)] + [t, \mathsf{w}(x)][t', \mathsf{r}(x)] + [t, \mathsf{r}(x)][t', \mathsf{w}(x)] \Big) \Sigma^*_{\mathsf{RW}}$$

The race prediction question then asks to check — given an execution σ, is there an execution $\rho \in \mathsf{CReorderings}(\sigma) \cap L_{\mathsf{race}}$? Observe that, L_{race} is a regular language over Σ_{RW}. Likewise, the deadlock prediction question can also be formulated analogously using a regular language L_{deadlock} over an alphabet that also contains information about lock acquire and release events:

$$\Sigma_{\mathsf{RWL}} = \Sigma_{\mathsf{RW}} \cup \{[t, \mathtt{acq}(\ell)], [t, \mathtt{rel}(\ell)] \mid t \in \mathcal{T}, \ell \in \mathcal{L}\}, \tag{1}$$

where $\mathcal{L}$ is a fixed set of lock identifiers. We skip the precise definition of L_{deadlock} here.

3 TRACE LANGUAGES AND PREDICTIVE MONITORING

In this section, we will recall trace[2] languages [Mazurkiewicz 1987], and discuss their membership question, its connections to predictive monitoring and prior complexity-theoretic results. We will finally present our hardness result, which is the first main contribution of this work.

Mazurkiewicz Trace Equivalence. Trace theory, introduced by A. Mazurkiewicz [Mazurkiewicz 1987] is a simple yet systematic framework for reasoning about the computation of concurrent programs. The broad idea of trace theory is to characterize when two executions of a concurrent programs must be deemed equivalent, based on the notion of commutativity of independent actions (or labels). Formally an *independence* (or *concurrency*) relation over an alphabet of actions Σ is

[2] Readers must note that the use of the term *trace* in this paper is specifically distinguished from its more contemporary use (to denote a specific log or sequence of events that happen during an execution). The usage of *trace* in this paper is derived from the more traditional language theoretic notion of Mazurkiewicz *traces* which denote (equivalence) *classes* of strings over some alphabet, instead of a single string modelling a single execution.

an irreflexive and symmetric relation $\mathcal{I} \subseteq \Sigma \times \Sigma$ denoting all pairs of actions that can intuitively be deemed pairwise independent. For simplicity, we will introduce the dependence relation $\mathcal{D} = \Sigma \times \Sigma \setminus \mathcal{I}$, induced from a fixed independence relation $\mathcal{I}$, to denote all pairs of dependent (i.e., not independent) actions. Together, the pair $(\Sigma, \mathcal{I})$ constitutes a *concurrent alphabet*. Then, the *trace equivalence* induced by $\mathcal{I}$, denoted by $\equiv_{\mathcal{I}}$, is the smallest equivalence class over Σ^* such that for every $(a, b) \in \mathcal{I}$ and for every two words $w_1, w_2 \in \Sigma^*$, we have

$$w_1 \cdot a \cdot b \cdot w_2 \equiv_{\mathcal{I}} w_1 \cdot b \cdot a \cdot w_2.$$

For a string $w \in \Sigma^*$, the trace equivalence class of w is $[w]_{\mathcal{I}} = \{w' \mid w \equiv_{\mathcal{I}} w'\}$.

Trace Partial Order. An alternative characterization of trace equivalence is often given in terms of the partial order induced due to the concurrent alphabet. Formally, given a concurrent alphabet $(\Sigma, \mathcal{I})$, with the dependence relation $\mathcal{D} = \Sigma \times \Sigma \setminus \mathcal{I}$, and a sequence of events σ labelled with symbols from Σ, the partial order induced due to σ, denoted by $\leq^{\sigma}_{\mathcal{D}}$, is the smallest partial order over Events_σ such that for any two events $e_1, e_2 \in \mathsf{Events}_\sigma$, if e_1 appears before e_2 (i.e., $e_1 <^{\sigma} e_2$) in the sequence σ and $(\mathsf{lab}(e_1), \mathsf{lab}(e_2)) \in \mathcal{D}$, then $e_1 \leq^{\sigma}_{\mathcal{D}} e_2$. One can then show that, the set of linearizations $\mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}})$ of this partial order precisely captures the set of words that are trace equivalent to the label of σ:

Proposition 3.1. Let σ be an execution over Σ. We have, $[\mathsf{lab}(\sigma)]_{\mathcal{I}} = \{\mathsf{lab}(\sigma') \mid \sigma' \in \mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}})\}$.

As a consequence of Proposition 3.1, we lift the notion of trace equivalence from words over Σ to executions over Σ as follows. Given two executions σ and σ' over the same set of events (i.e., $\mathsf{Events}_\sigma = \mathsf{Events}_{\sigma'}$), we say that $\sigma \equiv_{\mathcal{I}} \sigma'$ if $\sigma' \in \mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}})$. Likewise, we use $[\sigma]_{\mathcal{I}}$ to denote the set of executions σ' for which $\sigma \equiv_{\mathcal{I}} \sigma'$.

The partial order view of trace equivalence has, in fact, been routinely exploited in program analysis for concurrency. Dynamic analysis techniques such as those designed for data race detection [Flanagan and Freund 2009; Itzkovitz et al. 1999] construct a partial order, namely the *happens-before* partial order, computed using timestamps [Fidge 1991; Mathur et al. 2022; Mattern 1989], which essentially characterizes the Mazurkiewicz trace equivalence of an appropriately defined concurrency alphabet. Likewise, optimization techniques employed in bounded model checking such as dynamic partial order reduction [Flanagan and Godefroid 2005] are rooted in Mazurkiewicz trace theory in a similar precise sense.

Mazurkiewicz Equivalence v/s Correct Reorderings. Mazurkiewicz trace equivalence provides a sound (but incomplete) characterization of the space of correct reorderings in the following sense. We first fix an independence relation over the alphabet Σ that soundly characterizes the commutativity induced by the underlying programming languages. Consider, for example, the alphabet Σ_{RWL} described previously in Equation (1) over some set of threads $\mathcal{T}$, memory locations $\mathcal{X}$ and locks $\mathcal{L}$. Let $C_{\mathsf{RW}} = \{(a_1(x), a_2(x)) \mid a_1 = \mathtt{w} \text{ or } a_2 = \mathtt{w}, \text{ and } x \in \mathcal{X}\}$ be the set of conflicting memory operations, and let $C_{\mathsf{L}} = \{(a_1(\ell), a_2(\ell)) \mid a_1, a_2 \in \{\mathtt{acq}, \mathtt{rel}\} \text{ and } \ell \in \mathcal{L}\}$ be the set of conflicting lock operations. Now consider the independence relation defined as follows:

$$\mathcal{I}_{\mathsf{RWL}} = \{([t_1, o_1], [t_2, o_2]) \mid t_1 \neq t_2 \text{ and } (o_1, o_2) \notin C_{\mathsf{RW}} \cup C_{\mathsf{L}}\}$$

The above definition of independence relation has been carefully crafted to ensure that the resulting trace equivalence satisfies two properties. First, the relative order between a read event e_{r} and a conflicting write event e_{w} (i.e., $(\mathsf{lab}(e_{\mathsf{r}}), \mathsf{lab}(e_{\mathsf{w}})) \in C_{\mathsf{RW}}$) does not get flipped, ensuring that for any two Mazurkiewicz equivalent executions σ and σ', the control flow taken by σ will also be taken by σ'. Furthermore, the relative order of conflicting lock operations does not change, ensuring that if σ was well-formed (i.e., critical sections on the same lock do not overlap in σ), then so is σ'. This gives us the following; as before, we use $\mathcal{D}_{\mathsf{RWL}} = \Sigma_{\mathsf{RWL}} \times \Sigma_{\mathsf{RWL}} \setminus \mathcal{I}_{\mathsf{RWL}}$.

Proposition 3.2. Let σ be an execution over Σ_{RWL}. We have $\mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}_{\mathsf{RWL}}}) \subseteq \mathsf{CReorderings}(\sigma)$,

We remark that data race detection techniques like FastTrack [Flanagan and Freund 2009] are thus implicitly predictive and *sound* because they reason about trace equivalence induced by the concurrent alphabet $(\Sigma_{\mathsf{RWL}}, \mathcal{I}_{\mathsf{RWL}})$, at least until the first data race they report [Mathur et al. 2018].

Example 3.1. Consider again the program P (Figure 1a) from Example 2.1. Let $\mathsf{Dep}_{\mathsf{Ex}}$ be the set of dependent instructions, namely those pairs (op_1, op_2) such that there is an object $\mathsf{o} \in \{\mathtt{count}, \mathtt{inputs}\}$ such that $(op_1, op_2) \in \{(\mathtt{write}(\mathsf{o}), \mathtt{write}(\mathsf{o}))\}$. Using this, we can define the independence relation as $\mathcal{I}_{\mathsf{Ex}} = \{([t_1, op_1], [t_2, op_2]) \mid t_1 \neq t_2 \text{ and } (op_1, op_2) \notin \mathsf{Dep}_{\mathsf{Ex}}\}$. Observe that any two consecutive instructions which are independent according to $\mathcal{I}_{\mathsf{Ex}}$ can be commuted without affecting the control flow of the program. Thus, equivalence induced by $\mathcal{I}_{\mathsf{Ex}}$ soundly captures correct reorderings. Finally, observe that the two executions from Figure 1 are also deemed equivalent according to the independence relation defined here: $\sigma_{\mathsf{safe}} \equiv_{\mathcal{I}_{\mathsf{Ex}}} \sigma_{\mathsf{fail}}$.

3.1 Predictive Monitoring for Trace Equivalence

The framework of Mazurkiewicz traces is well-equipped to study the predictive monitoring problem defined henceforth in the context of trace equivalence.

Definition 1 (Predictive Trace Monitoring)**.** Fix a concurrent alphabet $(\Sigma, \mathcal{I})$ and a language $L \subseteq \Sigma^*$. Given an execution σ over Σ as input, the predictive monitoring problem asks to check if there is an execution σ' such that $\sigma' \equiv_{\mathcal{I}} \sigma$ and $\mathsf{lab}(\sigma') \in L$.

We remark that the above problem can be equivalently formulated in terms of words (instead of executions) — given an execution σ specified as the corresponding word $w = \mathsf{lab}(\sigma)$, check if $[w]_{\mathcal{I}} \cap L \neq \varnothing$. As an example, for the alphabet Σ_{Ex} from Example 2.1 and the independence relation $\mathcal{I}_{\mathsf{Ex}}$ defined in Example 3.1, the predictive monitoring question would return YES for the input σ_{safe} because $\sigma_{\mathsf{fail}} \in [\sigma_{\mathsf{safe}}]_{\mathcal{I}_{\mathsf{Ex}}} \cap L_{\mathsf{fail}}$.

Predictive Trace Monitoring against Regular Languages. Bertoni, Mauri and Sabadini [Bertoni et al. 1989] studied the problem of predictive trace monitoring against the class of regular (and context-free) languages and proposed an algorithm whose time complexity is given in terms of a parameter of the concurrency alphabet, which is defined as follows. The *width* $\mathsf{width}(\Sigma, \mathcal{I})$ of the concurrency alphabet $(\Sigma, \mathcal{I})$ is the size of the largest clique of the undirected graph whose vertices are elements of Σ and edges are elements of $\mathcal{I}$. We next recall the result of Bertoni et al. relevant to this exposition, and provide a high-level overview of their algorithm subsequently.

Theorem 3.1 (Theorem 6.2 in [Bertoni et al. 1989])**.** The predictive monitoring problem against regular languages can be solved using an algorithm that uses $O(n^{\alpha})$ time and space for the input execution of size n, where $\alpha = \mathsf{width}(\Sigma, \mathcal{I})$ is the width of the concurrency alphabet.

Ideals. The algorithm due to Bertoni et al. relies on the observation that the set of *prefixes* of an equivalence class of a given execution can be defined using an *ideal*. An ideal X of σ is a subset $\mathsf{X} \subseteq \mathsf{Events}_{\sigma}$ such that for every $e, e' \in \mathsf{Events}_{\sigma}$ such that $e \leq^{\sigma}_{\mathcal{D}} e'$, if $e' \in \mathsf{X}$, we also have $e \in \mathsf{X}$. An event $e \in X$ is said to be *maximal* in X if for every $f \in \mathsf{Events}_{\sigma}$ different from e such that $e \leq^{\sigma}_{\mathcal{D}} f$, we have $f \notin X$. We use $\max(\mathsf{X})$ to denote the (unique) set of maximal events of X. We will use $\mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}}, \mathsf{X})$ to denote the set of linearizations of the set X consistent with $\leq^{\sigma}_{\mathcal{D}}$ i.e., each $\rho \in \mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}}, \mathsf{X})$ is such that $\mathsf{Events}_{\rho} = \mathsf{X}$ and for every $e_1 \leq^{\sigma}_{\mathcal{D}} e_2$, if $e_1, e_2 \in \mathsf{X}$, then $e_1 <^{\rho} e_2$. Given two ideals X and Y of σ, we say that X is an immediate predecessor of Y if there is an event $e \in \mathsf{Y}$ such that $\mathsf{Y} = \mathsf{X} \uplus \{e\}$ (i.e., $e \notin \mathsf{X}$, and $e \in \max(\mathsf{Y})$). We use $\mathsf{pre}(\mathsf{X})$ to denote the set of all immediate predecessors of X. The principal ideal generated by an event $e \in \mathsf{Events}_{\sigma}$, denoted $\mathsf{PIdeal}(e)$, is the smallest ideal of σ that contains e. There is a one-to-one correspondence between an ideal

X and the set of its maximal elements. In particular, an ideal X can be uniquely represented as $\mathsf{X} = \bigcup_{e \in \max(\mathsf{X})} \mathsf{PIdeal}(e)$. The number of maximal elements of an ideal of σ is bounded by α. As a result, the number of ideals of σ is bounded by $O(|\sigma|^{\alpha})$.

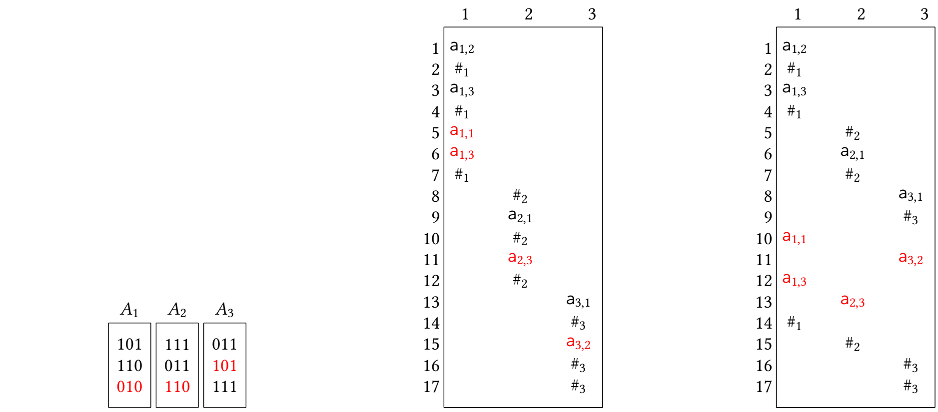
We now recall the algorithm due to Bertoni et al. Let $\alpha = \mathsf{width}(\Sigma, \mathcal{I})$ be the width of the concurrency alphabet. Let $\mathcal{A}_L = (Q, Q_0, \delta, F)$ be the non-deterministic finite automaton corresponding to the regular language L against which we perform predictive trace monitoring; the size of $\mathcal{A}_L$ is assumed to be $O(1)$. Let us also fix the input execution $\sigma \in \Sigma^*$. The algorithm computes, for every ideal X of σ, the set of states $S_{\mathsf{X}} = \{\delta^*(q_0, w) \mid \exists \rho \in \mathsf{Lin}(\leq^{\sigma}_{\mathcal{D}}, \mathsf{X}) \text{ such that } \mathsf{lab}(\rho) = w\}$ that can be reached by the automaton $\mathcal{A}_L$ on some linearization of X. Observe that, $S_{\mathsf{X}} \subseteq Q$ has constant size for any ideal X. This information can be defined inductively using immediate predecessor ideals — $S_{\mathsf{X}} = Q_0$ if $\mathsf{X} = \emptyset$, and otherwise, $S_{\mathsf{X}} = \{q' \mid \exists \mathsf{X}', q, f \text{ such that } \mathsf{X}' \in \mathsf{pre}(\mathsf{X}), q \in S_{\mathsf{X}'}, \{f\} = \mathsf{X} \setminus \mathsf{X}' \text{ and } (q, \mathsf{lab}(f), q') \in \delta\}$. The computation of the sets S_{X} can be performed incrementally in order of the size of the ideals, starting from ideals of the smallest size. The final check for determining if $[\mathsf{lab}(\sigma)]_{\mathcal{I}} \cap L \neq \emptyset$ simply corresponds to checking if $F \cap \mathsf{X}_\sigma \neq \emptyset$, where $\mathsf{X}_\sigma = \mathsf{Events}_\sigma$ is the largest ideal that includes all events. The time spent for each ideal X is $O(1)$, and the total number of ideals is $O(|\sigma|^{\alpha})$, giving us the desired bound of Theorem 3.1.

3.2 Hardness of Predictive Trace Monitoring

The algorithm in [Bertoni et al. 1989] runs in time (and space) $O(n^{\alpha})$ for executions of size n and where α is the width of the concurrent alphabet, in the case when the target language is regular. Even though this is polynomial time (assuming α is constant), it can become prohibitive when deployed for predictive monitoring for executions coming from large-scale concurrent software systems, which routinely generate executions with millions of events. In sharp contrast to this is the case of non-predictive monitoring, which boils down to checking membership in a regular language. The non-predictive monitoring problem for regular languages thus admits a linear-time, one-pass constant space streaming algorithm, the holy grail of runtime verification. The question we ask here is — in the case of predictive trace monitoring, is there a more efficient algorithm than the $O(n^{\alpha})$ algorithm proposed by Bertoni et al. [Bertoni et al. 1989]? Is the exponential dependence on α necessary?

At first glance, predictive trace monitoring, in general, does not admit a constant-space (automata-theoretic) algorithm. Consider, for example, the alphabet $\Sigma = \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3\}$ and the independence relation $\mathcal{I} = \{(\mathsf{a}_i, \mathsf{a}_j) \mid i \neq j \in \Sigma\}$, i.e., all pairs of distinct letters are deemed independent by $\mathcal{I}$. Now consider the language $L = (\mathsf{a}_1\mathsf{a}_2\mathsf{a}_3)^*$. Checking if $[w]_{\mathcal{I}} \cap L \neq \emptyset$, amounts to checking if $w \in L' = \{u \mid \text{the number of occurrences of } \mathsf{a}_1, \mathsf{a}_2 \text{ and } \mathsf{a}_3 \text{ is equal in } u\}$. Since L' is not a regular language (not even context-free), the predictive trace monitoring problem does not admit a constant-space linear-time algorithm for L. In this work, we establish an even tighter (conditional) lower bound (Theorem 3.2). Our hardness result is, in fact, stronger in that it applies even for the case of *star-free* regular languages [McNaughton and Papert 1971; Schützenberger 1965]. Further, our result also establishes that the $O(n^{\alpha})$ algorithm due to Bertoni et al. is in fact optimal.

Complexity Theoretic Conjectures. One of the most widely believed complexity-theoretic conjectures is the Strong Exponential Time Hypothesis (SETH) [Calabro et al. 2009], which states that for every $\epsilon > 0$, no (deterministic or randomized) algorithm determines the satisfiability of a 3SAT formula over n propositions in time $O(2^{(1-\epsilon)n})$. In the proof of our result Theorem 3.2, we will show a reduction from the orthogonal vectors problem k-OV. An instance of k-OV is k sets $\{A_1, \ldots, A_k\}$, each of which is a set of n Boolean vectors over d dimensions, i.e., $A_i \subseteq \{0, 1\}^d$ and $|A_i| = n$. The k-OV problem asks to check if there are k orthogonal vectors, i.e., if there are vectors $a_1 \in A_1, a_2 \in A_2, \ldots, a_k \in A_k$ such that the norm of their pointwise product vector is 0, i.e.,



(a) 3-OV instance with $n = 3$, $d = 3$ (b) Input execution σ (c) Execution σ' with $\sigma' \in [\sigma]_{\mathcal{I}} \cap L$

Fig. 2. Reduction from 3-OV. Symbol $a_{i,j}$ appears in the i^{th} column in (b) and (c). Symbols appearing in different columns are independent. $L = \Sigma^* \{a_{1,1}, a_{2,1}, a_{3,1}\}^+ \{a_{1,2}, a_{2,2}, a_{3,2}\}^+ \{a_{1,3}, a_{2,3}, a_{3,3}\}^+ \Sigma^*$.

$\sum_{j=1}^{d} \prod_{i=1}^{k} a_i[j] = 0$. Under SETH, there is no $O(n^{k-\epsilon} \cdot \text{poly}(d))$ algorithm for k-OV (no matter what $\epsilon > 0$ we pick) [Williams 2018].

Theorem 3.2. Assume SETH holds. For each $\epsilon > 0$, there is no algorithm that solves the predictive trace monitoring problem against star-free regular languages (over a concurrent alphabet with width α) in time $O(n^{\alpha-\epsilon})$ for input words of length n.

Proof. We show a fine-grained reduction from k-OV to the problem of predictive trace monitoring against star-free language. We fix $k, d \in \mathbb{N}$. Then we construct the concurrent alphabet $(\Sigma, \mathcal{I})$, and the star-free language L against which we check predictive trace monitoring, as follows. First, the alphabet Σ is partitioned into k disjoint alphabets, i.e., $\Sigma = \biguplus_{i=1}^{k} \Sigma_i$, where

$$\Sigma_i = \{a_{i,j} \mid 1 \le j \le d\} \cup \{\#_i\}$$

That is, the alphabet consists of one symbol $a_{i,j}$ for every pair (i, j) where $1 \le i \le k$ and $1 \le j \le d$ and additionally one symbol $\#_i$ for each $1 \le i \le k$. The independence relation is such that all symbols across partitions are deemed independent, while those within the same partition are deemed dependent, That is,

$$\mathcal{I} = \bigcup_{i \ne j} \Sigma_i \times \Sigma_j$$

Thus the width of the concurrency alphabet is the number of partitions, i.e., $\alpha = k$. The language L is given by the following regular expression:

$$r = \Sigma^* \cdot \{a_{1,1}, a_{2,1}, \dots, a_{k,1}\}^+ \cdots \{a_{1,d}, a_{2,d}, \dots, a_{k,d}\}^+ \cdot \Sigma^*$$

Observe that $|\Sigma| = k(d+1)$, $\mathcal{I} = k(k-1)(d+1)^2/2$ and $|r| = O(kd)$. Also, observe that L is a star-free language because $\Sigma^* = \varnothing^c$ and for a subset $\Delta \subsetneq \Sigma$, we have $\Delta^+ = \Delta \cdot \Delta^*$ and further $\Delta^* = \left(\bigcup_{a \in \Sigma \setminus \Delta} \Sigma^* a \Sigma^*\right)^c$.

Reduction. Given an instance $\mathcal{A} = \{A_1, A_2, \dots, A_k\}$ of k-OV with $|A_i| = n$ and $A_i \subseteq \{0,1\}^d$ for every i, we construct an execution σ such that $\mathcal{A}$ is a positive instance of k-OV iff $[\text{lab}(\sigma)]_{\mathcal{I}} \cap L \ne \varnothing$. Our construction ensures that $|\sigma| \le k(d+1)n$. For each i, let us denote the vectors of A_i using

$\{v_1^{(i)}, v_2^{(i)}, \ldots, v_n^{(i)}\}$. The execution σ is then a concatenation, obtained by successively appending smaller sub-executions, one for each set A_i:

$$\sigma = \sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_k.$$

Further, for $1 \leq i \leq k$, the sub-execution σ_i is a concatenation of smaller sub-executions corresponding to each vector in A_i:

$$\sigma_i = \sigma_{i,1} \circ \#_i \circ \sigma_{i,2} \circ \#_i \cdots \#_i \circ \sigma_{i,n} \circ \#_i$$

Further, for each $1 \leq i \leq k$ and $1 \leq l \leq n$, the trace $\sigma_{i,l}$ encodes the vector $v_l^{(i)}$:

$$\sigma_{i,l} = b_{i,1,l} \circ b_{i,2,l} \circ \cdots \circ b_{i,d,l}$$

where

$$b_{i,j,l} = \begin{cases} \mathsf{a}_{i,j} & \text{if } v_l^{(i)}[j] = 0 \\ \varepsilon & \text{otherwise} \end{cases}$$

Figure 2 illustrates our construction. The string σ' (Figure 2c) is equivalent to σ (Figure 2b) and also belongs to L; symbols marked red highlight membership in L and correspond to the vectors whose dot product is 0 (also marked in red in Figure 2a).

Correctness. We now argue for the correctness of our reduction.
($\Rightarrow$) Consider $v_{t_1}^{(1)} \in A_1, v_{t_2}^{(2)} \in A_2, \ldots, v_{t_k}^{(k)} \in A_k$ such that $v_{t_1}^{(1)} \cdot v_{t_2}^{(2)} \cdot \ldots \cdot v_{t_k}^{(k)} = 0$. This means, for every $j \in \{1, \ldots, d\}$, there is at least one $i \in \{1, \ldots, k\}$ such that $v_{t_i}^{(i)}[j] = 0$. σ' can now be described as follows:

$$\sigma' = \sigma'_{\mathsf{pre}} \circ \sigma'_{\mathsf{mid}} \circ \sigma'_{\mathsf{post}}.$$

The prefix σ'_{pre} (resp. σ'_{post}) is obtained by successively concatenating the first $t_i - 1$ (resp. last $n - t_i$) sub-executions of σ_i:

$$\begin{array}{rcl} \sigma'_{\mathsf{pre}} & = & \sigma_1[1 \ldots t_1 - 1] \circ \ldots \circ \sigma_k[1 \ldots t_k - 1] \\ \sigma'_{\mathsf{post}} & = & \sigma_1[t_1 + 1 \ldots n] \circ \ldots \circ \sigma_k[t_k + 1 \ldots n] \end{array}$$

where $\sigma_i[p \ldots q] = \sigma_{i,p} \circ \sigma_{i,p+1} \cdots \circ \sigma_{i,q}$. The sub-execution σ'_{mid} is obtained by successive concatenation of sub-executions corresponding to the j^{th} components of each vector $v_{t_i}^{(i)}$:

$$\sigma'_{\mathsf{mid}} = \sigma'_{\mathsf{mid},1} \circ \sigma'_{\mathsf{mid},2} \circ \ldots \circ \sigma'_{\mathsf{mid},d} \circ \#_1 \circ \#_2 \cdots \#_k$$

where for each $1 \leq j \leq d$,

$$\sigma'_{\mathsf{mid},j} = b_{1,j,t_1} \circ b_{2,j,t_2} \circ \ldots \circ b_{k,j,t_k}$$

First, observe that σ' does not flip the order of any two events that are dependent. Thus, $\sigma' \equiv_{\mathcal{I}} \sigma$. Second, σ'_{mid} has a prefix (namely the one that does not include the residual $\#_i$ symbols) that matches the regular expression $\{\mathsf{a}_{1,1}, \ldots, \mathsf{a}_{k,1}\}^+ \ldots \{\mathsf{a}_{1,d}, \ldots, \mathsf{a}_{k,d}\}^+$. This is because or every $j \in \{1, \ldots, d\}$, there is at least one $i \in \{1, \ldots, k\}$ such that $v_{t_i}^{(i)}[j] = 0$. Thus, $\sigma' \in L$.

($\Leftarrow$) Consider an equivalent execution $\sigma' \equiv_{\mathcal{I}} \sigma$ such that $\sigma' \in L$. Clearly, σ' must contain a substring σ'_{mid} which belongs to the language $\{\mathsf{a}_{1,1}, \ldots, \mathsf{a}_{k,1}\}^+ \ldots \{\mathsf{a}_{1,d}, \ldots, \mathsf{a}_{k,d}\}^+$. Let $\sigma'_{\mathsf{mid},i}$ be the subsequence of σ'_{mid} obtained by projecting it to Σ_i. We remark that for each such i, there must be a common sub-execution σ_{i,t_i} of σ such that the events of $\sigma'_{\mathsf{mid},i}$ belong to σ_{i,t_i} (recall that $\sigma_{i,l}$ corresponds to the vector $v_l^{(i)} \in A_i$). This is because any two subexecutions $\sigma_{i,\beta}$ and $\sigma_{i,\beta'}$, are separated by the separator $\#_i$ which is dependent with each symbol in $\sigma_{i,\beta}$ and $\sigma_{i,\beta'}$. We can thus choose indices $t_1, t_2, \ldots, t_k$ such that events of $\sigma'_{\mathsf{mid},i}$ belong to the unique subtrace σ_{i,t_i}; If for some i, σ'_{mid} contains no symbol in Σ_i, t_i is chosen arbitrarily. We will now argue that the vectors $v_{t_1}^1 \in A_1, \ldots, v_{t_k}^k \in A_k$

are orthogonal. This follows easily since For each dimension $j \in \{1, \ldots, d\}$, σ'_{mid} contains at least one symbol $\mathsf{a}_{i,j}$ for some $i \in \{1, \ldots, k\}$, which implies that $v^{(i)}_{t_i}[j] = 0$.

Time Complexity. Recall that $\Sigma = |k(d + 1)|$, $|\mathcal{I}| = k(k - 1)(d + 1)^2/2$, $\alpha = k$, $|r| = \text{poly}(d)$, and $|\sigma| \leq kn(d + 1)$. The time taken to construct σ is also $O(n \cdot k \cdot (d + 1))$. If predictive trace monitoring against L can be decided in time $|\sigma|^{\alpha-\epsilon}\text{poly}((|\Sigma|\cdot|r|)) \leq (kn(d + 1))^{k-\epsilon}\texttt{poly}(d) = n^{k-\epsilon}\texttt{poly}(d)$, then k-OV with set size n can also be solved in time $n^{k-\epsilon}\text{poly}(d)$, which refutes SETH (since SETH implies the absence of such an algorithm). □

4 PATTERN AND GENERALIZED PATTERN LANGAUGES

In this section, we describe the class of *generalized pattern languages*, which is the central object of our study. Later, in Section 5 we show that for this class of languages, the predictive trace monitoring problem becomes highly tractable, in that, it admits a constant-space streaming algorithm that works in linear time.

Definition 2 (Pattern Languages)**.** Let Σ be an alphabet. A language L is said to be a pattern language of dimension $d \in \mathbb{N}$ over Σ, if it is of the form

$$L = \Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_d \Sigma^*,$$

where $a_1, a_2, \ldots, a_d \in \Sigma$; we use $\texttt{Patt}_{a_1,\ldots,a_d}$ to denote the above pattern language. The class of all pattern languages will be denoted by PATT.

The above definition of pattern languages has been inspired from their potential target application in dynamic analysis for detecting bugs in concurrent and distributed applications. Over the years, a large class of concurrency bug-detection techniques has been proposed that enhance the effectiveness of otherwise exhaustive enumeration-style testing, by leveraging the hypothesis that "*many bugs in programs depend on the precise ordering of a small number of events ...*" [Chistikov et al. 2016]. The additional knowledge of "small bug depth" can then be leveraged to simplify testing, in that, only a small number of behaviours need to be analysed. This has been the key to tackling the interleaving explosion problem encountered during testing for concurrent software [Burckhardt et al. 2010; Musuvathi and Qadeer 2006]. In our setting, where the events are labelled from an alphabet (denoting, say instructions, function entry and exit points, synchronization operations, or even load and store operations), the natural analogue of this hypothesis asks to determine if there is some small d and labels $a_1, \ldots, a_d$ so that some d events $e_1, e_2, \ldots, e_d$ (not necessarily observed in this order) with these labels $\text{lab}(e_1) = a_1, \ldots, \text{lab}(e_d) = a_d$, can be organized in the precise order where e_i is followed by e_{i+1} for each $i \leq d$. This is precisely the predictive monitoring question against the pattern language $\texttt{Patt}_{a_1,\ldots,a_d} = \Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_d \Sigma^*$ and highlights our motivation behind the class of specifications we introduce in Definition 2. As a final remark, observe that the language L_{fail} from Example 2.1 is indeed a pattern language (of dimension $d = 4$).

In the following, we define a more general class of languages, thereby enhancing the class of specifications we consider.

Definition 3 (Generalized Pattern Languages)**.** Let Σ be an alphabet. A language $L \subseteq \Sigma^*$ is said to be a generalized pattern language if it is a finite union $L = L_1 \cup L_2 \cdots \cup L_m$, where for each L_i ($1 \leq i \leq m$), we have (a) $L_i = \emptyset$, or (b) $L_i = \{\varepsilon\}$, or (c) $L_i \in$ PATT is a pattern language.

4.1 Properties of Pattern and Generalized Pattern Languages

In this section, we study language theoretic properties of our proposed class of languages.

Topological Characterization. Pattern languages and generalized pattern languages are clearly regular languages. Here, we provide a finer characterization — they are also star-free languages. Recall that star-free languages over Σ are those that can be constructed by using $\varnothing$, $\{\varepsilon\}$, $\{a\}$ (where $a \in \Sigma$ is some symbol), and inductively composing them using concatenation, union, intersection, and complementation. We remark that Theorem 3.2 shows that the complete class of star-free regular languages are not amenable to efficient predictive trace monitoring.

Proposition 4.1. Every generalized pattern language is a star-free regular language.

Let us now examine the closure properties of our proposed class of languages.

Closure Under Union. Let us first consider closure under union for PATT. Consider the two patterns $\mathtt{Patt}_a$ and $\mathtt{Patt}_b$, where $a \neq b \in \Sigma$. We can show using contradiction that there is no pattern $\mathtt{Patt}_{c_1,\ldots,c_k}$ such that $\mathtt{Patt}_{c_1,\ldots,c_k} = \mathtt{Patt}_a \cup \mathtt{Patt}_b$. Suppose on the contrary that this holds. Then, $a, b \in c_1, \ldots, c_k$ as otherwise $(\Sigma \setminus \{a, b\}) \cap \mathtt{Patt}_{c_1,\ldots,c_k} \neq \varnothing$. This means that $a \notin \mathtt{Patt}_{c_1,\ldots,c_k}$ since every string in $\mathtt{Patt}_{c_1,\ldots,c_k}$ must contain b, thereby giving us a contradiction. On the other hand, the class of generalized pattern languages is clearly closed under finite union.

Closure Under Intersection. Consider two pattern languages $L_1 = \mathtt{Patt}_{a_1,\ldots,a_d}$ and $L_2 = \mathtt{Patt}_{b_1,\ldots,b_l}$. Then, a word $w \in L_1 \cap L_2$ must have a subsequence $a_1, \ldots, a_d$ as well as a sub-sequence $b_1, \ldots, b_l$. Let us use the notation $u \odot v$ to denote the shortest words that contain u and v as subsequences. Let us use the notation $\sqsubseteq$ ($\sqsubset$) to denote subsequence (strict subsequence) relation between two words.

$$u \odot v = \{w \in \Sigma^* \mid u \sqsubseteq w,\ v \sqsubseteq w,\ \forall w' \sqsubset w, u \not\sqsubseteq w' \text{ or } v \not\sqsubseteq w'\}$$

Then, we remark that $L_1 \cap L_2 = \bigcup_{u \in a_1,\ldots,a_d \odot b_1,\ldots,b_l} \mathtt{Patt}_u$. The class of generalized pattern languages is closed under intersection, since intersection distributes over union.

Closure under Complementation. The class of pattern languages is not closed under complementation. Consider the pattern language $L = \mathtt{Patt}_a$. Observe that $L^c = (\Sigma \setminus \{a\})^*$ which cannot be a pattern language. Assume on the contrary that there was a pattern language $\mathtt{Patt}_{b_1,\ldots,b_l} = L^c$; observe that $a \cdot b_1 \cdot \ldots \cdot b_l \in \mathtt{Patt}_{b_1,\ldots,b_l}$ giving us a contradiction right away. In fact, the language L^c here is different from $\{\varepsilon\}$, and $\varnothing$, and is also not a finite union of multiple pattern languages (each of which will otherwise contain a string containing a). Thus, even the class of generalized pattern languages is not closed under complementation.

Closure Under Concatenation. Consider two pattern languages $L_1 = \mathtt{Patt}_{a_1,\ldots,a_d}$ and $L_2 = \mathtt{Patt}_{b_1,\ldots,b_l}$. Observe that the language $L = \mathtt{Patt}_{a_1,\ldots,a_d \circ b_1,\ldots,b_l}$ is indeed the concatenation of L_1 and L_2, i.e., $L = L_1 \circ L_2$. As a result, the class PATT is closed under concatenation. It is easy to see that the class of generalized pattern languages is also closed under concatenation.

Closure Under Kleene Star. Pattern languages are certainly not closed under Kleene star because $\varepsilon \notin \mathtt{Patt}_a$, but $\varepsilon \in (\mathtt{Patt}_a)^*$. However, the class of generalized pattern languages is closed under the Kleene star operation, because of the observation that for a pattern language $L \in \mathsf{PATT}$, we have $L^* = L \cup \{\varepsilon\}$ which can be proved inductively.

To summarize, we have the following closure properties. The formal proof is presented in the extended version of our paper [Ang and Mathur 2023a].

Theorem 4.1 (Closure properties of Pattern Languages)**.** The class of pattern languages is closed under finite concatenation, but not under union, intersection, complementation, and Kleene star.

Theorem 4.2 (Closure properties of generalized Pattern Languages)**.** The class of generalized pattern languages is closed under finite union, finite intersection, finite concatenation, and Kleene star but not under complementation.

5 PREDICTIVE MONITORING AGAINST GENERALIZED PATTERN LANGUAGES

In Section 3.2, we have established that the problem of predictive trace monitoring against regular languages does not admit an algorithm that runs faster than $O(n^{\alpha})$, assuming SETH holds. In this section, we show that, the case of pattern languages and generalized pattern languages (defined in Section 4) admit a more efficient — constant-space linear-time streaming — algorithm.

5.1 Overview of the Algorithm

Recall that a generalized pattern language is a union of pattern languages, $\{\varepsilon\}$ or $\varnothing$, and predictive monitoring against the latter two can trivially be performed in constant space. We instead show a constant-space algorithm for predictive monitoring against pattern languages, which would essentially imply a constant-space algorithm for the larger class of generalized pattern languages. Our algorithm (for pattern languages) is based on several crucial insights: here we briefly discuss them and give details in subsequent sections. In order to give intuitions behind these insights, we will define some subproblems (Problem 5.1, Problem 5.2). We fix the concurrent alphabet $(\Sigma, \mathcal{I})$. The corresponding dependence relation is $\mathcal{D} = \Sigma \times \Sigma \setminus \mathcal{I}$. We also fix the pattern language $\mathtt{Patt}_{a_1,\ldots,a_d}$ of dimension d and the input execution σ over $(\Sigma, \mathcal{I})$. Let us now define some useful notations.

Let $\tau = \langle e_1, \ldots, e_m \rangle$ be a tuple of events of σ such that $e_1 <^{\sigma} \ldots <^{\sigma} e_m$. Let $b_1, \ldots, b_m$ be a sequence of m labels from Σ. τ is said to be a *shuffle of* $b_1, \ldots, b_m$ if $\langle \mathsf{lab}(e_1), \ldots, \mathsf{lab}(e_m) \rangle$ is a permutation of $\langle b_1, \ldots, b_m \rangle$. τ is said to be a *partial candidate* tuple with respect to our pattern $\mathtt{Patt}_{a_1,\ldots,a_d}$, if it is a shuffle of some subsequence of $a_1, \ldots, a_d$. τ is said to be a *(complete) candidate* tuple with respect to our pattern $\mathtt{Patt}_{a_1,\ldots,a_d}$, if it is a shuffle of $a_1, \ldots, a_d$.

Definition 4 (Partial and Complete Admissible Tuples)**.** Let τ be a tuple of m events such that τ is a partial candidate tuple. τ is said to be admissible with respect to $\mathtt{Patt}_{a_1,\ldots,a_d}$ if there is an execution $\sigma' \equiv_{\mathcal{I}} \sigma$ and a permutation $\tau' = \langle e_{i_1}, \ldots, e_{i_m} \rangle$ of τ, such that $\langle \mathsf{lab}(e_{i_1}), \ldots, \mathsf{lab}(e_{i_m}) \rangle$ is a subsequence of $\langle a_1, \ldots, a_d \rangle$ and τ' is subsequence of σ'. As a special case, τ is a complete admissible tuple if it is a *complete* candidate and also, admissible.

We remark that the admissibility of a tuple τ, if true, must be witnessed as a special permutation of τ. Assume that τ is a partial candidate tuple, which is a shuffle of $a_{j_1}, \ldots, a_{j_m}$, a subsequence of $a_1, \ldots, a_d$. Then, there is a *special* permutation $\tau^{\dagger} = \langle e_{i_1^{\dagger}}, \ldots, e_{i_m^{\dagger}} \rangle$ of τ, such that $\langle \mathsf{lab}(e_{i_1^{\dagger}}), \ldots, \mathsf{lab}(e_{i_m^{\dagger}}) \rangle = \langle a_{j_1}, \ldots, a_{j_m} \rangle$; more importantly, if τ is admissible, then for every $\sigma' \equiv_{\mathcal{I}} \sigma$ that witnesses the admissibility of τ, $\tau^{\dagger}$ is a subsequence of σ'. Observe that this special permutation can be obtained by sorting τ according to $\langle a_1, \ldots, a_d \rangle$ while additionally ensuring that events with the same label do not get flipped. Henceforth, we will use the notation $\mathsf{sort}(\tau, \langle a_{j_1}, \ldots, a_{j_m} \rangle)$.

The first subproblem we consider pertains to checking if a given sequence of events is admissible.

Problem 5.1. Given an execution σ and a candidate tuple τ of events from σ, check whether τ is admissible with respect to $\mathtt{Patt}_{a_1,\ldots,a_d}$.

A naive approach to solving this problem would enumerate all $O(|\sigma|!)$ linearizations and check membership in $\mathtt{Patt}_{a_1,\ldots,a_d}$ for each of them in total time $O(|\sigma| \cdot |\sigma|!)$, which is prohibitive. A different and better approach instead relies on the partial order view of Mazurkiewicz traces. Here, one considers the graph $G_{\sigma} = (V_{\sigma}, E_{\sigma})$ corresponding to the partial order $\leq^{\sigma}_{\mathcal{D}}$, where $V_{\sigma} = \mathsf{Events}_{\sigma}$ and E_{σ} captures immediate edges of $\leq^{\sigma}_{\mathcal{D}}$. One can get a graph G'_{σ} by adding additional edges

$\{(e_{i_1^\dagger}, e_{i_2^\dagger}), \ldots, (e_{i_{d-1}^\dagger}, e_{i_d^\dagger})\}$ derived from the special permutation $\langle e_{i_1^\dagger}, \ldots, e_{i_d^\dagger}\rangle = \mathsf{sort}(\tau, \langle a_1, \ldots, a_d\rangle)$. This reduces the admissibility of τ to the acyclicity of G'_σ, which can be checked in time $O(|\sigma|)$ since the degree of each vertex is constant. However, the space utilization of this graph-theoretic algorithm is also linear. In Section 5.2, we will show that Problem 5.1 can be solved using a streaming constant-space algorithm, that uses *after sets* (Definition 6). After sets succinctly capture causality induced by the corresponding partial order and can be used to check the admissibility of a candidate tuple in a streaming fashion.

The second subproblem asks whether there exists an admissible tuple in the given execution.

Problem 5.2. Given an execution σ, check whether there is a candidate tuple τ of events from σ, such that τ is admissible with respect to $\mathtt{Patt}_{a_1,\ldots,a_d}$.

In other words, Problem 5.2 asks there is a $\sigma' \equiv_{\mathcal{I}} \sigma$ such that $\mathsf{lab}(\sigma') \in \mathtt{Patt}_{a_1,\ldots,a_d}$. How do we solve Problem 5.2? Again, a naive approach would enumerate all $O(|\sigma|^d)$ candidate tuples and check their admissibility by repeatedly invoking our algorithm for Problem 5.1. This is again prohibitive, in terms of both its time and space usage. On the other hand, in Section 5.3, we design a linear-time streaming algorithm for Problem 5.2 whose overall space usage is constant. The high-level insights behind this algorithm are as follows. First, our algorithm runs in a streaming fashion, and tracks not just complete candidate tuples, but also *partial* tuples which can potentially be extended to complete candidate tuples. Second, our algorithm is *eager*, in that, it determines whether a partial tuple is admissible, i.e., if it can be extended to a complete admissible tuple, and proactively discards all other partial tuples. The problem of checking if a partial tuple is admissible can, in fact, be solved in constant space using an algorithm similar to Algorithm 1 that solves Problem 5.1. Nevertheless, even the number of partial admissible tuples is still large ($O(|\sigma|^d)$). Our third and most important insight, formalized in Lemma 5.3, tackles this problem — we only need to track constantly many partial admissible tuples, those that are maximum in some precise sense.

Equipped with the above insights, the high-level description of our algorithm for predictive monitoring against pattern languages is the following. The algorithm processes one event at a time and tracks the maximum partial admissible tuple of each kind. When processing a new event of the given execution, it checks whether any existing partial tuple can be extended to another (partial or complete) admissible tuple. If at any point, a complete admissible tuple can be constructed, the algorithm terminates and declares success. Otherwise, there is no complete admissible tuple in the entire execution.

In Section 5.2, we discuss the details of how we solve Problem 5.1 in constant space. In Section 5.3, we present the details of our overall algorithm for predictive trace monitoring against pattern regular languages (Problem 5.2).

5.2 Checking Admissibility of Candidate Tuples

Our first observation towards solving Problem 5.1 is that in order to check the admissibility of a candidate tuple τ, it suffices to examine only pairs of events that appear in τ, which, thankfully, is a local property of τ. We formalize this using the notion of locally admissible tuple. Intuitively, τ is locally admissible when for every pair of events (e, e') in τ, if the order in which e and e' appear in τ is different from their order in the target $\tau^\dagger = \mathsf{sort}(\tau, \langle a_1, \ldots, a_d\rangle)$, then they are incomparable according to $\leq^\sigma_{\mathcal{D}}$. This observation can be generalized to partial candidate tuples as well:

Definition 5 (Locally Admissible Tuple). Let $\tau = \langle e_1, \ldots, e_m\rangle$ be a partial candidate tuple of an execution σ with respect to $\mathtt{Patt}_{a_1,\ldots,a_d}$, so that it is a shuffle of $a_{j_1}, \ldots, a_{j_m}$, a subsequence of $a_1, \ldots, a_d$. Let $\langle e_{i_1^\dagger}, \ldots, e_{i_m^\dagger}\rangle = \mathsf{sort}(\tau, \langle a_{j_1}, \ldots, a_{j_m}\rangle)$. We say that τ is partially locally admissible with respect to $\mathtt{Patt}_{a_1,\ldots,a_d}$ if for all $i_r^\dagger \neq i_s^\dagger$, we have $s < r \Rightarrow e_{i_r^\dagger} \not\leq_{\mathcal{D}}^{\sigma} e_{i_s^\dagger}$. As a special case, τ is completely locally admissible if it is additionally a complete candidate.

A simple intuitive implication of Definition 5 is the following. Consider again the directed acyclic graph G_σ of induced partial order $\leq_{\mathcal{D}}^{\sigma}$. The local admissibility of a tuple $\tau = \langle e_1, \ldots, e_d \rangle$ essentially means that for every two vertices $e_{i_r^\dagger}, e_{i_s^\dagger}$ that need to be flipped, the addition of the edge $(e_{i_r^\dagger}, e_{i_s^\dagger})$ alone does not introduce any cycle in this graph.

In the following, we establish a clear connection between admissibility and local admissibility. The proof of Lemma 5.1 can be found in [Ang and Mathur 2023a].

Lemma 5.1. Let $\tau = \langle e_1, \ldots, e_m \rangle$ be a partial candidate tuple of an execution σ with respect to $\mathtt{Patt}_{a_1,\ldots,a_d}$, which is a shuffle of $a_{j_1}, \ldots, a_{j_m}$, a subsequence of $a_1, \ldots, a_d$. We have,

$$\tau \text{ is admissible} \quad \iff \quad \tau \text{ is locally admissible}$$

The above result is interesting, yet subtle. The intuitive implication of this lemma is that if each flipped edge $(e_{i_r^\dagger}, e_{i_s^\dagger})$ individually does not introduce any cycle in G_σ, then, in fact, the simultaneous addition of all such edges also does not introduce a cycle.

While the criterion of local admissibility and its equivalence with the admissibility are important ingredients towards our solution for Problem 5.1, it is not immediately amenable to a space-efficient algorithm. In particular, we cannot afford to construct a linear-sized graph and check the reachability in order to decide local admissibility. Fortunately, our next ingredient (Definition 6) effectively tackles this challenge.

Definition 6. For an execution σ and an event $e \in \mathsf{Events}_\sigma$, the after set of e in a prefix ρ of σ is defined as $\mathsf{After}_\rho(e) = \{\mathsf{lab}(f) \mid f \in \mathsf{Events}_\rho \text{ and } e \leq_{\mathcal{D}}^{\sigma} f\}$.

Observe that, for any event e and prefix ρ, $\mathsf{After}_\rho(e) \subseteq \Sigma$, thus this set is of constant size. A direct consequence of Definition 6 is that after sets are sufficient to check causality between two events.

Proposition 5.1. Let σ be an execution and ρ be a prefix of σ that ends at an event f. Let e be an event such that $e <^{\rho} f$. We have, $\mathsf{lab}(f) \in \mathsf{After}_\rho(e) \iff e \leq_{\mathcal{D}}^{\sigma} f$.

Our algorithm maintains after sets of the events participating in the candidate tuple. Observe that the after set of an event e is parameterized on a prefix ρ of a prefix of σ. The next crucial observation, formally stated in Lemma 5.2, is that this set can be updated incrementally (see [Ang and Mathur 2023a] for the proof).

Lemma 5.2. Let σ be an execution and $\rho, \rho \circ f$ be prefixes of σ for some event f. Let e be an event in ρ. We have

$$\mathsf{After}_{\rho\circ f}(e) = \begin{cases} \mathsf{After}_\rho(e) \cup \{\mathsf{lab}(f)\} & \text{if } \exists a \in \mathsf{After}_\rho(e) \text{ s.t. } (a, \mathsf{lab}(f)) \in \mathcal{D}, \\ \mathsf{After}_\rho(e) & \text{otherwise.} \end{cases}$$

Equipped with Lemma 5.1, Proposition 5.1, and Lemma 5.2, we can now design the constant-space streaming algorithm for Problem 5.1 in Algorithm 1. The algorithm first constructs the target tuple $\tau^\dagger$ which is uniquely determined from τ. It then scans the execution by processing one event at a time in the order of the given execution. At each event, it updates the after sets of all the events occurring in τ (line 4). Then, when we see an event from τ, we also check the violations of local admissibility proactively (line 8).

Algorithm 1: Constant-Space Algorithm for Checking Admissibility

Input: $\sigma \in \Sigma^*$, candidate tuple $\tau = \langle e_1, \dots, e_d \rangle$
Output: YES if τ is admissible; NO otherwise

Let $\tau^\dagger = \mathsf{sort}(\tau, \langle a_1, \dots, a_d \rangle)$;
foreach $e \in \tau$ **do** $\mathbb{A}_e \leftarrow \varnothing$;
for $f \in \sigma$ **do**
  foreach $e \in \tau$ **do** `// Update after sets`
    if $\exists a \in \mathbb{A}_e$ *s.t.* $(a, \mathsf{lab}(f)) \in \mathcal{D}$ **then** $\mathbb{A}_e \leftarrow \mathbb{A}_e \cup \{\mathsf{lab}(f)\}$;
  if $f \in \tau$ **then**
    $\mathbb{A}_f \leftarrow \{\mathsf{lab}(f)\}$
    foreach $e \in \tau$ **do** `// Check local admissibility`
      if *f occurs before e in $\tau^\dagger$ and* $\mathsf{lab}(f) \in \mathbb{A}_e$ **then return** *NO* ;
return *YES*;

5.3 Checking Existence of Admissible Tuples

Recall that the naive algorithm which enumerates all $O(n^d)$ d-tuples consumes $O(n^d)$ time and space, even though we have a constant-space algorithm to check their admissibility. Our algorithm for solving Problem 5.2 instead runs in a streaming fashion and uses constant space. To do so, the algorithm tracks not only complete candidates but also partial ones. It incrementally extends those partial candidates as it processes new events. A distinct and crucial feature of our algorithm is that it eagerly discards those partial candidates that cannot be extended to a complete admissible tuple. In other words, it tracks only partially admissible tuples. Observe that Algorithm 1 can be easily adjusted to check partial admissibility.

Notice that, compared with the number of partial candidates, the number of partially admissible tuples is not reduced significantly, and is still $O(n^d)$. However, tracking only partially admissible tuples (instead of all partial candidates) allows us to bound the number of tuples we track. In particular, for a set of partially admissible tuples with the same label, we can afford to track only the maximum element. We observe that the number of all kinds of labels is the number of all permutations of all subsequences of $a_1, \dots, a_d$, which is $1! + \cdots + d!$, a constant number. This is also the number of partially admissible tuples we track at any point when processing an execution. The next lemma (Lemma 5.3) shows that the maximum element exists and is unique.

Before we formally show the existence of the maximum element, let us fix some helpful notations. Let $\tau_1 = \langle e_1^1, \dots, e_m^1 \rangle, \tau_2 = \langle e_1^2, \dots, e_m^2 \rangle$ be two sequences of events in the execution σ, such that $\mathsf{lab}(\tau_1) = \mathsf{lab}(\tau_2)$. We say that $\tau_1 \trianglelefteq \tau_2$ if for all $1 \le i \le m$, $e_i^1 <^\sigma e_i^2$ or $e_i^1 = e_i^2$. Also, we use $\tau_1 \triangledown \tau_2$ to denote the tuple $\langle e_1, \dots, e_m \rangle$, where for $1 \le i \le m$, $e_i = e_i^2$ if $e_i^1 <^\sigma e_i^2$, and e_i^1 otherwise. Let $\langle b_1, \dots, b_m \rangle$ be a permutation of a subsequence of $\langle a_1, \dots, a_d \rangle$. We use $\partial\mathsf{AdTpls}(\sigma, \langle b_1, \dots, b_m \rangle)$ to denote the set of all partially admissible tuples τ in σ, such that $\mathsf{lab}(\tau) = \langle b_1, \dots, b_m \rangle$.

Lemma 5.3. Let σ be an execution and ρ be a prefix of σ. Let $\langle b_1, \dots, b_m \rangle$ be a permutation of a subsequence of $\langle a_1, \dots, a_d \rangle$. Then the set $\partial\mathsf{AdTpls}(\rho, \langle b_1, \dots, b_m \rangle)$ has a unique maximum tuple τ_0 with regard to $\trianglelefteq$. That is, for every $\tau \in \partial\mathsf{AdTpls}(\rho, \langle b_1, \dots, b_m \rangle)$, we have $\tau \trianglelefteq \tau_0$.

The above lemma states the uniqueness of the maximum element. But observe that existence implies uniqueness. Consider τ_1 and τ_2 that both are maximum, then $\tau_1 \trianglelefteq \tau_2$ and $\tau_2 \trianglelefteq \tau_1$. From the definition, this implies that $\tau_1 = \tau_2$. The proof of existence relies on two observations. First, the set $\partial\mathsf{AdTpls}(\rho, \langle b_1, \dots, b_m \rangle)$ is closed under $\triangledown$, which is formalized in Lemma 5.4. Second, from the

Algorithm 2: Constant-Space Algorithm for Checking Existence of Admissible Tuples

Input: $\sigma \in \Sigma^*$
Output: YES if there exists an admissible τ; NO otherwise
1 Let (M: Permutations of prefixes of $a_1, \ldots, a_d \longrightarrow$ Partial admissible tuples) be an empty map;
2 **for** $f \in \sigma$ **do**
3   **if** $\mathsf{lab}(f) \in \langle a_1, \ldots, a_d \rangle$ **then**
4   **foreach** *π which is a permutation of a prefix of $a_1, \ldots, a_d$ ending with* $\mathsf{lab}(f)$ **do**
5   Let $\pi' = \pi[: -1]$;
6   **if** *M contains π' and $M(\pi') \circ f$ is partially admissible* **then**
                `// compute the new maximum partial admissible tuple`
7                 $M(\pi) \leftarrow M(\pi') \circ f$;
8   **if** *M contains a key of length d* **then**
9   **return** *YES*;
10 **return** *NO*;

definition, $\tau = \tau_1 \triangledown \tau_2$ implies that $\tau_1 \trianglelefteq \tau$ and $\tau_2 \trianglelefteq \tau$. Combined the above two observations, we know that $\triangledown_{\tau \in \partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)} \tau$ is in $\partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)$ and also, is the unique maximum element.

Lemma 5.4 (Closure Property under Join)**.** Let σ be an execution and ρ be a prefix of σ. Let $\langle b_1, \ldots, b_m \rangle$ be a permutation of a subsequence of $\langle a_1, \ldots, a_d \rangle$. The set $\partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)$ is closed under operation $\triangledown$. To be specific, for all $\tau_1, \tau_2 \in \partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)$, we have $\tau_1 \triangledown \tau_2 \in \partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)$.

Our final ingredient is the observation that for any ρ, a prefix of the execution σ, the maximum partially admissible tuple of each kind in ρ can be computed incrementally using the maximum ones computed in the immediate prefix of ρ. In the following Lemma 5.5, we formally establish this observation. From now on, we use $\max(S)$ to denote the maximum element of the set S of partially admissible tuples.

Lemma 5.5. Let σ be an execution. Let $\rho' = \rho \circ f$ be a prefix of σ, where ρ is also a prefix of σ and f is an event. Let $\langle b_1, \ldots, b_m \rangle$ be a permutation of a subsequence of $\langle a_1, \ldots, a_d \rangle$. If $\mathsf{lab}(f) = b_m$ and $\max(\partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_{m-1} \rangle)) \circ f$ is partially admissible, then we have

$$\max(\partial\mathsf{AdTpls}(\rho \circ f, \langle b_1, \ldots, b_m \rangle)) = \max(\partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_{m-1} \rangle)) \circ f.$$

Otherwise, we have

$$\max(\partial\mathsf{AdTpls}(\rho \circ f, \langle b_1, \ldots, b_m \rangle)) = \max(\partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)).$$

The above lemma ensures the feasibility of our streaming algorithm. Recall that we only keep track of the maximum element of each kind for a processed prefix. When processing a new event, the algorithm can compute the new maximum partially admissible tuples easily from those old ones tracked so far. The proofs of Lemmas 5.3, 5.4 and 5.5 are presented in [Ang and Mathur 2023a].

Equipped with Lemma 5.3 and Lemma 5.5, we now present a constant-space streaming algorithm for Problem 5.2 in Algorithm 2, where we omit some details about how to maintain after sets and check partial local admissibility. A complete algorithm which combines Algorithm 1 and Algorithm 2 will be discussed in Section 5.4. We use a map M to track maximum partially admissible tuples with their labels as keys, and is initialized to be empty (line 1). The algorithm processes events in the execution one after another. When processing an event whose label is in $\langle a_1, \ldots, a_d \rangle$, we

Algorithm 3: Constant-Space Algorithm for Predictive Monitoring against Pattern Languages

Input: $\sigma \in \Sigma^*$
Output: YES if there exists a $\sigma' \equiv_{\mathcal{I}} \sigma$ such that $\mathsf{lab}(\sigma') \in \mathtt{Patt}_{a_1,\ldots,a_d}$; NO otherwise

1 Let (M: Permutations of prefixes of $a_1, \ldots, a_d \rightarrow$ Sequence of After Sets) be an empty map;
2 **for** $f \in \sigma$ **do**
3   **foreach** $\mathbb{A}_e \in M$ **do**   // Update after sets
4   **if** $\exists a \in \mathbb{A}_e$ *s.t.* $(a, \mathsf{lab}(f)) \in \mathcal{D}$ **then** $\mathbb{A}_e \leftarrow \mathbb{A}_e \cup \{\mathsf{lab}(f)\}$;
5   **if** $\mathsf{lab}(f) \in \langle a_1, \ldots, a_d \rangle$ **then**
6   **foreach** π *which is a permutation of a prefix of* $a_1, \ldots, a_d$ *ending with* $\mathsf{lab}(f)$ **do**
7   Let $\pi' = \pi[: -1]$;
8   **if** M *contains* π' **then**
9   Let $\mathbb{A}_f = \{\mathsf{lab}(f)\}$;
10   Let PartCandTuple $= M(\pi') \circ \mathbb{A}_f$;
11   Let PartCandTuple† = sort(PartCandTuple, π);
                // Check partial admissibility
12   **if** $\forall \mathbb{A}_e \in M(\pi')$, $\mathbb{A}_f$ *occurs after* $\mathbb{A}_e$ *in* PartCandTuple† *or* $\mathsf{lab}(f) \notin \mathbb{A}_e$ **then**
13                     $M(\pi) \leftarrow M(\pi') \circ \{\mathsf{lab}(f)\}$
14   **if** M *contains a key of length d* **then**
15   **return** *YES*;
16 **return** *NO*;

update the maximum partially admissible tuples we track if needed (line 3), as stated in Lemma 5.5. If a complete admissible tuple is constructed (line 8), which is of length k, the algorithm returns "YES" to claim the existence of admissible tuples. On the other hand, when finishing processing the whole execution and no complete admissible tuple is constructed (line 10), it returns "NO" to refute the existence.

5.4 Algorithm for Predictive Monitoring against Pattern Languages

We now present in Algorithm 3 the one-pass constant-space streaming algorithm for solving predictive monitoring against pattern languages, by combining Algorithm 1 and Algorithm 2. In this section, we first introduce the data structure used in the algorithm, then illustrate how the algorithm works. In particular, we will elaborate on how to check partial admissibility using Algorithm 1 and the data structure, which is omitted in Section 5.3.

The data structure is similar to what is used in Algorithm 2. First, recall that our algorithm tracks the maximum partially admissible tuple for each kind of label. So, for any processed execution ρ, we use a map to store key-value pairs, $(\langle b_1, \ldots, b_m \rangle, \max(\partial\mathsf{AdTpls}(\rho, \langle b_1, \ldots, b_m \rangle)))$, where $\langle b_1, \ldots, b_m \rangle$ is a permutation of a subsequence of $\langle a_1, \ldots, a_d \rangle$. The difference lies in that we keep track of the after set of each event in maximum partially admissible tuples, instead of events themselves. It is sufficient to do so because the only information we need to check admissibility is the label and the after set of an event. Notice that the label information is implicitly stored in the key of the map.

In Algorithm 3, we first initialize the map as empty (line 1). When processing an event f, we first update all after sets we track by accommodating f (line 3 - line 4). If $\mathsf{lab}(f)$ participates in $\langle a_1, \ldots, a_d \rangle$, we try to update partially admissible tuples based on Lemma 5.2 (line 5 - line 15). Event

f might participate in a maximum partially admissible tuples, whose labels end with $\mathsf{lab}(f)$. From line 8 to line 13, we check whether a partially admissible tuple extended with f is still admissible. The algorithm first constructs the after set for f, which contains only $\mathsf{lab}(f)$. Then it computes the target $\mathsf{PartCandTuple}^{\dagger}$ uniquely determined from a partial candidate tuple PartCandTuple, which is a concatenation of a previously computed maximum partially admissible tuples and f (Lemma 5.2). It finally uses Algorithm 1 to determine partial admissibility of PartCandTuple. This can be done because we have maintained the label and the after set of each event.

Finally, similar to Algorithm 2, when it witnesses a k-length complete admissible tuple, the algorithm claims the existence of a $\sigma' \equiv_{\mathcal{I}} \sigma$, such that $\mathsf{lab}(\sigma') \in \mathtt{Patt}_{a_1,\dots,a_d}$, and says "YES" to the predictive monitoring problem (line 14). Otherwise, after the whole execution is processed, it refutes the existence of such a reordering (line 16).

Fix $(\Sigma, \mathcal{I})$ and the pattern language L. The following two theorems summarize the correctness and the complexity of our Algorithm 3, where we assume that $d, |\Sigma| \in O(1)$. The proof of Theorem 5.1 is presented in [Ang and Mathur 2023a].

Theorem 5.1 (Correctness). *On input execution σ, Algorithm 3 declares "YES" iff there is a $\sigma' \equiv_{\mathcal{I}} \sigma$ such that $\sigma' \in L$.*

Theorem 5.2 (Complexity). *On input execution σ, Algorithm 3 runs in time $O(|\sigma|)$ and uses constant space.*

For a generalized pattern language L, we have the following corollary.

Corollary 5.1. *The problem of predictive trace monitoring against generalized pattern languages can be solved in linear time and constant space.*

5.5 Vector Clock Algorithm

In Algorithm 3, we store the after set for each event we track. Although the size of an after set is constant and bounded by Σ, it can become too large when $|\Sigma|$ is large. To overcome this problem, in this section, we propose the use of vector timestamps that can be used to efficiently check happens-before relation in the context of a concurrent execution [Fidge 1991; Mathur et al. 2022; Mattern 1989]. Our resulting timestamp-based algorithm will be referred to as PatternTrack.

We first provide a brief introduction to vector timestamps and vector clocks. Formally, a vector timestamp is a mapping $\mathsf{VC} : \mathcal{T} \to \mathbb{N}$ from the threads to natural numbers. Vector clocks are variables taking values from the space of vector timestamps. Three common operations on vector clocks are *join*, *comparison*, and *update*. We use $\mathsf{VC}_1 \sqcup \mathsf{VC}_2$ to denote the join of two vector timestamps VC_1 and VC_2, which is the vector timestamp $\lambda t, \max(\mathsf{VC}_1(t), \mathsf{VC}_2(t))$. The partial order $\sqsubseteq$ is defined so that $\mathsf{VC}_1 \sqsubseteq \mathsf{VC}_2$ iff $\forall t, \mathsf{VC}_1(t) \leq \mathsf{VC}_2(t)$. The minimum timestamp $\lambda t, 0$ is denoted by $\bot$. For $c \in \mathbb{N}$, the updated timestamp $\mathsf{VC}[t \to c]$ is $\lambda u,\ \mathtt{if}\ u = t\ \mathtt{then}\ c\ \mathtt{else}\ \mathsf{VC}(u)$. Our algorithm will assign a timestamp VC_e for each event e so that for each thread $t \in \mathcal{T}$, we have $\mathsf{VC}_e(t) = |\{f \mid f \leq^{\sigma}_{\mathcal{D}} e, \mathsf{lab}(f) = [t, op]\}|$. The following captures the relation between vector timestamps and the order $\leq^{\sigma}_{\mathcal{D}}$:

Lemma 5.6. *Let σ be an execution. Let e, f be two events in σ. $\mathsf{VC}_e \sqsubseteq \mathsf{VC}_f \iff e \leq^{\sigma}_{\mathcal{D}} f$.*

The aforementioned lemma guarantees that our algorithm can utilize vector clocks to perform the same task as long as we can compute the vector timestamp for each event in a streaming fashion and constant space. Algorithm 4 demonstrates how to compute vector timestamps by maintaining a vector clock for each label $[t, op] \in \Sigma$ and each $t \in \mathcal{T}$.

In Algorithm 4, we maintain a vector clock, for each label $a \in \Sigma$, to track the vector timestamp of the last event labeled with a in the prefix ρ seen so far. Additionally, we also maintain a vector

Algorithm 4: Constant-Space Streaming Algorithm for Computing Vector Timestamps

Input: $\sigma \in \Sigma^*$

1 Let $\mathsf{VC}_{[t,op]} = \bot$ for every $[t, op] \in \Sigma$;
2 Let $\mathsf{VC}_t = \bot$ for every $t \in \mathcal{T}$;
3 **for** $f \in \sigma$ **do**
4   Let $[t, op] = \mathsf{lab}(f)$;
5     $\mathsf{VC}_t \leftarrow \mathsf{VC}_t[t \rightarrow (\mathsf{VC}_t(t) + 1)]$;
6   **for** $[t', op'] \in \Sigma$ *such that* $([t, op], [t', op']) \notin \mathcal{I}$ **do**
7         $\mathsf{VC}_t \leftarrow \mathsf{VC}_t \sqcup \mathsf{VC}_{[t',op']}$
8     $\mathsf{VC}_{[t,op]} \leftarrow \mathsf{VC}_t$;
9     $\mathsf{VC}_f \leftarrow \mathsf{VC}_{[t,op]}$

clock to track the timestamp of the latest event of each thread. Initially, in line 1, all vector clocks for every label and thread are initialized to $\bot$. When processing an event f executed by thread t, the entry corresponding to t is first incremented in VC_t (line 5). Then VC_t is updated by joining it with all vector clocks corresponding to labels that are dependent on $\mathsf{lab}(f)$, since the last events with these labels happen before f (line 7). Finally, we update the clock for $\mathsf{lab}(f)$ with VC_t. The vector timestamp of f is precisely $\mathsf{VC}_{\mathsf{lab}(f)}$.

Therefore, obtaining a vector-clock version of Algorithm 3 is simple. We can replace initialization (Algorithm 3 line 1) and maintain (Algorithm 3 line 3) parts with the initialization of all vector clocks and the computation of $\mathsf{VC}_{\mathsf{lab}(f)}$ as in Algorithm 4. For checking partial admissibility, we first update VC_f to be $\mathsf{VC}_{\mathsf{lab}(f)}$ and compute PartCandTuple as $M(\pi') \circ \mathsf{VC}_f$ (Algorithm 3 line 9 and line 10). Next we modify the condition in Algorithm 3 line 12 as follows: "$\forall \mathsf{VC}_e \in M(\pi')$, VC_f occurs after VC_e in PartCandTuple† or $\mathsf{VC}_e \not\sqsubseteq \mathsf{VC}_f$". Notice that we use the $\sqsubseteq$ relation between vector timestamps to determine happens-before relation.

Fix $(\Sigma, \mathcal{I})$ and the pattern language L. In the following, we assume that every arithmetic operation takes $O(1)$ time and $d, |\mathcal{T}|, |\Sigma| \in O(1)$.

Theorem 5.3 (Correctness)**.** On input execution σ, PatternTrack declares "YES" iff there is a $\sigma' \equiv_{\mathcal{I}} \sigma$ such that $\sigma' \in L$.

Theorem 5.4 (Complexity)**.** On input execution σ, PatternTrack runs in time $O(|\sigma|)$ and uses constant space.

6 IMPLEMENTATION AND EVALUATION

In this section, we discuss the implementation and evaluation of our vector clock algorithm, which we call PatternTrack, and the algorithm due to [Bertoni et al. 1989], which we call Bertoni. The goal of our evaluation is two-fold. First, we want to understand whether the framework of pattern languages can express high-level temporal properties beyond traditional concurrency bugs, such as data races and atomicity violations. Further, we want to demonstrate whether the algorithm we propose can effectively expose those violations. Our second objective is to understand the performance improvement of our linear-time algorithm PatternTrack over the classical algorithm Bertoni and show the scalability of our algorithm. For this, we collected benchmarks from the Java Grande forum benchmark suite [Smith et al. 2001], the DaCapo benchmark suite [Blackburn et al. 2006], as well as open-source GitHub projects used in prior work [Legunsen et al. 2016].

6.1 Experimental Setup

Algorithm Implementation. We implemented the vector clock version of Algorithm 3 and the classical algorithm from [Bertoni et al. 1989] in a public prototype tool RAPID [Mathur 2023], written in Java. Both algorithms have been implemented in a streaming fashion, in that, they process each event as soon as it is observed. To ensure a fair comparison, we input the same executions to both algorithms. For this, we first generated one or more execution logs in each benchmark (a total of 33 executions) using the instrumentation and logging facility provided by RoadRunner [Flanagan and Freund 2010], and then, used the generated trace logs instead of performing monitoring at runtime. The events of the logs are related to a full program which may span many classes during the execution of a program. For native operations like read and write, the instrumentation resolves references precisely, while for the calls to external APIs, we used manual annotations to print the addresses of the objects.

Time Reporting. Comparing the two algorithms in a streaming way and with the same termination criteria ensures the fairness and correctness of the comparison. Both algorithms return "YES" as soon as they witness that a processed prefix can be reordered as a member of the pattern language. Thus, the length of execution processed by both algorithms is always the same, either the same prefix or the whole execution. This approach to the comparison ensures that any differences in performance are due to algorithmic differences rather than variations in the amount of data processed. In addition, we have imposed a 3-hour timeout (TO). Moreover, the maximum heap size of the Java Virtual Machine is set to 256 GB, which puts a limit on the amount of memory that can be used during execution (OOM).

Machine Configuration. Our experiments were conducted on a 1996.250 MHz 64-bit Linux machine with Java 19 and 256 GB heap space.

6.2 Bug Finding

We demonstrate the effectiveness of the class of pattern languages in encoding concurrency properties, and of our algorithm to predict violations against these specifications using five Java GitHub projects derived from [Legunsen et al. 2016]: `logstash-logback-encoder` [log 2023], `exp4j` [exp 2023], `jfreechart` [jfr 2023], `zmq-appender` [zer 2023], `antlrworks` [ant 2023].

We examined five high-level properties that we will briefly describe. The first property ensures consistency between two fields of a class. The last four come from Java API specifications [Legunsen et al. 2016]. A violation of these specifications would result in runtime errors. In the following, we describe these properties briefly and present how to encode them as pattern languages.

- **`Class_Invariant`.** A common requirement in many object-oriented software designs is to ensure that two (or, in general, more) fields of a given class are in a consistent state after every method call. Consider two fields a,b and also two methods `f1`,`f2`, both of which write to both a and b. The design specification of such a class asks that, in every execution, calls to `f1` and `f2` behave atomically. In other words, an interleaving of `f1` and `f2` might leave the values in an inconsistent state. We can encode one of the violations as the following pattern language: $\texttt{Patt}_{\texttt{f1.write(a), f2.write(a), f2.write(b), f1.write(b)}}$.
- **`Buffer_ManipulateAfterClose`.** This property asks whether there is an access (write) to a buffer that follows a call to the `close()` method on the buffer. We can encode the violation of this property as the following pattern language: $\texttt{Patt}_{\texttt{buf.close(), buf.write()}}$.
- **`UnsafeIterator`.** The property `Collection_UnsafeIterator` checks if an execution modifies a collection while iterating it. We encode the violation as $\texttt{Patt}_{\texttt{iter.next(), c.add(), iter.next()}}$,

Table 1. Experimental Results on Violation Prediction: Columns 1 - 3 present the benchmark name, number of processed events, and number of threads. Column 4 shows the name of the violation that our algorithm predicts. Columns 5 and 6 report the processing time of PatternTrack and Bertoni.

Program	$\mathcal{N}$	$\mathcal{T}$	Property Violation	PatternTrack	Bertoni
logstash-logback-encoder	587	3	`Buffer_ManipulateAfterClose`	23ms	64ms
jfreechart	2753	3	`Collection_UnsafeIterator`	63ms	683ms
zmq-appender	10742	8	`Map_UnsafeIterator`	168ms	OOM
exp4j	1420	3	`Collection_UnsynchronizedAddAll`	56ms	171ms
antlrworks	1027	3	`Class_Invariant`	76ms	379ms

```java
class JFreeChart{
  public LegendTitle getLegend(int index) {
    int seen = 0;
    Iterator iterator = this.subtitles.iterator();
    while (iterator.hasNext()) {
      Title subtitle = (Title) iterator.next();
      ...
      /* get title */
      ...
    }
    return null;
  }

  public void addSubtitle(Title subtitle) {
    ...
    this.subtitles.add(subtitle);
    ...
  }
}
```

(a) `jfreechart`

```java
class ReusableByteBuffer extends OutputStream {
  @Override
  public void close() {
    this.closed = true;
  }

  @Override
  public void write(byte[] data ...) {
    ...
    if (this.closed) {
      /* throw IOException */
    }
    while (length > 0) {
      ...
      /* write to buffer */
      ...
    }
  }
}
```

(b) `logstash`

Fig. 3. Code snippets. Violation of `Collection_UnsafeIterator` in `jfreechart`. Violation of `Buffer_ManipulateAfterClose` in `logstash`.

where `c` is a collection and `iter` is one of its iterators. The property `Map_UnsafeIterator` is almost the same except that `iter` is one of the iterators of `c.entrySet()`.

`Collection_UnsynchronizedAddAll`. This property checks whether a call to `addAll()` interleaves with another modification, such as `add()`, to the same collection. Here, we encode the violation of this property as $\texttt{Patt}_{\texttt{c.addAll()Enter},\, \texttt{c.add()},\, \texttt{c.addAll()Exit}}$, where `c.addAll()Enter` and `c.addAll()Exit` are events corresponding to the invocation and the return of this method respectively.

We executed multi-threaded test cases of the aforementioned 5 GitHub Java projects and logged the executions. We ensure that these executions do not contain any pattern violations themselves either by ensuring that no runtime exception is thrown or by manual inspection. Nevertheless, as shown in Table 1, our algorithm is able to predict violations in these executions, indicating the ability of PatternTrack to find real-world bugs. Table 1 also demonstrates the performance improvement of PatternTrack over Bertoni. In the execution of `zmq-appender`, we observed that Bertoni throws an out-of-memory exception. Recall that the space complexity of Bertoni is $O(\mathcal{N}^{|\mathcal{T}|})$ (since the width is $\alpha = |\mathcal{T}|$), which is large when the number of processed events and the number of threads is even moderately large, and this is evident in its memory usage.

Let us highlight some of the snippets from the source code of these benchmarks and patterns exhibited by them that lead to violations of the mentioned properties. First, Figure 3a shows a real bug violating the property `Collection_UnsafeIterator` in `JFreeChart`. When two different threads call the methods `getLegend` and `addSubtitle` concurrently, a `ConcurrentModification`

Table 2. Experimental results on patterns of dimension 3: Columns 1 - 3 contain benchmark information, including name, number of events, and number of threads. Columns 4 - 13 report the time spent until the first instance of an admissible tuple is found, or the time to process the entire execution (if there is no match) of PatternTrack and Bertoni on five different 3-dimension patterns. It is important to note that the patterns in the same column are different for different benchmarks. For the sake of clarity and ease of presentation, we have named the five patterns as Pattern 1-5.

1	2	3	4	5	6	7	8	9	10	11	12	13
Program	$\mathcal{N}$	$\mathcal{T}$	Pattern 1		Pattern 2		Pattern 3		Pattern 4		Pattern 5	
			P.Tk	B.ni	P.Tk	B.ni	P.Tk	B.ni	P.Tk	B.ni	P.Tk	B.ni
exp4j-1	1.0K	2	0.03s	1.80s	0.03s	0.66s	0.04s	1.10s	0.03s	0.33s	0.03s	0.16s
crawler-1	1.8K	2	0.04s	0.18s	0.07s	0.37s	0.05s	0.46s	0.04s	0.38s	0.05s	0.32s
crawler-2	2.0K	2	0.06s	0.51s	0.06s	0.23s	0.06s	0.19s	0.03s	0.23s	0.06s	0.38s
junit4-1	3.1K	3	0.07s	0.10s	0.08s	2m27s	0.07s	40.19s	0.06s	11m43s	0.10s	OOM
junit4-2	5.5K	101	0.05s	TO	0.04s	TO	0.06s	TO	0.04s	TO	0.04s	TO
junit4-3	5.5K	101	0.05s	TO	0.07s	TO	0.04s	TO	0.05s	TO	0.05s	TO
junit4-4	25K	501	0.09s	TO	0.08s	TO	0.09s	TO	0.11s	TO	0.11s	TO
metafacture	195K	3	0.83s	4.76s	0.98s	19.27s	0.92s	24.18s	1.04s	4.30s	1.17s	22.51s
logstash-1	371K	11	0.19s	OOM	0.14s	OOM	0.13s	OOM	0.14s	OOM	0.14s	OOM
logstash-2	446K	3	0.02s	3.31s	0.02s	2.40s	0.02s	3.22s	1.69s	OOM	1.68s	OOM
cassandra-1	1.3M	4	0.10s	0.52s	0.08s	1.42s	0.08s	1.61s	1.78s	OOM	1.75s	OOM
cassandra-2	1.5M	4	3.04s	OOM	0.09s	OOM	0.07s	OOM	3.50s	OOM	0.07s	OOM
cassandra-3	1.6M	3	3.66s	OOM	0.34s	26.57s	3.44s	OOM	3.26s	OOM	3.22s	OOM
cassandra-4	1.6M	3	0.30s	36.48s	0.32s	1.67s	0.35s	1.71s	3.38s	OOM	0.31s	34.68s
cassandra-5	1.6M	3	6.14s	OOM	0.51s	2.11s	0.49s	1.46s	3.77s	OOM	4.33s	OOM
exp4j-2	1.9M	11	0.05s	3.88s	0.04s	29.93s	0.04s	1.50s	0.04s	2.91s	0.05s	8m48s
zookeeper-1	2.2M	18	0.30s	OOM	0.25s	OOM	0.28s	OOM	0.31s	8m41s	0.39s	OOM
tomcat	2.6M	11	5.39s	OOM	5.48s	OOM	5.41s	OOM	5.44s	OOM	5.31s	OOM
zookeeper-2	5.6M	10	0.76s	OOM	0.28s	OOM	0.30s	OOM	14.48s	OOM	0.32s	OOM
zookeeper-3	7.5M	18	0.29s	TO	0.43s	TO	0.21s	TO	0.20s	TO	0.33s	TO
zookeeper-4	8.0M	10	0.09s	0.37s	0.22s	OOM	0.18s	OOM	0.20s	OOM	0.13s	6.62s
batik	95M	4	1m34s	OOM	1m42s	OOM	1m34s	OOM	1m47s	OOM	1m8s	OOM
xalan	248M	3	3.97s	OOM	4.97s	OOM	4.97s	OOM	3.39s	OOM	6.09s	OOM
sunflow	262M	5	25.41s	OOM	25.83s	OOM	26.22s	OOM	26.04s	OOM	25.91s	OOM
lusearch	325M	4	0.76s	OOM	0.40s	OOM	0.98s	OOM	2m2s	OOM	0.48s	OOM
moldyn	401M	4	14.21s	OOM	14.90s	OOM	14.80s	OOM	14.64s	OOM	14.55s	OOM
series	509M	4	0.09s	OOM	0.62s	OOM	0.17s	OOM	0.07s	OOM	0.12s	OOM
montecarlo	530M	4	0.47s	0.83s	1.38s	OOM	1.32s	OOM	1.25s	OOM	1.31s	OOM
raytracer	545M	4	2m45s	OOM	2m36s	OOM	2m34s	OOM	2m33s	OOM	2m34s	OOM
sparsematmult	602M	4	7m45s	OOM	7m57s	OOM	7m48s	OOM	7m55s	OOM	7m55s	OOM
lufact	616M	4	1m29s	OOM	1m26s	OOM	57.77s	OOM	2m26s	OOM	3m33s	OOM
sor	642M	4	1m39s	OOM	1m42s	OOM	1m40s	OOM	18.92s	43.08s	1m0s	OOM
avrora	739M	8	19.09s	OOM	31.29s	OOM	48.93s	OOM	52.88s	OOM	36.23s	OOM
Mean	168M	-	30.29s	2h0m	30.75s	2h0m	29.86s	2h5m	35.19s	2h17m	32.59s	2h22m

error might be thrown. This is because the call to `add()` in `addSubtitle` might interleave between two `next()` calls. Next, consider the code snippet in Figure 3b derived from `logstash`. Here, `Buffer_ManipulateAfterClose` property might be violated. Even though the `write()` method checks `this.closed` before writing to the buffer, it is still possible that setting `this.closed` in `close()` occurs between the checking (line 10) and the writing (lines 13 - 17). The author of this project has claimed that this class is not thread-safe. The last example of `antlrworks` has been presented as the motivating example in Example 2.1. In conclusion, pattern languages are expressive enough to encode simple and intuitive real-world high-level correctness specifications, and our algorithm can predict violations of these effectively.

6.3 Performance Evaluation

The goal of this section is to evaluate the performance of PatternTrack over large execution traces and a variety of patterns. We first discuss how we obtained the patterns we monitor against. Next,

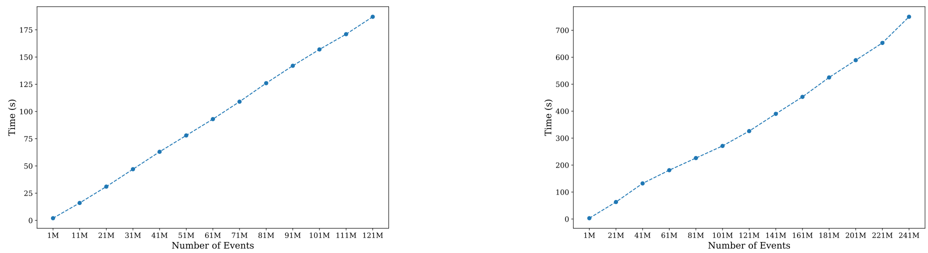
Table 3. Experimental results on patterns of dimension 5. Columns 1 - 3 contain benchmark information, including name, number of events, and number of threads. Columns 4 - 13 report the time spent until the first instance of an admissible tuple is found, or the time to process the entire execution (if there is no match) of PatternTrack and Bertoni on five different patterns.

1	2	3	4	5	6	7	8	9	10	11	12	13
Program	$\mathcal{N}$	$\mathcal{T}$	Pattern 1		Pattern 2		Pattern 3		Pattern 4		Pattern 5	
			P.Tk	B.ni	P.Tk	B.ni	P.Tk	B.ni	P.Tk	B.ni	P.Tk	B.ni
exp4j-1	1.0K	2	0.04s	1.62s	0.04s	1.15s	0.05s	1.17s	0.04s	1.56s	0.04s	0.36s
crawler-1	1.8K	2	0.07s	0.33s	0.06s	0.34s	0.05s	0.46s	0.05s	0.33s	0.06s	0.44s
crawler-2	2.0K	2	0.08s	0.49s	0.10s	0.43s	0.05s	0.48s	0.07s	0.48s	0.08s	0.45s
junit4-1	3.1K	3	0.09s	5m5s	0.13s	OOM	0.08s	OOM	0.09s	OOM	0.10s	OOM
junit4-2	5.5K	101	0.06s	TO	0.04s	TO	0.05s	TO	0.05s	TO	0.06s	TO
junit4-3	5.5K	101	0.06s	TO	0.06s	TO	0.05s	TO	0.06s	TO	0.06s	TO
junit4-4	25K	501	0.10s	TO	0.09s	TO	0.16s	TO	0.12s	TO	0.09s	TO
metafacture	195K	3	1.19s	22.75s	1.01s	19.18s	0.91s	19.17s	1.12s	22.53s	0.92s	22.98s
logstash-1	371K	11	0.26s	OOM	0.30s	OOM	0.28s	OOM	0.28s	OOM	0.40s	OOM
logstash-2	446K	3	1.83s	OOM	0.03s	2.45s	2.11s	OOM	0.02s	3.18s	2.08s	OOM
cassandra-1	1.3M	4	1.99s	OOM	1.99s	OOM	1.92s	OOM	2.05s	OOM	0.08s	1.24s
cassandra-2	1.5M	4	49.05s	OOM	1m14s	OOM	15.41s	OOM	6.14s	OOM	9.50s	OOM
cassandra-3	1.6M	3	3.42s	OOM	4.11s	OOM	7.96s	OOM	4.56s	OOM	4.10s	OOM
cassandra-4	1.6M	3	3.62s	OOM	5.19s	OOM	4.06s	OOM	3.25s	OOM	3.90s	OOM
cassandra-5	1.6M	3	5.19s	OOM	20.42s	OOM	5.42s	OOM	4.20s	OOM	3.87s	OOM
exp4j-2	1.9M	11	0.04s	6m49s	0.05s	10m34s	0.06s	56.35s	0.07s	9m14s	0.09s	OOM
zookeeper-1	2.2M	18	6.02s	OOM	0.45s	OOM	0.36s	OOM	0.45s	OOM	0.33s	OOM
tomcat	2.6M	11	5.33s	OOM	5.35s	OOM	5.42s	OOM	5.43s	OOM	5.48s	OOM
zookeeper-2	5.6M	10	0.39s	19.50s	0.33s	OOM	18.48s	OOM	21.51s	OOM	28.82s	OOM
zookeeper-3	7.5M	18	0.34s	TO	0.21s	TO	0.23s	TO	1.77s	TO	0.58s	TO
zookeeper-4	8.0M	10	0.23s	OOM	0.29s	OOM	0.34s	OOM	0.27s	OOM	24.08s	OOM
batik	95M	4	2m27s	OOM	2m25s	OOM	2m27s	OOM	2m26s	OOM	2m25s	OOM
xalan	248M	3	3.59s	OOM	6.98s	OOM	4.71s	OOM	7.17s	OOM	8.14s	OOM
sunflow	262M	5	25.43s	OOM	27.92s	OOM	25.63s	OOM	25.36s	OOM	26.28s	OOM
lusearch	325M	4	0.75s	OOM	2m5s	OOM	1.57s	OOM	39.54s	OOM	3.65s	OOM
moldyn	401M	4	15.48s	OOM	15.61s	OOM	13.39s	OOM	16.44s	OOM	15.25s	OOM
series	509M	4	0.14s	OOM	0.09s	OOM	0.14s	OOM	0.24s	OOM	0.12s	OOM
montecarlo	530M	4	2.12s	OOM	1.62s	OOM	1.75s	OOM	1.15s	OOM	1.54s	OOM
raytracer	545M	4	2m36s	OOM	2m41s	OOM	3m49s	OOM	51m11s	OOM	47m11s	OOM
sparsematmult	602M	4	7m43s	OOM	7m50s	OOM	7m52s	OOM	7m47s	OOM	7m51s	OOM
lufact	616M	4	3m45s	OOM	58.28s	OOM	1m31s	OOM	3m19s	OOM	1m6s	OOM
sor	642M	4	1m43s	OOM	2m1s	OOM	52.96s	OOM	1m53s	OOM	52.18s	OOM
avrora	739M	8	1m57s	OOM	3m20s	OOM	42.06s	OOM	1m28s	OOM	53.27s	OOM
Mean	168M	-	40.66s	2h22m	43.97s	2h27m	34.74s	2h32m	2m8s	2h27m	1m53s	2h32m

we evaluate both PatternTrack as well Bertoni [Bertoni et al. 1989] against these specifications and compare their performance. Finally, we investigate the impact of several parameters that may affect the practical performance of our algorithm. Specifically, we examine how the performance of our algorithm evolves with changes in the length of executions and the number of threads.

Pattern Generation. In order to ensure that the comparison is sufficient and fair, we generated pattern languages of dimension $d = 5$ and $d = 3$ at random by sampling events in each execution log. The choice of $d = 5$ and $d = 3$ is in line with prior empirical observations that most concurrent bugs in real-world scenarios can be observed by the ordering of a small number of events [Burckhardt et al. 2010]. In our experiment, our pattern is a sequence of program locations $\langle l_1, \dots, l_d \rangle$, instead of labels. Each program location can be viewed as a set of program labels, so we evaluate the membership problem against a generalized pattern language $L = \bigcup_{a_i^1 \in l_1, \dots, a_i^d \in l_d} \mathtt{Patt}_{a_i^1, \dots, a_i^d}$.

To generate a pattern, we randomly chose 5 or 3 events and logged their program locations. We employed two policies for selecting events. The first policy we utilized is locality. We divided each execution into 100 parts of equal length and randomly selected one of those parts as the



(a) raytracer (b) sparsematmult

Fig. 4. Scalability with number of events in the executions

source of one pattern. Events chosen from widely separated parts of the execution are less likely to participate in a bug-inducing pattern. The second policy is diversity. We chose events from as many different threads as possible, as this leads to more concurrency.

Speedup over [Bertoni et al. 1989]. Theoretical analysis indicates that our algorithm runs in $O(|\sigma|)$ time, while the algorithm in [Bertoni et al. 1989] runs in $O(|\sigma|^{\alpha})$ time. As shown in Table 2 and Table 3, our algorithm PatternTrack exhibits significant performance advantages over Bertoni in practice. Additionally, Bertoni is not scalable when the length of the execution increases, as evidenced by the numerous instances of "OOM" (Out of Memory) cases in the tables. This is because Bertoni needs to store all ideals of a partial order, which consumes $O(|\sigma|^{\alpha})$ space. As a result, this limits its scalability only to short executions. In contrast, our constant-space algorithm PatternTrack, catered to pattern regular languages, is not burdened by this limitation and can handle executions with millions of events.

Performance w.r.t. execution length. We next investigate whether the theoretical linear-time complexity of our algorithm also translates empirically. We selected two executions, from programs raytracer and sparsematmult, based on the fact that their length and the time taken to witness the first pattern match are both large. This selection criterion ensures that we can gather sufficient data to evaluate the performance across a range of execution lengths. To accomplish this, we recorded the time taken by our algorithm to process every 10 million events for raytracer and every 20 million events for sparsematmult. Figure 4 presents the result, and is in alignment with the theoretical linear running time of PatternTrack.

Performance w.r.t number of threads. The time taken by each vector clock operation is proportional to the number of threads, and we expect that PatternTrack's running time is also proportional to this number. To test this assumption, we generated seven executions from benchmark raytracer with 5, 10, 15, 20, 25, and 35 threads respectively. The raytracer application allows us to configure the number of threads when generating executions. We then recorded the time taken by PatternTrack to process $1,000,000$ events for each execution. The results, in Figure 5, show that the time consumption increases as the number of threads increases, in line with PatternTrack's theoretical complexity.

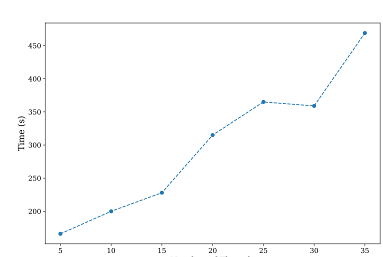


Fig. 5. Performance over #threads

7 RELATED WORK

Runtime verification has emerged as a popular class of techniques instrumental in popularizing the adoption of lightweight formal methods in large-scale industrial settings. Several classes of specifications such as LTL [Rosu and Havelund 2005], extended regular expressions [Sen and Rosu 2003], and CFG [Meredith et al. 2010] and techniques to monitor the runtime behaviour of the underlying software in conjunction with these specifications have been proposed. Aspect-oriented monitoring frameworks have been developed to bridge the gap between formal algorithms and practical tools. These include Java-MaC [Kim et al. 2004] and JavaMOP [Jin et al. 2012]. However, such techniques are catered towards the problem of the "detection" of violations of these specifications at runtime, in contrast to the "prediction" problem, which we address in our work when the underlying software is concurrent or multi-threaded.

Predictive techniques have recently emerged as a fundamental paradigm to enhance the efficacy of runtime verification techniques — instead of focusing on bugs in the observed sequence of events, such techniques attempt to find bugs in alternate correct reorderings. Most works in the runtime predictive analysis are centered around the detection of implicit concurrency-centric specifications such as the presence of data races, deadlocks, or violations of atomicity. Some of the early works that studied predictive analysis for data races are instrumental in formulating core ideas such as sound and maximal causal models and correct reorderings [Chen and Rosu 2007; Chen et al. 2008; Huang et al. 2015, 2014; Said et al. 2011; Sen et al. 2005, 2006; Serbanuta et al. 2012]. Many of these techniques, however, primarily employ heavy-weight approaches such as exhaustive enumeration or the use of SAT/SMT solvers, limiting their scalability in practice. The focus of recent works in predictive analysis tools [Cai et al. 2021; Kini et al. 2017; Mathur et al. 2018, 2021; Pavlogiannis 2019; Roemer and Bond 2019; Roemer et al. 2020] has been on efficiency and practical scalability. Such techniques develop carefully crafted algorithms relying on specific heuristics or partial orders and avoid heavy-weight SMT solving. Beyond data races, recent improvements in these techniques have been used for predicting other concurrency bugs such as deadlocks [Cai et al. 2021; Kalhauge and Palsberg 2018; Mathur et al. 2018; Sorrentino et al. 2010; Tunç et al. 2023]. Some works also propose predictive methods for higher level, but specific properties such as use-after-free violations [Huang 2018], null-pointer dereferences [Farzan et al. 2012], and violations of atomicity specifications [Biswas et al. 2014; Farzan and Madhusudan 2008, 2009; Flanagan et al. 2008a; Mathur and Viswanathan 2020; Sinha et al. 2011]. Our work is a step towards adapting predictive techniques to explicitly defined specifications. GPredict [Huang et al. 2015] proposes the use of SMT solving for generalized predictive monitoring over explicitly defined specifications (under the maximal causality model) but has limited scalability.

We remark that for the case of implicitly defined bugs such as data races, the computational intractability [Gibbons and Korach 1997; Mathur et al. 2020] in prediction arises primarily because of the exhaustive space of correct reorderings considered in these techniques [Serbanuta et al. 2012; Smaragdakis et al. 2012]. Reasoning based on Mazurkiewicz traces, though less exhaustive, is in general more scalable. Indeed, the most widely adopted data race detection techniques crucially rely on the happens-before partial order [Flanagan and Freund 2009; Mathur et al. 2018; Poznianski and Schuster 2003] based on a Mazurkiewicz dependence relation, and typically run in linear time (assuming the alphabet is of constant size) or in low-degree polynomial time (when the alphabet is not constant [Kulkarni et al. 2021]). Our work has been inspired by this contrast in computational complexity of reasoning with Mazurkiewicz traces v/s correct reorderings. Unfortunately, though, our hardness result depicts that this insight does not trivially generalize to arbitrary properties beyond data races. Our work therefore proposes pattern languages to offer a good balance of expressiveness and algorithmic efficiency. Our central result essentially says that the *trace closure* of pattern languages is regular. A complete characterization of regular languages whose closure is regular is that of star-connected regular languages due to [Ochmański 1985]. However, this result does not immediately yield an algorithm to translate an arbitrary star-connected languages to an automaton that recognizes its trace closure. Languages generated by Alphabetic Pattern Constraints

(APC) [Bouajjani et al. 2001] are a subclass of star-connected language, are closed under trace closure, and are strictly more expressive than generalized pattern languages. However, an efficient vector clock algorithm for predictive monitoring against APCs is not known. Our work thus fills the gap between theoretical results and practical techniques by designing small-sized monitors for predictive monitoring against pattern languages. The use of after sets in our constant space algorithm has been inspired from work in data race detection [Kini et al. 2018].

Soundness (absence of false positives), although highly desirable, often comes at a cost of computational complexity. Towards this, many runtime predictive analysis techniques often forego soundness in favour of the simplicity of analysis, including the Eraser algorithm based on locksets [Savage et al. 1997] or works on deadlock prediction [Bensalem and Havelund 2005; Cai et al. 2020]. Some approaches perform post-hoc analysis to reduce false positives [Roemer et al. 2018, 2020]. Other approaches rely on re-executions to confirm the predicted bugs [Joshi et al. 2009; Sorrentino et al. 2010] to reduce false positives.

Concurrency bug detection has been an active area of research for several decades. Besides runtime verification and predictive monitoring, there are many other techniques for the analysis of concurrent programs. Model checking [Clarke et al. 1986] has emerged as a popular paradigm for finding bugs, thanks to advances such as dynamic partial order reduction [Flanagan and Godefroid 2005] and stateless model checking [Abdulla et al. 2014; Kokologiannakis et al. 2022, 2019; Oberhauser et al. 2021]. Randomized testing techniques [Burckhardt et al. 2010; Joshi et al. 2009; Ozkan et al. 2019; Yuan et al. 2018] as well as fuzz testing techniques [Jeong et al. 2019] have also been shown effective in practice. Static analysis techniques [Blackshear et al. 2018; Engler and Ashcraft 2003; Naik et al. 2006; Voung et al. 2007] have been developed, but their adoption is often limited by high false positive rates. Type systems for preventing data races [Boyapati et al. 2002; Flanagan et al. 2008b; Flanagan and Qadeer 2003] have been instrumental in the design of programming languages such as Rust.

8 CONCLUSIONS

In this work, we study the predictive monitoring problem — given an execution of a concurrent program, can it be reordered to witness the violation of a specification? We show that for specifications expressed using regular languages, and when the reorderings are restricted to the *trace equivalence* class of the observed execution, this problem suffers a high complexity polynomial lower bound. Towards this, we propose a sub-class of regular languages, called (generalized) pattern languages, and show that this class of languages can be effectively monitored, in a predictive sense, using a constant-space linear-time algorithm. Our experimental evaluation, using an implementation of the algorithm PatternTrack we develop, shows the effectiveness of our proposed class of specification languages and the algorithms for (predictive) monitoring against them.

There are many avenues for future work. We expect pattern languages to be useful in controlled concurrency testing techniques [Agarwal et al. 2021; Musuvathi and Qadeer 2006] and fuzz testing for concurrent software [Jeong et al. 2019]. Another interesting direction is to generalize the algorithms developed here to a larger class of temporal specifications, such as APCs [Bouajjani et al. 2001], which remain easier to write, are developer-friendly, and house the potential of widespread adoption. Building optimal stateless model checking [Kokologiannakis et al. 2022] algorithms for richer specifications such as pattern languages is another interesting direction.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for several comments that helped improve the paper. We thank Vladimir Gladshtein and Martin Mirchev for initial discussions, and Georg Zetzsche for pointing us to the connection with the work of Edward Ochmański on star-connected languages. This work was partially supported by a Singapore Ministry of Education (MoE) Academic Research Fund (AcRF) Tier 1 grant.

DATA AVAILABILITY STATEMENT

The artefact for this work is available [Ang and Mathur 2023b], which contains source code and benchmarks to reproduce our evaluation in Section 6.

REFERENCES

2023. AntLRWorks. https://github.com/antlr/antlrworks. [Online; accessed 24-October-2023].

2023. exp4j. https://github.com/fasseg/exp4j. [Online; accessed 24-October-2023].

2023. JFreeChart. https://github.com/jfree/jfreechart. [Online; accessed 24-October-2023].

2023. Logstash Logback Encoder. https://github.com/logfellow/logstash-logback-encoder. [Online; accessed 24-October-2023].

2023. ZeroMQ log4j appender. https://github.com/lusis/zmq-appender. [Online; accessed 24-October-2023].

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

Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of Programs under the Release-Acquire Semantics. 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, 1117–1132. https://doi.org/10.1145/3314221.3314649

Udit Agarwal, Pantazis Deligiannis, Cheng Huang, Kumseok Jung, Akash Lal, Immad Naseer, Matthew Parkinson, Arun Thangamani, Jyothi Vedurada, and Yunpeng Xiao. 2021. Nekara: Generalised Concurrency Testing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 679–691. https://doi.org/10.1109/ASE51524.2021.9678838

Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining Sequential Patterns. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, Philip S. Yu and Arbee L. P. Chen (Eds.). IEEE Computer Society, 3–14. https://doi.org/10.1109/ICDE.1995.380415

Rajeev Alur, Kousha Etessami, and P. Madhusudan. 2004. A Temporal Logic of Nested Calls and Returns. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 467–481. https://doi.org/10.1007/978-3-540-24730-2_35

Zhendong Ang and Umang Mathur. 2023a. Predictive Monitoring against Pattern Regular Languages. arXiv:2310.14611 https://arxiv.org/abs/2310.14611

Zhendong Ang and Umang Mathur. 2023b. *Predictive Monitoring against Pattern Regular Languages.* https://doi.org/10.5281/zenodo.8424626 Artifact.

Cyrille Artho, Klaus Havelund, and Armin Biere. 2003. High-level data races. *Softw. Test. Verification Reliab.* 13, 4 (2003), 207–227. https://doi.org/10.1002/STVR.281

Saddek Bensalem and Klaus Havelund. 2005. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3875)*, Shmuel Ur, Eyal Bin, and Yaron Wolfsthal (Eds.). Springer, 208–223. https://doi.org/10.1007/11678779_15

A. Bertoni, G. Mauri, and N. Sabadini. 1989. Membership problems for regular and context-free trace languages. *Information and Computation* 82, 2 (1989), 135–150. https://doi.org/10.1016/0890-5401(89)90051-5

Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: efficient sound and precise atomicity checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 28–39. https://doi.org/10.1145/2594291.2594323

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

Ahmed Bouajjani, Anca Muscholl, and Tayssir Touili. 2001. Permutation Rewriting and Algorithmic Verification. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 399–408. https://doi.org/10.1109/LICS.2001.932515

Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 211–230. https://doi.org/10.1145/582419.582440

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

Yan Cai, Ruijie Meng, and Jens Palsberg. 2020. Low-overhead deadlock prediction. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1298–1309. https://doi.org/10.1145/3377811.3380367

Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and efficient concurrency bug prediction. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 255–267. https://doi.org/10.1145/3468264.3468549

Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. 2009. The Complexity of Satisfiability of Small Depth Circuits. In *Parameterized and Exact Computation: 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 75–85. https://doi.org/10.1007/978-3-642-11269-0_6

Feng Chen and Grigore Rosu. 2007. Parametric and Sliced Causality. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 240–253. https://doi.org/10.1007/978-3-540-73368-3_27

Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. 2008. jPredictor: a predictive runtime analysis tool for java. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 221–230. https://doi.org/10.1145/1368088.1368119

Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting Families of Schedules for Asynchronous Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 157–176. https://doi.org/10.1007/978-3-319-41540-6_9

Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (1986), 244–263. https://doi.org/10.1145/5397.5399

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-Bounded Scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 411–422. https://doi.org/10.1145/1926385.1926432

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

Azadeh Farzan and P. Madhusudan. 2008. Monitoring Atomicity in Concurrent Programs. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5123)*, Aarti Gupta and Sharad Malik (Eds.). Springer, 52–65. https://doi.org/10.1007/978-3-540-70545-1_8

Azadeh Farzan and P. Madhusudan. 2009. The Complexity of Predicting Atomicity Violations. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5505)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, 155–169. https://doi.org/10.1007/978-3-642-00768-2_14

Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting null-pointer dereferences in concurrent programs. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 47. https://doi.org/10.1145/2393596.2393651

Colin Fidge. 1991. Logical Time in Distributed Computing Systems. *Computer* 24, 8 (Aug. 1991), 28–33. https://doi.org/10.1109/2.84874

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)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008b. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.* 30, 4 (2008), 20:1–20:53. https://doi.org/10.1145/1377492.1377495

Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008a. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 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

Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 338–349. https://doi.org/10.1145/781131.781169

Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 603–616. https://doi.org/10.1145/3575693.3575729

Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, Francesca Rossi (Ed.). IJCAI/AAAI, 854–860. https://doi.org/10.5555/2540128.2540252

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

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Jeff Huang. 2018. UFO: predictive concurrency use-after-free detection. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 609–619. https://doi.org/10.1145/3180155.3180225

Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic Predictive Concurrency Analysis. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 847–857. https://doi.org/10.1109/ICSE.2015.96

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)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

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

Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 1427–1430. https://doi.org/10.1109/ICSE.2012.6227231

Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 110–120. https://doi.org/10.1145/1542476.1542489

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

Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 582–598. https://doi.org/10.1145/3132747.3132767

Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. 2004. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods Syst. Des.* 24, 2 (2004), 129–155. https://doi.org/10.1023/B:FORM.0000017719.43755.7C

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)*. Association for Computing Machinery, 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

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, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 96–110. https://doi.org/10.1145/3314221.3314609

Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real Time Syst.* 2, 4 (1990), 255–299. https://doi.org/10.1007/BF01995674

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, August 24-27, 2021, Virtual Conference (LIPIcs, Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:23. https://doi.org/10.4230/LIPICS.CONCUR.2021.16

Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Rosu, and Darko Marinov. 2016. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 602–613. https://doi.org/10.1145/2970276.2970356

Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3253)*, Yassine Lakhnech and Sergio Yovine (Eds.). Springer, 152–166. https://doi.org/10.1007/978-3-540-30206-3_12

Umang Mathur. 2023. *RAPID*. https://github.com/umangm/rapid Accessed: 2023-10-25.

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, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 710–725. https://doi.org/10.1145/3503222.3507734

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

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 *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 183–199. https://doi.org/10.1145/3373376.3378475

Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, M. Cosnard et. al. (Ed.). Elsevier Science Publishers B. V., 215–226.

Antoni Mazurkiewicz. 1987. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–324.

Robert McNaughton and Seymour Papert. 1971. Counter-Free Automata.

Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Rosu. 2010. Efficient monitoring of parametric context-free patterns. *Autom. Softw. Eng.* 17, 2 (2010), 149–180. https://doi.org/10.1007/S10515-010-0063-Y

Vijayaraghavan Murali, Edward Yao, Umang Mathur, and Satish Chandra. 2021. Scalable Statistical Root Cause Analysis on App Telemetry. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in*

Practice (Virtual Event, Spain) *(ICSE-SEIP '21)*. IEEE Press, 288–297. https://doi.org/10.1109/ICSE-SEIP52600.2021.00038

Madan Musuvathi and Shaz Qadeer. 2006. CHESS: Systematic Stress Testing of Concurrent Software. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4407)*, Germán Puebla (Ed.). Springer, 15–16. https://doi.org/10.1007/978-3-540-71410-1_2

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 530–545. https://doi.org/10.1145/3445814.3446748

Edward Ochmański. 1985. Regular behaviour of concurrent systems. *Bull. EATCS* 27 (1985), 56–67.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 180:1–180:29. https://doi.org/10.1145/3360606

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

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, USA, 46–57. https://doi.org/10.1109/SFCS.1977.32

Eli Poznianski and Assaf Schuster. 2003. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 287. https://doi.org/10.1109/IPDPS.2003.1213513

Jake Roemer and Michael D. Bond. 2019. Online Set-Based Dynamic Analysis for Sound Predictive Race Detection. *CoRR* abs/1907.08337 (2019). arXiv:1907.08337 http://arxiv.org/abs/1907.08337

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, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 374–389. https://doi.org/10.1145/3192366.3192385

Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 747–762. https://doi.org/10.1145/3385412.3385993

Grigore Rosu and Klaus Havelund. 2005. Rewriting-Based Techniques for Runtime Verification. *Autom. Softw. Eng.* 12, 2 (2005), 151–197. https://doi.org/10.1007/S10515-005-6205-Y

Grigore Rosu and Mahesh Viswanathan. 2003. Testing Extended Regular Language Membership Incrementally by Rewriting. In *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2706)*, Robert Nieuwenhuis (Ed.). Springer, 499–514. https://doi.org/10.1007/3-540-44881-0_35

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (mar 2018), 58–66. https://doi.org/10.1145/3188720

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. 2011. Generating Data Race Witnesses by an SMT-Based Analysis. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 313–327. https://doi.org/10.1007/978-3-642-20398-5_23

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411. https://doi.org/10.1145/265924.265927

M.P. Schützenberger. 1965. On finite monoids having only trivial subgroups. *Information and Control* 8, 2 (1965), 190–194. https://doi.org/10.1016/S0019-9958(65)90108-7

Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems* (Athens, Greece) *(FMOODS'05)*. Springer-Verlag, Berlin, Heidelberg, 211–226. https://doi.org/10.1007/11494881_14

Koushik Sen and Grigore Rosu. 2003. Generating Optimal Monitors for Extended Regular Expressions. In *Third Workshop on Run-time Verification, RV@CAV 2003, Boulder, Colorado, USA, July 14, 2003 (Electronic Notes in Theoretical Computer Science, Vol. 89)*, Oleg Sokolsky and Mahesh Viswanathan (Eds.). Elsevier, 226–245. https://doi.org/10.1016/S1571-0661(04)81051-X

Koushik Sen, Grigore Rosu, and Gul Agha. 2006. Online efficient predictive safety analysis of multithreaded programs. *Int. J. Softw. Tools Technol. Transf.* 8, 3 (2006), 248–260. https://doi.org/10.1007/S10009-005-0192-Y

Traian-Florin Serbanuta, Feng Chen, and Grigore Rosu. 2012. Maximal Causal Models for Sequentially Consistent Systems. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7687)*, Shaz Qadeer and Serdar Tasiran (Eds.). Springer, 136–150. https://doi.org/10.1007/978-3-642-35632-2_16

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. 2011. Predicting Serializability Violations: SMT-Based Search vs. DPOR-Based Search. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7261)*, Kerstin Eder, João Lourenço, and Onn Shehory (Eds.). Springer, 95–114. https://doi.org/10.1007/978-3-642-34188-5_11

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400. https://doi.org/10.1145/2103656.2103702

L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, Colorado) *(SC '01)*. ACM, New York, NY, USA, 8–8. https://doi.org/10.1145/582034.582042

Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 37–46. https://doi.org/10.1145/1882291.1882300

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

Mandana Vaziri, Frank Tip, and Julian Dolby. 2006. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 334–345. https://doi.org/10.1145/1111037.1111067

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 205–214. https://doi.org/10.1145/1287624.1287654

Virginia Vassilevska Williams. 2018. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*. World Scientific, 3447–3487. https://doi.org/10.1142/9789813272880_0188

Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 317–335. https://doi.org/10.1007/978-3-319-96142-2_20

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