RESEARCH-ARTICLE

The Complexity of Testing Message-Passing Concurrency

Published: 08 January 2026
Accepted: 06 November 2025
Received: 30 June 2025

The Complexity of Testing Message-Passing Concurrency

ZHENG SHI, National University of Singapore, Singapore
LASSE MØLDRUP, Aarhus University, Denmark
UMANG MATHUR, National University of Singapore, Singapore
ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

A key computational question underpinning the automated testing and verification of concurrent programs is the *consistency question — given a partial execution history, can it be completed in a consistent manner?* Due to its importance, consistency testing has been studied extensively for memory models, as well as for database isolation levels. A common theme in all these settings is the use of shared-memory as the primal mode of interthread communication. On the other hand, modern programming languages, such as Go, Rust and Kotlin, advocate a paradigm shift towards channel-based (i.e., message-passing) communication. However, the consistency question for channel-based concurrency is currently poorly understood.

In this paper we lift the study of fundamental consistency problems to channels, taking into account various input parameters, such as the number of threads executing, the number of channels, and the channel capacities. We draw a rich complexity landscape, including upper bounds that become polynomial when certain input parameters are fixed, as well as hardness lower bounds. Our upper bounds are based on algorithms that can drive the verification of channel consistency in automated verification tools. Our lower bounds characterize minimal input parameters that are sufficient for hardness to arise, and thus shed light on the intricacies of testing channel-based concurrency. In combination, our upper and lower bounds characterize the boundary of *tractability/intractability* of verifying channel consistency, and imply that our algorithms are often (nearly) optimal. We have also implemented our main consistency checking algorithm and designed optimizations to enhance its performance. We evaluated the performance of our implementation over a set of 103 instances obtained from open source Go projects, and compared it against a constraint-solving based algorithm. Our experimental results demonstrate the power of our consistency-checking algorithm; it scales to around 1M events, and is significantly faster in running-time performance, compared to a constraint-solving approach.

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

Additional Key Words and Phrases: concurrency, message passing, testing, consistency, channels, Golang

ACM Reference Format:

Zheng Shi, Lasse Møldrup, Umang Mathur, and Andreas Pavlogiannis. 2026. The Complexity of Testing Message-Passing Concurrency. *Proc. ACM Program. Lang.* 10, POPL, Article 1 (January 2026), 32 pages. https://doi.org/10.1145/3776643

Authors' Contact Information: Zheng Shi, National University of Singapore, Singapore, Singapore, shizheng@u.nus.edu; Lasse Møldrup, Aarhus University, Aarhus, Denmark, moeldrup@cs.au.dk; Umang Mathur, National University of Singapore, Singapore, Singapore, umathur@comp.nus.edu.sg; Andreas Pavlogiannis, Aarhus University, Aarhus, Denmark, pavlogiannis@cs.au.dk.

1 Introduction

The verification and testing of concurrent programs has been a major challenge in programming languages and formal methods. Inter-thread communication leads to a combinatorial blow-up in the set of program behaviors, which makes program development error prone and program analysis computationally challenging. Nevertheless, a multitude of techniques have been developed for analyzing concurrent programs automatically, such as bounded model checking [28, 94], partial order reduction [1, 53, 82], predictive runtime testing [52, 67, 75], fuzz testing [71, 91, 93], and static analysis [60, 70]. The vast majority of these techniques operate under the assumption that interthread communication takes place over *shared memory*.

One key problem that has driven the development of algorithmic techniques for concurrency testing and verification is *consistency testing*. At a high level, the input to the problem is a thread-level observable execution of the program (e.g., a sequence of events executed by each thread), without memory-level information about how threads interacted (e.g., a precise thread interleaving, or the order in which writes appeared in the shared memory). The output to the problem is YES iff the thread-level behavior is aligned with the specifics of the underlying architecture (e.g., the memory model). The complexity of consistency testing has been a subject of systematic study for both Sequential Consistency (SC) [20, 41, 42, 66, 88] and weak memory models [23, 38, 58, 87], as well as for database isolation levels [13, 14, 18, 69]. These results have propelled the development of techniques for model checking programs under SC [2, 5, 24, 25, 54] and weak memory [3, 19, 55, 73], as well as for effective testing [15, 49, 51, 62, 67, 72]. In the context of model checking for concurrent programs, efficient consistency testing is key for balancing optimality (i.e., non-redundancy) and performance of the partial-order-reduction based exploration [1, 3, 54]. On the other hand, in the context of runtime predictive analysis, consistency checking plays the dual role of efficiently exploring an entire class of executions that can be inferred from a given execution, without explicitly enumerating members of the class [49, 66, 75].

In order to make concurrent programming more seamless and reliable, modern programming languages advocate for interthread communication mechanisms that are structured and offer clean abstractions. One such case is the use of *message-passing*, popularized by the use of *channels* in Go [45], and also used frequently in other mainstream languages, such as Rust [74], Scala [77], Erlang [34] and Kotlin [56].

Despite the structured communication offered by explicit channel-based concurrency in such languages, subtle concurrency bugs in large-scale software written in this paradigm remain widespread [21, 61, 86]. In turn, this requires the development of verification and testing methods that are capable of reasoning about channels effectively, in order to capture the behaviors entailed by the programs they target [22, 83, 84]. However, the core problem of consistency testing has thus far been elusive for channel-based communication: *How fast can we verify the consistency of message-passing executions?* We address this question in this work, by drawing a rich landscape of the complexity of the problem depending on various input parameters. Besides the technical merit of our results, they also provide a precise characterization of the ingredients that make the consistency problem hard. Likewise, the algorithms we propose can be employed in techniques where soundness and completeness are paramount and a strict upper bound on computational resources is desired.

1.1 Motivating Example

We illustrate the need for consistency checking on channels by means of a simple example where this problem arises naturally. The Go programming language primarily uses the message-passing concurrency paradigm, and offers *channels* as a first class abstraction for interthread communication. A channel in Go is a FIFO queue, possibly with some capacity [43], which a thread can create, close, send to and receive from [44].

```
1 func main() {
2     asyncCh := make(chan int, 2)
3     go func(asyncCh chan int) {
4         asyncCh <- 1
5     }(asyncCh)
6     asyncCh <- 1
7     <-asyncCh
8     close(asyncCh)
9 }
```

(a) A buggy Go code snippet

	τ_1	τ_2
1	create(ch)	
2	spawn(τ_2)	
3		snd(ch, 1)
4	snd(ch, 1)	
5	rcv(ch, 1)	
6	close(ch)	

(b) A non-buggy execution σ.

	τ_1	τ_2
1	create(ch)	
2	spawn(τ_2)	
3	snd(ch, 1)	
4	rcv(ch, 1)	
5	close(ch)	
6		snd(ch, 1)

(c) A buggy execution σ'.

Fig. 1. A buggy Go code snippet on channels with two possible executions

Channel operations. Figure 1a presents a snippet showing the basic channel operations in Go. The main thread creates an asynchronous channel of capacity 2 (Line 2), and passes it as an argument to a child thread executing the goroutine (Line 3–Line 5). The child thread sends value 1 to the channel (Line 4). The main thread sends value 1 to the channel (Line 6), and then receives from it (Line 7), before closing it (Line 8).

Consistency checking in predictive testing. Observe that the program in Figure 1a has a bug: the main thread may execute all its operations and close the channel before the child thread begins to execute. This will cause the child thread to attempt to send to a closed channel, causing the program to panic. Like with many concurrency bugs, exposing this faulty program behavior depends on the scheduler and can be quite challenging. One popular approach to tackle this challenge is through *predictive* runtime testing [52, 67, 75]. Here, as the first step, the program is executed without any explicit schedule control, giving rise to an execution σ. Due to the lack of explicit control, σ is likely to be error-free, i.e., it does not expose the presence of a bug. Figure 1b shows such an execution of the program in Figure 1a. In the second step, σ is analyzed with the goal of constructing an alternative execution σ' that exposes the bug. Here σ' is a permutation of (a slice of) σ and is required to be *sound*, i.e., it can provably be executed by any program that produced σ. Figure 1c shows such a permutation σ'.

The soundness requirement for σ' essentially entails a consistency check. Specifically, each thread must execute the same sequence of operations in σ' as it did in σ, but the interleaving between threads may differ. This leads us to consider an *abstract execution* that can be extracted from σ and captures the per-thread event sequences together with additional ordering constraints; for instance, in our example, we additionally require that close(ch) of thread τ_1 executes before snd(ch, 1) of thread τ_2, while leaving the precise interleaving unspecified. The core question then becomes: can this abstract execution be realized as a concrete trace σ' that respects channel semantics? This is precisely the consistency checking problem.

1.2 The Consistency Checking Problem: Two Variants

We begin with a brief overview of the message-passing consistency checking problem, which is formally defined in Section 2. In general, this problem takes as input a set S of send and receive events, along with a partial order P over S. The partial order P captures the ordering of events within each thread and may also order a channel send before its corresponding receive. The objective is to determine whether there exists a total order on S that is consistent with P and satisfies the FIFO constraints induced by the channels. As an intuitive example, the input to the consistency checking problem may be the set of 6 events in the execution of Figure 1b together with the program order (for example, the send event $\texttt{snd}(\mathsf{ch}, 1)$ of thread τ_1 is ordered before the receive event $\texttt{rcv}(\mathsf{ch}, 1)$ of the same thread), but without the interleaving of the two threads (and the output would be true in this case, since this set can be linearized to both the total order of Figure 1b as well as of Figure 1c).

More formally, the message-passing consistency problem takes as input either a pair $\langle \mathcal{X}, \mathsf{cap}\rangle$ or a triplet $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$, where

- $\mathcal{X}$ is an abstract execution of the form $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}\rangle$, where S is a set of events and po (*program order*) specifies a total order on the events of each thread. The optional component rf (*reads-from* relation) specifies, for each channel receive event $\texttt{rcv}$, the corresponding channel send event $\texttt{snd}$ that $\texttt{rcv}$ obtains its value from.
- The function $\mathsf{cap}\colon \mathsf{Channels}(\mathcal{X}) \to \mathbb{N}$, specifies the capacity of each channel; here $\mathsf{Channels}(\mathcal{X})$ is the set of channels accessed in $\mathcal{X}$.

In line with prior works on consistency testing [23, 41, 87], we distinguish between the following two variants.

- The *verify channel consistency (VCh)* problem takes as input the pair $\langle \mathcal{X}, \mathsf{cap}\rangle$ without any reads-from information. This is the most general variant.
- The *verify channel consistency with reads-from (VCh-rf)* problem takes as input a triplet $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ that contains reads-from information. This variant naturally arises when, e.g., every write to a channel writes a distinct value (for example, this is often imposed during litmus testing [6, 7]), or as a general abstraction mechanism [2, 24, 54].

In each case, the task is to find a linear trace σ realizing $\mathcal{X}$, i.e., σ consists of the events S and agrees with $\mathcal{X}$ on the po (and rf, in the case of VCh-rf).

Remark 1. *For simplicity of presentation, we consider that all interthread communication occurs via channels, and there is no shared memory. This is not a limitation, since a shared register can be simulated by a channel of capacity 1, as we prove in Section 4.2.*

1.3 Summary of Results

We now present the main results of the paper, summarized in Table 1 and Table 2, while we refer to the following sections for details. In the following, we let n, t and m be the total number of events, threads and channels, respectively, in $\mathcal{X}$. We also let $k = \max_{\mathsf{ch}} \mathsf{cap}(\mathsf{ch})$ be the maximum channel capacity. To capture common paradigms of channel programming, we distinguish between channels ch that are *synchronous* ($\mathsf{cap}(\mathsf{ch}) = 0$), *capacity-bounded* or *capacity-unbounded*. We remark that, in our setting, ch is regarded as capacity-unbounded if $\mathcal{X}$ contains $\leq \mathsf{cap}(\mathsf{ch})$ $\texttt{snd}$ events to ch, since then ch cannot block, regardless of how $\mathcal{X}$ is scheduled[1]. For example, if $\mathsf{cap}(\mathsf{ch}) = 3$ but $\mathcal{X}$ only contains two send events to ch, then ch behaves as a capacity-unbounded channel in $\mathcal{X}$ (even though its capacity is finite).

To illustrate the intricacies of channels, we begin with two restricted cases of VCh for which the problem is nevertheless intractable. First, let us consider the case where all channel events send/receive the same value, and thus each receive may observe from any send. In this setting, we show the following via a reduction from the Hamiltonian cycle problem.

Theorem 1.1. *VCh is NP-complete even if all events send/receive the same value.*

[1]This is in contrast to the colloquial use of "unbounded" meaning "of infinite capacity".

Table 1. Results for the channel consistency problem VCh on abstract executions of n events, t threads, m channels, each with capacity $\leq k$.

Reference	Variant	Complexity
Theorem 1.1	Every event sends/receives the same value	NP-complete
Theorem 1.2	$t = 2$ and each channel is capacity-unbounded	NP-complete
Theorem 1.3	$m = 1$ and either $k = 0$ (synchronous channel) or $k = 1$	NP-complete
Theorem 1.4	General case	$O\left(n^{t+1} \cdot t^{km}\right)$

Table 2. Results for the channel consistency problem with a reads-from relation VCh-rf on abstract executions of n events, t threads, m channels, each with capacity $\leq k$. (†) holds under SETH.

Reference	Variant	Complexity
Theorem 1.5	General case	$O(n^{t+1} \cdot (\min(k!, t^k)^m)$
Theorem 1.6	$k = 1$ and every channel is asynchronous, or $t = 3$ and $k = 2$, or $t = 3$ and $m = 5$ and each channel is capacity-unbounded	NP-complete
Theorem 1.7	Acyclic topology and each channel has capacity ≤ 1 or is unbounded	$O(n^2)$
Theorem 1.8	$t = 2$ and each channel has capacity 1, or $t = 2$ and each channel is capacity-unbounded	Not in† $O(n^{2-\epsilon})$
Theorem 1.9	Each channel is synchronous	$O(n)$

The corresponding consistency problem for shared memory is trivial: as reads/writes are on the same value, any linearization that starts with a write is a valid trace. This is not the case for VCh, as σ must also respect channel capacities. Second, we show that the problem is intractable already with just two threads using a reduction from postive 1-in-3 3SAT.

Theorem 1.2. *VCh is NP-complete even if $t = 2$ and each channel is capacity-unbounded.*

In contrast, the smallest number of threads which make consistency for shared memory intractable is $t = 3$ [42]. Third, we show that the problem becomes intractable already with just a single channel, which is either synchronous or has capacity 1 using a reduction from the VSC-read studied and proved to be NP-hard in [42]; the VSC-read problem is the analogue of the VCh-rf problem we study in this work, but for the case of executions with registers instead of channels. This result is analogous to the hardness for shared memory on a single location [20] (but is not subsumed by it, since synchronous channels are blocking, in contrast to shared memory).

Theorem 1.3. *VCh is NP-complete even if $m = 1$ and either $k = 0$ (synchronous channel) or $k = 1$.*

Given the above hardness results even on restricted inputs, it is imperative to ask — how fast can we solve VCh in general? The following theorem establishes an upper bound with explicit dependence on the input parameters. Our algorithm for checking VCh traverses a *frontier graph* whose nodes track succinct information about configurations of threads and channels.

Theorem 1.4. *VCh can be solved in $O\left(n^{t+1} \cdot t^{km}\right)$ time.*

Let us now turn our attention to the simpler problem, VCh-rf. Since VCh-rf is a special case of VCh, the upper bound in Theorem 1.4 also applies to VCh-rf. We show that VCh-rf admits, in fact, a somewhat faster algorithm using a more succinct frontier graph.

THEOREM 1.5. *VCh-rf can be solved in $O(n^{t+1} \cdot (\min(k!, t^k)^m)$ time.*

Observe that both upper bounds (Theorem 1.4 and Theorem 1.5) become polynomial when the input parameters are bounded (i.e., fixed constants). When this is not the case, we ask whether one has to suffer an exponential dependence on each of these parameters. In other words, *does the problem become tractable when only some, but not all, of the parameters are bounded?* Unfortunately, as the next theorem states, even the easier problem VCh-rf remains intractable when only some parameters are bounded, via reductions from 3SAT and VSC-read problem.

THEOREM 1.6. *VCh-rf is NP-complete if any of the following three conditions holds: (i) $k = 1$ and every channel is asynchronous, or (ii) $t = 3$ and $k = 2$, or (iii) $t = 3$ and $m = 5$ and each channel is capacity-unbounded.*

Given the hardness of Theorem 1.6, the next natural question is whether VCh-rf becomes tractable for any natural (semantic) classes besides the (syntactic) restrictions governed by the parameters above. Towards this, we consider the *communication topology* $G = (V, E)$ of $\mathcal{X}$, where V contains the set of threads of $\mathcal{X}$, and we have an edge $(\tau_1, \tau_2) \in E$ iff threads τ_1 and τ_2 access a common channel. We prove that the problem becomes tractable when G is acyclic and can be solved via a reduction to the satisfiability problem of a quadratically long 2CNF formula.

THEOREM 1.7. *VCh-rf is solvable in $O(n^2)$ time on acyclic communication topologies if each channel is either capacity-unbounded or has capacity ≤ 1.*

Common acyclic topologies include pipelines, server-client architectures, and general tree structures. We remark that Theorem 1.7 allows for any combination of channels that are capacity-unbounded, have capacity 1, or are synchronous (i.e., have capacity 0). Observe that the case $t = 2$ results in an acyclic communication topology. Due to Theorem 1.2, an analogous polynomial bound for VCh on acyclic topologies is not possible, as the problem is NP-complete already for $t = 2$ threads.

At this point, it is natural to ask whether any improvements are possible over this quadratic bound, e.g., does the problem admit a linear-time solution on acyclic topologies? To answer this question, we equip techniques from fine-grained complexity theory, and in particular, the popular strong exponential time hypothesis (SETH). We establish the following lower bound with a reduction from *Orthogonal Vector* (OV) problem, which is hard for quadratic time conditioned on SETH.

THEOREM 1.8. *Under SETH, VCh-rf cannot be solved in time $O(n^{2-\epsilon})$ for any $\epsilon > 0$, even if $t = 2$ and either (i) all channels are capacity-unbounded, or (ii) all channels have capacity 1.*

Together, Theorem 1.7 and Theorem 1.8 yield a tight dichotomy: the problem takes quadratic time on acyclic topologies, and this bound is optimal, even for the simplest such topology. Finally, we consider fully synchronous channels, and show that the problem reduces to cycle detection on a graph, which admits a linear time algorithm (no matter what topology).

THEOREM 1.9. *VCh-rf is solvable in $O(n)$ time if all channels are synchronous.*

Observe that this is in sharp contrast to VCh, for which the problem is intractable already with only one synchronous channel (Theorem 1.3).

Overview of empirical evaluation. We have implemented our algorithm for channel consistency with reads-from (Theorem 1.5), primarily to demonstrate its efficacy over a vanilla approach of encoding the (NP-complete) channel consistency problem as an SMT formula. We evaluated the performance of our algorithm as well as the vanilla SMT solving based approach on a comprehensive suite of 103 benchmarks derived from real-world Golang programs. The results indicate that our algorithm exhibits superior scalability compared to a SMT-based approach, achieving a faster completion time while encountering fewer timeouts. Furthermore, despite VCh-rf being an NP-hard problem, an optimized version of our algorithm successfully scales to large instances, handling up to 35k events, 2k threads, and 14k channels. These findings confirm our hypothesis that our frontier graph based algorithm (Theorem 1.5) is a highly efficient solution for channel consistency checking.

Outline. The technical parts of the paper are organized as follows. In Section 2, we set up relevant notation and define the consistency problem for channels. In Section 3, we develop algorithms for the upper bounds in Theorem 1.4, Theorem 1.5, Theorem 1.7 and Theorem 1.9. Finally, in Section 4 we prove item (iii) of Theorem 1.6 and Theorem 1.8. Due to space restrictions, all formal proofs, Theorem 1.1, Theorem 1.2, Theorem 1.3, and items (i) and (ii) of Theorem 1.6 are relegated to our companion technical report [81].

2 Preliminaries

In this section we formalize the basic concepts of channel-based executions and define the corresponding consistency-checking problems.

2.1 Events and Executions

Channels. We model channels as FIFO queues with (bounded or unbounded) capacities. A *send* operation on a channel ch enqueues a message to the FIFO queue, while a *receive* operation pops a message from the queue. The *capacity* $\mathsf{cap}(\mathsf{ch})$ of ch dictates how many messages can be enqueued in it simultaneously. When ch is full (i.e., contains $\mathsf{cap}(\mathsf{ch})$ messages), send operations on it will block, until at least one receive operation is executed on it. We further call ch *synchronous* if $\mathsf{cap}(\mathsf{ch}) = 0$. Intuitively, synchronous channels do not buffer any messages, and thus a send operation on ch must be immediately followed by a receive operation. An *asynchronous* channel ch, on the other hand, has $\mathsf{cap}(\mathsf{ch}) > 0$ and allows for asynchronous send and receive operations.

Events. An event is a tuple $e = \langle id, \tau, \mathsf{op}(\mathsf{ch}, \mathsf{val})\rangle$, consisting of the unique identifier id of e, the identifier τ of the thread that performs e, the operation $\mathsf{op} \in \{\mathsf{snd}, \mathsf{rcv}\}$ (a channel send or receive)[2] performed by e, the identifier of the channel ch involved in the event e and the value val sent or received. We write $\mathsf{th}(e)$, $\mathsf{op}(e)$, $\mathsf{ch}(e)$, $\mathsf{val}(e)$ for the thread, operation, channel and value of e, respectively. We often use the more succinct notation $\mathsf{snd}(\mathsf{ch}, \mathsf{val})$ / $\mathsf{rcv}(\mathsf{ch}, \mathsf{val})$, when the unique identifier id and thread identifier τ are clear from the context, or not important.

Executions and well-formedness. An execution is a finite sequence of events $\sigma = e_1 e_2 \dots e_n$ of length $|\sigma| = n$. We denote by $\mathsf{Events}(\sigma) = \{e_1, \dots, e_n\}$ the set of events, by $\mathsf{Threads}(\sigma)$ the set of threads, and by $\mathsf{Channels}(\sigma)$ the set of channels appearing in σ. For some channel $\mathsf{ch} \in \mathsf{Channels}(\sigma)$, we use $\sigma{\downharpoonleft}_{\mathsf{ch}}$ to denote the maximal subsequence of σ containing only events accessing ch. Likewise, we use $\sigma{\downharpoonleft}_{\mathsf{snd}(\mathsf{ch})}$ (resp. $\sigma{\downharpoonleft}_{\mathsf{rcv}(\mathsf{ch})}$) to denote the projection of σ onto the send (resp. receive) events on ch. We require that executions are *well-formed*, meaning that they respect the channel semantics. Well-formedness requires that σ satisfies the following two types of constraints.

Capacity Constraints. These require that σ respects the channel capacities. In particular, for each channel $\mathsf{ch} \in \mathsf{Channels}(\sigma)$, the following hold.

[2] Our results are easily extended to a setting that contains other common events such as thread `fork`/`join` and channel `create`/`close`. We omit such events for ease of presentation.

1. (*Asynchronous channels*) If $\mathsf{cap}(\mathsf{ch}) > 0$, then for each prefix π of σ, we have
$$|\pi\downharpoonright_{\mathsf{rcv}(\mathsf{ch})}| \leq |\pi\downharpoonright_{\mathsf{snd}(\mathsf{ch})}| \leq |\pi\downharpoonright_{\mathsf{rcv}(\mathsf{ch})}| + \mathsf{cap}(\mathsf{ch}).$$
In other words, every receive event should observe a send event and the number of buffered send events cannot exceed the channel capacity.
2. (*Synchronous channels*) If $\mathsf{cap}(\mathsf{ch}) = 0$, then each send event $e = \langle\tau, \mathsf{snd}(\mathsf{ch})\rangle$ on ch must immediately be followed by a matching receive event $e' = \langle\tau', \mathsf{rcv}(\mathsf{ch})\rangle$ from a different thread $\tau' \neq \tau$. Likewise, each receive event $e = \langle\tau, \mathsf{rcv}(\mathsf{ch})\rangle$ on σ must be immediately preceded by a matching send event $e' = \langle\tau', \mathsf{snd}(\mathsf{ch})\rangle$ from a different thread $\tau' \neq \tau$. Observe that this implies an equal number of send and receive events on ch.

A thread attempting to send on a full channel is blocked (normally by the runtime), until the channel is received, freeing up space for the new incoming message. The events listed in σ are executed events, meaning that each channel send completed successfully, and was thus performed on a non-full channel. For synchronous channels, a send operation is executed simultaneously with its matching receive, since capacity 0 does not allow storing the message sent.

Value Constraints. These require that matching $\mathsf{snd}/\mathsf{rcv}$ events on the same channel observe identical values. In particular, for each channel $\mathsf{ch} \in \mathsf{Channels}(\sigma)$, for each $1 \leq i \leq |\sigma\downharpoonright_{\mathsf{rcv}(\mathsf{ch})}|$, if the i-th send (resp., receive) event in ch is $\mathsf{snd}(\mathsf{ch}, \mathsf{val}_1)$ (resp., $\mathsf{rcv}(\mathsf{ch}, \mathsf{val}_2)$), then $\mathsf{val}_1 = \mathsf{val}_2$.

	τ_1	τ_2
1	$\mathsf{snd}(\mathsf{ch}_1, 1)$	
2	$\mathsf{snd}(\mathsf{ch}_1, 2)$	
3		$\mathsf{rcv}(\mathsf{ch}_1, 1)$
4		$\mathsf{rcv}(\mathsf{ch}_1, 2)$
5		$\mathsf{snd}(\mathsf{ch}_2, 3)$
6	$\mathsf{rcv}(\mathsf{ch}_2, 3)$	

(a) Execution σ_1

	τ_1	τ_2
1	$\mathsf{snd}(\mathsf{ch}_1, 1)$	
2	$\mathsf{snd}(\mathsf{ch}_1, 2)$	
3		$\mathsf{snd}(\mathsf{ch}_1, 3)$
4	$\mathsf{rcv}(\mathsf{ch}_1, 1)$	
5		$\mathsf{rcv}(\mathsf{ch}_1, 2)$
6		$\mathsf{rcv}(\mathsf{ch}_1, 3)$

(b) Execution σ_2

	τ_1	τ_2
1	$\mathsf{snd}(\mathsf{ch}_1, 1)$	
2	$\mathsf{snd}(\mathsf{ch}_2, 2)$	
3	$\mathsf{rcv}(\mathsf{ch}_1, 1)$	
4		$\mathsf{snd}(\mathsf{ch}_1, 3)$
5		$\mathsf{rcv}(\mathsf{ch}_2, 2)$
6		$\mathsf{rcv}(\mathsf{ch}_1, 3)$

(c) Execution σ_3

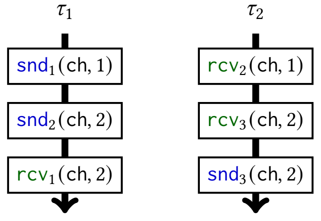
	τ_1	τ_2
1	$\mathsf{snd}(\mathsf{ch}_1, 1)$	
2	$\mathsf{snd}(\mathsf{ch}_1, 2)$	
3	$\mathsf{rcv}(\mathsf{ch}_1, 1)$	
4		$\mathsf{snd}(\mathsf{ch}_1, 3)$
5		$\mathsf{rcv}(\mathsf{ch}_1, 3)$
6		$\mathsf{rcv}(\mathsf{ch}_1, 2)$

(d) Execution σ_4

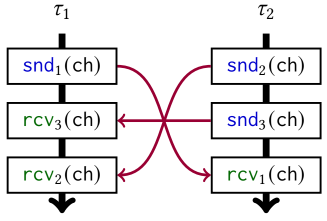
Fig. 2. Four executions on two channels ch_1 and ch_2 with capacities $\mathsf{cap}(\mathsf{ch}_1) = 2$ and $\mathsf{cap}(\mathsf{ch}_2) = 0$. Execution σ_1 is well-formed but $\sigma_2, \sigma_3, \sigma_4$ are not.

Example 1. *Consider the four executions $\sigma_1, \sigma_2, \sigma_3$ and σ_4 in Figure 2. Each σ_i contains 6 events and uses two channels ch_1 and ch_2 whose capacities are $\mathsf{cap}(\mathsf{ch}_1) = 2$ (i.e., asynchronous channel) and $\mathsf{cap}(\mathsf{ch}_2) = 0$ (i.e., synchronous channel) respectively. We use e_i to denote the i^{th} event of an execution. First, consider the execution σ_1 (Figure 2a), which is well-formed. The capacity constraint on ch_2 is met because the (unique) send (e_5) and receive (e_6) events on ch_2 appear consecutively. Further, the two events access the same value. Moreover, in every prefix of σ_1, the number of buffered messages in ch_1 never exceeds its capacity 2, and the order of values being sent ($1 \longrightarrow 2$) matches that of the values being received on ch_1, ensuring the value constraint for ch_1 as well. Now, consider σ_2 in Figure 2b, which is not well-formed since, at e_3, ch_1 contains 3 messages, exceeding its capacity. Next, the execution σ_3 in Figure 2c is not well-formed, because the send and receive events (e_2 and e_5) on the synchronous channel ch_2 are not consecutive. Finally, the execution σ_4 in Figure 2d is not well-formed since the order of values sent ($1 \longrightarrow 2 \longrightarrow 3$) is not the same as the order of values received ($1 \longrightarrow 3 \longrightarrow 2$).*

Trace order, program order and the reads-from relation. The *trace order* of an execution σ, denoted $<^{\sigma}_{\mathsf{tr}}$, is the total order on $\mathsf{Events}(\sigma)$ induced by the sequence σ. The *program order* po_σ of σ defines a total order on the events of each thread, i.e., for any two events $e_1, e_2 \in \mathsf{Events}(\sigma)$, we have $(e_1, e_2) \in \mathsf{po}_\sigma$ iff $e_1 <^{\sigma}_{\mathsf{tr}} e_2$ and $\mathsf{th}(e_1) = \mathsf{th}(e_2)$. The (binary) *reads-from* relation rf_σ induced by σ maps receive events to their matching send events. That is, $(\mathsf{snd}, \mathsf{rcv}) \in \mathsf{rf}_\sigma$, iff there is a channel



(a) A VCh instance $\langle \mathcal{X}_1, \mathrm{cap}_1 \rangle$



(b) A VCh-rf instance $\langle \mathcal{X}_2, \mathrm{cap}_2, \mathrm{rf} \rangle$

Fig. 3. A positive VCh instance (a) and a negative VCh-rf instance (b). $\mathrm{cap}_1(\mathrm{ch}) = \mathrm{cap}_2(\mathrm{ch}) = 1$. Event subscripts are used to distinguish send/receive events. The same convention applies in subsequent figures.

$\mathrm{ch} \in \mathrm{Channels}(\sigma)$ and some $i \in \mathbb{N}$ such that snd is the i^{th} send event and rcv is the i^{th} receive event on ch. We often use the shorthand $\mathrm{rf}_\sigma(\mathrm{rcv})$ for the event snd such that $(\mathrm{snd}, \mathrm{rcv}) \in \mathrm{rf}_\sigma$.

Example 2. *Consider again the execution σ_1 in Figure 2a. We have $rf_{\sigma_1}(e_3) = e_1$, $rf_{\sigma_1}(e_4) = e_2$ and $rf_{\sigma_1}(e_6) = e_5$. We have $(e_1, e_3) \in rf_{\sigma_1}$ and $(e_2, e_4) \in rf_{\sigma_1}$. The program order of σ_1 is $po_{\sigma_1} = \{(e_1, e_2), (e_2, e_6), (e_3, e_4), (e_4, e_5)\}^+$, where, R^+ denotes the transitive closure of the binary relation R.*

2.2 Verifying the Consistency of Message-Passing Concurrency

We now state the consistency problem we study in this work.

Abstract executions and consistency. The consistency problem is phrased on a pair $\langle \mathcal{X}, \mathrm{cap} \rangle$, where an *abstract* execution $\mathcal{X}$ captures the local execution of each thread and a capacity function $\mathrm{cap}\colon \mathrm{Channels}(\mathcal{X}) \longrightarrow \mathbb{N}$ specifies the capacity of each channel, where $\mathrm{Channels}(\mathcal{X})$ is the set of channels accessed by events in $\mathcal{X}$. An abstract execution is a tuple $\mathcal{X} = \langle \mathsf{S}, \mathrm{po} \rangle$, where S is some set of events, and po describes a per-thread total order on events in S. An execution σ is a *concretization* of $\mathcal{X} = \langle \mathsf{S}, \mathrm{po} \rangle$ with capacity function cap if (i) $\mathrm{Events}(\sigma) = \mathsf{S}$, (ii) $\mathrm{po}_\sigma = \mathrm{po}$, and (iii) σ is well-formed with respect to the channel capacities specified by cap. Finally, $\langle \mathcal{X}, \mathrm{cap} \rangle$ is *consistent* if there exists an execution σ that concretizes it. The consistency checking problem is thus formally stated below.

Problem 1 (Verify channel consistency, VCh)**.** *Given an abstract execution $\mathcal{X} = \langle \mathsf{S}, po \rangle$ and capacity function* cap*, decide if $\langle \mathcal{X}, \mathrm{cap} \rangle$ is consistent.*

Consistency with a reads-from relation. The *consistency problem with a reads-from relation* is a tuple $\langle \mathcal{X}, \mathrm{cap}, \mathrm{rf} \rangle$, where S and po are, as before, respectively a set of events and a per-thread total order on this set, while rf matches send and receive events of S on the same channel. An execution σ concretizes $\mathcal{X} = \langle \mathsf{S}, \mathrm{po} \rangle$ and rf if it concretizes $\langle \mathsf{S}, \mathrm{po} \rangle$ (as in VCh), and moreover $\mathrm{rf}_\sigma = \mathrm{rf}$. In later sections, we omit the values of events in VCh-rf instances, as the values are not relevant once the reads-from relation is given. The corresponding consistency problem is defined analogously.

Problem 2 (Verify channel consistency with reads-from, VCh-rf)**.** *Given an abstract execution $\mathcal{X} = \langle \mathsf{S}, po \rangle$ with reads-from relation rf and capacity function* cap*, decide if $\langle \mathcal{X}, \mathrm{cap}, rf \rangle$ is consistent.*

It is not hard to see that VCh-rf is an easier problem than VCh, in the sense that the former is a special case of the latter (e.g., by requiring that every send uses a unique value).

Example 3. *Figure 3a is a positive instance of VCh, witnessed by the execution $\sigma_1 = \mathrm{snd}_1 \cdot \mathrm{rcv}_2 \cdot \mathrm{snd}_2 \cdot \mathrm{rcv}_3 \cdot \mathrm{snd}_3 \cdot \mathrm{rcv}_1$. Figure 3b is a negative instance of VCh-rf. This is because any execution σ that concretizes $\langle \mathcal{X}_2, \mathrm{cap}_2, rf \rangle$ must satisfy $\mathrm{rcv}_3 <_{tr}^{\sigma} \mathrm{rcv}_2$ and $\mathrm{snd}_2 <_{tr}^{\sigma} \mathrm{snd}_3$, due to the imposed program order. The former, however, implies $\mathrm{snd}_3 <_{tr}^{\sigma} \mathrm{snd}_2$, contradicting the latter.*

3 Algorithms for Checking Consistency

In this section we present algorithms for solving VCh and VCh-rf. In particular, in Section 3.1 we develop the general algorithms for the two problems, leading to Theorem 1.4 and Theorem 1.5. Then, in Section 3.2, we focus on the special case of fully synchronous channels, and develop an efficient (linear-time) algorithm towards Theorem 1.9. Finally, in Section 3.3 we focus on acyclic communication topologies, and develop a quadratic-time algorithm towards Theorem 1.7.

3.1 Algorithms for VCh and VCh-rf

We now present our algorithms for VCh and VCh-rf. A naive algorithm for either problem would enumerate all possible permutations of the input set of events and look for one permutation that serves as the witness of consistency. However, this approach takes $\Omega(n!)$ time, which is significantly worse than the bounds we aim for.

Our algorithms for each problem circumvent this prohibitive complexity by succinctly encoding executions as paths in a *frontier graph*, which has polynomial size when the number of threads t, the number of channels m and the maximum channel capacity k are bounded.

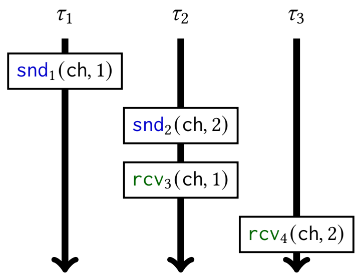
Frontier graphs have previously been developed for consistency testing under shared memory [2, 5, 41], but not for channel-based concurrency. In the shared-memory setting, each read observes the most recent write, so nodes in the frontier graph only need to record the latest value of each memory location. In contrast, channels pose greater challenges for constructing succinct frontier graphs due to the vastly larger space of possible consistent executions. Here, nodes must also encode channel contents, making state counting both complex and subtle.

The frontier graph for VCh. Given a VCh instance $\langle \mathcal{X}, \mathsf{cap}\rangle$ where $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}\rangle$, we define its frontier graph $G_{\mathsf{frontier}} = (V, E)$ as follows.

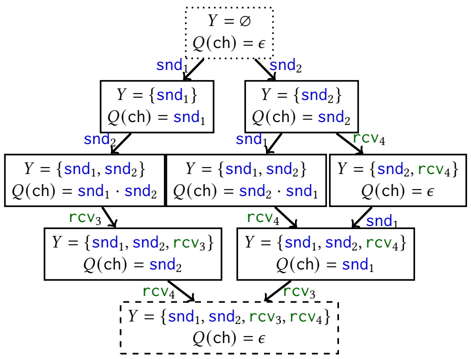
The node set V. Each node $v \in V$ is a tuple of the form $v = \langle Y, Q, I\rangle$. Intuitively, Y specifies the subset of events of $\mathcal{X}$ that an execution has executed when it reaches the corresponding node in G_{frontier}. Q specifies the contents of the asynchronous channels, while I specifies the (at most one) send event on a synchronous channel that must be matched in the next step. We now formally specify Y, Q and I as follows.

1. $Y \subseteq \mathsf{S}$, and Y is downward closed with respect to po, i.e., for each $(e, f) \in \mathsf{po}$ and if $f \in Y$, then $e \in Y$. Given a channel ch, let $\mathsf{num}_Y(\mathsf{snd}(\mathsf{ch}))$ and $\mathsf{num}_Y(\mathsf{rcv}(\mathsf{ch}))$ denote the number of send and receive events on ch in Y. First, we require that there is at most one synchronous channel ch with $\mathsf{num}_Y(\mathsf{rcv}(\mathsf{ch})) = \mathsf{num}_Y(\mathsf{snd}(\mathsf{ch})) - 1$, while for all other synchronous channels ch′, we have $\mathsf{num}_Y(\mathsf{rcv}(\mathsf{ch}')) = \mathsf{num}_Y(\mathsf{snd}(\mathsf{ch}'))$. Second, we require that for any asynchronous channel ch, the following holds.
$$\mathsf{num}_Y(\mathsf{rcv}(\mathsf{ch})) \leq \mathsf{num}_Y(\mathsf{snd}(\mathsf{ch})) \leq \mathsf{num}_Y(\mathsf{rcv}(\mathsf{ch})) + \mathsf{cap}(\mathsf{ch})$$
2. Q: $\mathsf{Channels}(\mathcal{X}) \rightarrow Y^{\leq k}$ maps each asynchronous channel ch in S (i.e., $\mathsf{cap}(\mathsf{ch}) > 0$) to a sequence of events in Y, whose length is bounded by $\mathsf{cap}(\mathsf{ch})$, i.e., $Q(\mathsf{ch}) = e_1 \cdot e_2 \cdots e_p$, where $0 \leq p \leq \mathsf{cap}(\mathsf{ch}) \leq k$, and $e_1, \dots, e_p \in Y$. Moreover, for any asynchronous channel ch, Q must satisfy that if some event e appears in $Q(\mathsf{ch})$, then e is one of the last $|Q(\mathsf{ch})|$ send events to channel ch in thread $\mathsf{th}(e)$.
3. I is either $\perp$ or points to a send event of a synchronous channel. In particular, if there is a synchronous channel ch such that $\mathsf{num}_Y(\mathsf{rcv}(\mathsf{ch})) = \mathsf{num}_Y(\mathsf{snd}(\mathsf{ch})) - 1$, then $I = \mathsf{snd}$, for some send event snd on ch. Otherwise, $I = \perp$.

Finally, we have a distinguished *source node*, defined as $\langle \emptyset, \lambda\, \mathsf{ch}.\epsilon, \perp\rangle$, as well as one or more *sink nodes*, defined as $\langle \mathsf{S}, Q, \perp\rangle$. In words, the source node captures the case that no event of $\mathcal{X}$ has been


τ_1
τ_2
τ_3
$\mathsf{snd}_1(\mathsf{ch}, 1)$
$\mathsf{snd}_2(\mathsf{ch}, 2)$
$\mathsf{rcv}_3(\mathsf{ch}, 1)$
$\mathsf{rcv}_4(\mathsf{ch}, 2)$

(a) A VCh instance $\langle \mathcal{X}, \mathsf{cap} \rangle$ with $\mathsf{cap}(\mathsf{ch}) = 2$.


$Y = \emptyset$, $Q(\mathsf{ch}) = \epsilon$
snd_1
snd_2
$Y = \{\mathsf{snd}_1\}$, $Q(\mathsf{ch}) = \mathsf{snd}_1$
$Y = \{\mathsf{snd}_2\}$, $Q(\mathsf{ch}) = \mathsf{snd}_2$
snd_2
snd_1
rcv_4
$Y = \{\mathsf{snd}_1, \mathsf{snd}_2\}$, $Q(\mathsf{ch}) = \mathsf{snd}_1 \cdot \mathsf{snd}_2$
$Y = \{\mathsf{snd}_1, \mathsf{snd}_2\}$, $Q(\mathsf{ch}) = \mathsf{snd}_2 \cdot \mathsf{snd}_1$
$Y = \{\mathsf{snd}_2, \mathsf{rcv}_4\}$, $Q(\mathsf{ch}) = \epsilon$
rcv_3
rcv_4
snd_1
$Y = \{\mathsf{snd}_1, \mathsf{snd}_2, \mathsf{rcv}_3\}$, $Q(\mathsf{ch}) = \mathsf{snd}_2$
$Y = \{\mathsf{snd}_1, \mathsf{snd}_2, \mathsf{rcv}_4\}$, $Q(\mathsf{ch}) = \mathsf{snd}_1$
rcv_4
rcv_3
$Y = \{\mathsf{snd}_1, \mathsf{snd}_2, \mathsf{rcv}_3, \mathsf{rcv}_4\}$, $Q(\mathsf{ch}) = \epsilon$

(b) The frontier graph G_{frontier} for $\langle \mathcal{X}, \mathsf{cap} \rangle$

Fig. 4. A VCh instance (a) and its frontier graph (b), witnessing the consistency of $\langle \mathcal{X}, \mathsf{cap} \rangle$. There is a path from source (dotted node) to sink (dashed node), and the events labelling this path form a valid concretization, i.e., $\sigma = \mathsf{snd}_1 \cdot \mathsf{snd}_2 \cdot \mathsf{rcv}_3 \cdot \mathsf{rcv}_4$. Therefore, $\langle \mathcal{X}, \mathsf{cap} \rangle$ is consistent.

executed, while a sink node captures that all events of $\mathcal{X}$ have been executed (sink nodes might differ on the contents of the channels Q, containing messages that are never received).

The edge set E. Concrete executions that serve as potential witnesses of the consistency of $\langle \mathcal{X}, \mathsf{cap} \rangle$ are captured as paths in G_{frontier} starting from the source node. An edge $(v_1, v_2) \in E$ intuitively captures whether *any* execution reaching v_1 can be extended to v_2. The information contained in v_1 is sufficient to decide whether this is possible. In particular, let $v_1 = \langle Y_1, Q_1, I_1 \rangle$ and $v_2 = \langle Y_2, Q_2, I_2 \rangle$. We have $(v_1, v_2) \in E$ if there is an event $e \in \mathsf{S} \setminus Y_1$ such that $Y_2 = Y_1 \cup \{e\}$ and the following conditions hold, where $\mathsf{ch} = \mathsf{ch}(e)$.

1. If ch is asynchronous and $\mathsf{op}(e) = \mathsf{rcv}$, then we require that the following hold.
   a) $I_1 = \bot, I_2 = \bot$.
   b) $Q_1(\mathsf{ch}) \neq \epsilon$, and the first event of $Q_1(\mathsf{ch})$, i.e., $e_{Q_1,\mathsf{ch},\mathsf{first}} = Q_1(\mathsf{ch})[0]$ satisfies $\mathsf{val}(e_{Q_1,\mathsf{ch},\mathsf{first}}) = \mathsf{val}(e)$. Moreover, $Q_2(\mathsf{ch})$ is obtained by removing the first event of $Q_1(\mathsf{ch})$, i.e., $Q_1(\mathsf{ch}) = e_{Q_1,\mathsf{ch},\mathsf{first}} \cdot Q_2(\mathsf{ch})$.
   c) For all other asynchronous channels $\mathsf{ch}' \neq \mathsf{ch}$, we have $Q_2(\mathsf{ch}') = Q_1(\mathsf{ch}')$.
2. If ch is asynchronous and $\mathsf{op}(e) = \mathsf{snd}$, then we require that the following hold.
   a) $I_1 = \bot, I_2 = \bot$.
   b) $|Q_1(\mathsf{ch})| < \mathsf{cap}(\mathsf{ch})$, and $Q_2(\mathsf{ch})$ is obtained by appending e at the end of $Q_1(\mathsf{ch})$, i.e., $Q_2(\mathsf{ch}) = Q_1(\mathsf{ch}) \cdot e$.
   c) For all other asynchronous channels $\mathsf{ch}' \neq \mathsf{ch}$, we have $Q_2(\mathsf{ch}') = Q_1(\mathsf{ch}')$.
3. If ch is synchronous and $\mathsf{op}(e) = \mathsf{snd}$, then we require that (1) $I_1 = \bot, I_2 = e$, and (2) for all asynchronous channels ch', $Q_1(\mathsf{ch}') = Q_2(\mathsf{ch}')$.
4. If ch is synchronous and $\mathsf{op}(e) = \mathsf{rcv}$, then we require that (1) $I_1 = e' \neq \bot, I_2 = \bot$, and e' satisfies $\mathsf{op}(e') = \mathsf{snd}$, $\mathsf{ch}(e') = \mathsf{ch}$, $\mathsf{val}(e') = \mathsf{val}(e)$, and $\mathsf{th}(e) \neq \mathsf{th}(e')$, and, (2) for all asynchronous channels ch', $Q_1(\mathsf{ch}') = Q_2(\mathsf{ch}')$.

If the above hold, we say that the edge (v_1, v_2) is labeled by e, and often write $v_1 \xrightarrow{e} v_2$. See Figure 4 for an example. The following lemma states that G_{frontier} captures the consistency of $\langle \mathcal{X}, \mathsf{cap} \rangle$.

Lemma 3.1. *$\langle \mathcal{X}, \mathsf{cap} \rangle$ is consistent iff there is a sink node reachable from the source node in G_{frontier}.*

Time complexity. Given Lemma 3.1, we can solve VCh by constructing G_{frontier} and solving standard graph reachability on it. Recall that each node is a tuple $\langle Y, Q, I \rangle$. Y is a po-downward closed set, and there are at most (n^t/t^t) many distinct subsets of S of this form (using AM-GM inequality).

For each fixed Y, the number of different possible I is upper bounded by $t + 1$, since I is either $\perp$ or points to the last event of a thread in Y. Finally, consider the component Q. For any asynchronous channel ch, the number of messages in $Q(\mathsf{ch})$ is $\ell = \mathsf{num}_Y(\mathtt{snd}(\mathsf{ch})) - \mathsf{num}_Y(\mathtt{rcv}(\mathsf{ch}))$. A naive counting approach would enumerate the last ℓ sends to channel ch in every thread, forming a potential set of unreceived sends of size $t\ell$. Selecting ℓ elements from this set yields a total of $O((tk)^k)$ possible combinations, as $\ell \leq k$. However, our counting method is more refined. We notice that the sequence of events in the queue $Q(\mathsf{ch})$ is uniquely determined by the sequence $\tau_{i_0}, \tau_{i_1} \ldots, \tau_{i_{\ell-1}}$ of their thread identifiers — if $\tau_{i_{\ell-j}} = \tau$, then the j^{th} *last* event e in $Q(\mathsf{ch})$ belongs to thread τ and, further e is the m^{th} last send event on channel ch that thread τ performs, where $m = |\{z \mid \tau_z = \tau, \ell - j \leq z\}|$. Since the number of threads is t, the total number of possible sequences corresponding to $Q(\mathsf{ch})$ is thus $\leq t^\ell \in O(t^k)$. This implies that the total number of different values that Q can take on is in $O(t^{km})$. Thus, the total number of nodes of G_{frontier} is $O(n^t/t^t \cdot t \cdot t^{km})$.

We now count the number of edges in G_{frontier}. Each node has at most t out-degree since the set Y is po downward closed for each node. Hence the number of edges in G_{frontier} is bounded by $(n^t/t^t \cdot t^2 \cdot t^{km})$. Thus, the time for reachability checking is $O(|V| + |E|) = O(n^t \cdot t^{km})$.

The graph can be constructed using a simple worklist algorithm. The worklist is initialized with only the source node. The algorithm proceeds by repeatedly extracting a node v from the worklist and inserting its successors until the worklist is empty. To compute the successor node v' of the current node v by extending v with event e, we first copy v into v' and update v' according to the rules of the frontier graph. Copying v takes $O(n)$ time and updating v' takes constant time, and thus time to construct the graph is $O(n \cdot V + E) = O(n^{t+1} \cdot t^{km})$ as in Theorem 1.4.

The algorithm for VCh-rf has similar flavor to that for VCh, but relies on a different frontier graph.

Frontier graph for VCh-rf. The reads-from frontier graph $G^{\mathsf{rf}}_{\mathsf{frontier}}$ of $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ is slightly different from G_{frontier}. First, for a node $v = \langle Y, Q, I\rangle$, the set of unmatched send events buffered in $Q(\mathsf{ch})$ and I is already determined by Y and rf. Therefore, we only need to consider the permutations of these events in $Q(\mathsf{ch})$. Moreover, for an edge $v_1 \xrightarrow{e} v_2$ labeled with a receive event $e = \mathtt{rcv}(\mathsf{ch})$ over an asynchronous (resp. synchronous) channel ch, we require that the first entry $f = v_1.Q(\mathsf{ch})$ (resp. unique element $f = v_1.I$) is such that $(e, f) \in \mathsf{rf}$. The following lemma states how $G^{\mathsf{rf}}_{\mathsf{frontier}}$ captures the consistency of $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$.

Lemma 3.2. *$\langle \mathcal{X}, \mathsf{cap}, rf\rangle$ is consistent iff there is a sink node reachable from the source node in $G^{rf}_{\mathsf{frontier}}$.*

Time complexity for VCh-rf. For each node, the set Y, together with rf, uniquely determine send events that are unmatched, giving us a better bound on the number of possible values for the Q and I components of the node. The number of distinct Y sets is still (n^t/t^t). For each Y, I is uniquely determined by Y and rf. Likewise, the set of events in $Q(\mathsf{ch})$ for an asynchronous channel is the set of unmatched send events in Y, whose size is bounded by $\mathsf{cap}(\mathsf{ch}) \leq k$. The total number of permutations for $Q(\mathsf{ch})$ is thus $\leq \mathsf{cap}(\mathsf{ch})! \leq k!$ and is also $\leq t^k$ as argued before. Considering all m channels, the number of Q is bounded by $O((\min(k!, t^k)^m)$. In total, the number of nodes in the graph is $O(n^t/t^t \cdot (\min(k!, t^k)^m)$, while the number of edges is $O(n^t/t^t \cdot t \cdot (\min(k!, t^k)^m)$, thereby concluding Theorem 1.5.

3.2 VCh-rf with Synchronous Channels

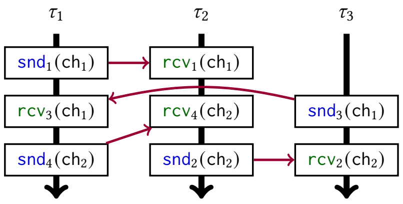
We now focus on VCh-rf in the case where all channels are synchronous and present a linear-time algorithm for Theorem 1.9. Previous work shows that purely synchronous communication enjoys some sort of "deterministic replay", which implies a $O(n \cdot t)$-time consistency algorithm [85]. Here we show that this setting admits a linear-time solution, irrespectively of the number of threads.

Our algorithm is based on the following insight. Since all channels are synchronous, every pair of events (snd, rcv) related by reads-from must execute consecutively. Our algorithm packs such event pairs in a single atomic event, and checks whether all atomic events can be scheduled in a way that respects partial order dependencies due to po. In turn, this reduces to checking for cycles in a suitably defined graph. We now make the above insight formal.

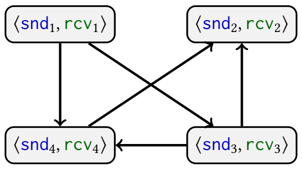
We assume wlog that the input instance $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$, where $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}\rangle$, is such that each send (resp. receive) event has exactly one receive (resp. send) event matched to it using rf, and the two events belong to different threads. Otherwise, the instance is clearly inconsistent.

The send-receive graph. The *send-receive graph* of $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ is a directed graph $G_{\mathsf{sync}} = (V, E)$ where V is the node set and E is the edge set, defined as follows. (1) $V \subseteq \mathsf{S} \times \mathsf{S}$ is the set of matching send and receive pairs, i.e., $\langle \mathsf{snd}, \mathsf{rcv}\rangle \in V$ iff $(\mathsf{snd}, \mathsf{rcv}) \in \mathsf{rf}$ (2) edges E capture po dependencies, i.e., $(\langle \mathsf{snd}_1, \mathsf{rcv}_1\rangle, \langle \mathsf{snd}_2, \mathsf{rcv}_2\rangle) \in E$ iff some $e_1 \in \{\mathsf{snd}_1, \mathsf{rcv}_1\}$ is the immediate po predecessor of some $e_2 \in \{\mathsf{snd}_2, \mathsf{rcv}_2\}$. See Figure 5 for an illustration. The send-receive graph precisely captures consistency, as stated in the following lemma.

Lemma 3.3. *$\langle \mathcal{X}, \mathsf{cap}, rf\rangle$ is consistent iff G_{sync} is acyclic.*


τ_1
τ_2
τ_3
$\mathsf{snd}_1(\mathsf{ch}_1)$
$\mathsf{rcv}_1(\mathsf{ch}_1)$
$\mathsf{rcv}_3(\mathsf{ch}_1)$
$\mathsf{rcv}_4(\mathsf{ch}_2)$
$\mathsf{snd}_3(\mathsf{ch}_1)$
$\mathsf{snd}_4(\mathsf{ch}_2)$
$\mathsf{snd}_2(\mathsf{ch}_2)$
$\mathsf{rcv}_2(\mathsf{ch}_2)$

(a) VCh-rf instance $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ with synchronous channels.


$\langle \mathsf{snd}_1, \mathsf{rcv}_1\rangle$
$\langle \mathsf{snd}_2, \mathsf{rcv}_2\rangle$
$\langle \mathsf{snd}_4, \mathsf{rcv}_4\rangle$
$\langle \mathsf{snd}_3, \mathsf{rcv}_3\rangle$

(b) The graph G_{sync}.

Fig. 5. A VCh-rf instance $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ (a) and the corresponding send-receive graph G_{sync} (b). As G_{sync} is acyclic, $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ is consistent.

Algorithm and time complexity. Following Lemma 3.3, the algorithm for checking VCh-rf when all channels are synchronous is straightforward — construct G_{sync} and check for acyclicity. For each pair $\langle \mathsf{snd}, \mathsf{rcv}\rangle$, there are at most two immediate po predecessors, so the in-degree of each node is at most 2. Therefore, G_{sync} has $O(n)$ nodes and $O(n)$ edges and the time to construct the graph is also $O(n)$. Checking for a cycle in G_{sync} also takes $O(n)$ time, which concludes the proof of Theorem 1.9.

3.3 Acyclic Communication Topologies

Finally, we turn our attention to acyclic communication topologies and prove that VCh-rf can be solved in quadratic time, establishing Theorem 1.7. We first formally define the communication topology of an abstract execution.

Communication topologies. A set of events S induces a communication topology, represented as an undirected graph $G = (V, E)$ where V is the set of threads appearing in S, and we have $(\tau_i, \tau_j) \in E$ iff τ_i and τ_j access a common channel, i.e., there exist two events $e_1, e_2 \in \mathsf{S}$ such that $\mathsf{th}(e_1) = \tau_i$, $\mathsf{th}(e_2) = \tau_j$, and $\mathsf{ch}(e_1) = \mathsf{ch}(e_2)$. The communication topology induced by an abstract execution $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}\rangle$ is the topology induced by its event set S.

Given two threads τ_i and τ_j, let $\mathsf{Channels}(\mathcal{X})\downharpoonright_{\tau_i,\tau_j}$ be the set of channels accessed by both τ_i, τ_j, and $\mathsf{cap}\downharpoonright_{\tau_i,\tau_j}$ be the restriction of the capacity function cap to the channels in $\mathsf{Channels}(\mathcal{X})\downharpoonright_{\tau_i,\tau_j}$. We define $\mathcal{X}\downharpoonright_{\tau_i,\tau_j}$ and $\mathsf{rf}\downharpoonright_{\tau_i,\tau_j}$ as the abstract execution obtained from $\mathcal{X}$ and reads-from relation obtained from rf by only keeping events from τ_i, τ_j that access a channel in $\mathsf{Channels}(\mathcal{X})\downharpoonright_{\tau_i,\tau_j}$.

Our proof of Theorem 1.7 is based on two key insights. First, we prove that VCh-rf on acyclic topologies is *compositional*: $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ is consistent iff $\langle \mathcal{X}\downharpoonleft_{\tau_i,\tau_j}, \mathsf{cap}\downharpoonleft_{\tau_i,\tau_j}, \mathsf{rf}\downharpoonleft_{\tau_i,\tau_j}\rangle$ is consistent, for every $(\tau_i, \tau_j) \in E$. Second, we show that VCh-rf over two threads is solvable in quadratic time, by a reduction to 2SAT on formulas of size quadratic in the size of the input.

Comparison with 2SAT encodings for shared-memory setting. The 2SAT encodings have also been explored in the shared-memory setting [24], but our work introduces several novel aspects. First, our encoding captures constraints unique to FIFO channel semantics and bounded capacities (including synchronous and capacity-1 channels), while naturally extending to unbounded channels without extra constraints. Second, our acyclic-topology result relies on a new compositionality lemma (Lemma 3.4), showing that any efficient solution for two threads extends compositionally to arbitrary acyclic systems, enabling direct performance gains from improved two-thread algorithms.

Compositionality. The compositionality lemma is formally stated as follows.

Lemma 3.4. *Let $\langle \mathcal{X}, \mathsf{cap}, rf\rangle$ be a VCh-rf instance, and $G = (V, E)$ the communication topology of $\mathcal{X}$ such that G is acyclic. Then $\langle \mathcal{X}, \mathsf{cap}, rf\rangle$ is consistent iff $\langle \mathcal{X}\downharpoonleft_{\tau_i,\tau_j},, \mathsf{cap}\downharpoonleft_{\tau_i,\tau_j}, rf\downharpoonleft_{\tau_i,\tau_j}\rangle$ is consistent, for every pair of threads $(\tau_i, \tau_j) \in E$.*

The intuition behind Lemma 3.4 is as follows. First, clearly for $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ to be consistent, we must have that $\langle \mathcal{X}\downharpoonleft_{\tau_i,\tau_j}, \mathsf{cap}\downharpoonleft_{\tau_i,\tau_j}, \mathsf{rf}\downharpoonleft_{\tau_i,\tau_j}\rangle$ is consistent for every two threads τ_i, τ_j. The other direction is more interesting. Consider a thread τ_1 with two neighbors τ_2, τ_3 in the communication topology, $(\tau_1, \tau_2), (\tau_1, \tau_3) \in E$, such that $\langle \mathcal{X}\downharpoonleft_{\tau_1,\tau_2}, \mathsf{cap}\downharpoonleft_{\tau_1,\tau_2}, \mathsf{rf}\downharpoonleft_{\tau_1,\tau_2}\rangle$ and $\langle \mathcal{X}\downharpoonleft_{\tau_1,\tau_3}, \mathsf{cap}\downharpoonleft_{\tau_1,\tau_3}, \mathsf{rf}\downharpoonleft_{\tau_1,\tau_3}\rangle$ are consistent, witnessed by the corresponding executions $\sigma_{1,2}$ and $\sigma_{1,3}$. Then we can interleave $\sigma_{1,2}$ and $\sigma_{1,3}$ in any way that respects the program order of thread τ_1, and the resulting execution σ_1 will be well-formed. This is because, owning to the acyclicity of G, we have $(\tau_2, \tau_3) \notin E$, meaning that τ_2 and τ_3 do not communicate over a common channel. In turn, this implies that the interleaving of events from τ_2 and τ_3 in σ cannot violate the well-formedness of σ_1. Composing all executions along edges of G in such a way results in an execution σ that witnesses the consistency of $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$.

The case of $t = 2$ threads. Given Lemma 3.4, we now focus on the case of VCh-rf over 2 threads, when every channel is capacity-unbounded, has capacity 1, or is synchronous (i.e., the setting captured in Theorem 1.7). We obtain a quadratic bound based on two insights. First, for each channel, channel-related constraints on the order of events accessing it can be encoded as 2SAT. The search for well-formed execution must also satisfy transitivity constraints, i.e., if $e_1 \longrightarrow e_2$ and $e_2 \longrightarrow e_3$, then $e_1 \longrightarrow e_3$. Transitivity involves three events, and thus does not immediately fit our 2SAT approach. Our second observation is that, with 2 threads, every three events e_1, e_2, e_3, must contain two events in the same thread, thus already ordered by po. Then, transitivity can be succinctly captured by a 2SAT formula as well. In the following we make these insights formal.

Consider a VCh-rf instance $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$ where $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}\rangle$ is an abstract execution involving two threads τ_1, τ_2. We construct a 2SAT formula $\varphi_{\langle \mathcal{X},\mathsf{cap},\mathsf{rf}\rangle}$ over propositional variables $x_{e,f}$, where $e, f \in \mathsf{S}$. Assigning $x_{e,f} = \top$ means ordering e before f in the execution witnessing the consistency of $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$. Overall, $\varphi_{\langle \mathcal{X},\mathsf{cap},\mathsf{rf}\rangle}$ is a conjunction of 8 subformulae:

$$\varphi_{\langle \mathcal{X},\mathsf{cap},\mathsf{rf}\rangle} \equiv \varphi_{\mathsf{exactly\text{-}1}} \land \varphi_{\mathsf{po}} \land \varphi_{\mathsf{rf}} \land \varphi_{\mathsf{unmatched}} \land \varphi_{\mathsf{FIFO}} \land \varphi_{\mathsf{trans}} \land \varphi_{\mathsf{cap=1}} \land \varphi_{\mathsf{sync}}$$

We now proceed with defining each subformula.

Exactly one. This formula requires that the order of two events must be resolved exactly in one way.

$$\varphi_{\mathsf{exactly\text{-}1}} \equiv \bigwedge_{e,f \in \mathsf{S}} \left(x_{e,f} \implies \neg x_{f,e}\right)$$

Program order. This formula requires that the order of two events must respect po.

Reads from. This formula requires that each receive event is ordered after its matched send event.

$$\varphi_{\mathsf{po}} \equiv \bigwedge_{(e,f)\in\mathsf{po}} x_{e,f} \quad \text{and} \quad \varphi_{\mathsf{rf}} \equiv \bigwedge_{(e,f)\in\mathsf{rf}} x_{e,f}$$

Unmatched sends. This formula requires that all unmatched send events are scheduled after all send events that have a matching receive event. Given a channel ch, let

$$\mathsf{Unmatched}_{\mathsf{ch}} = \{e \in \mathsf{S} \mid \mathsf{op}(e) = \mathsf{snd}, \mathsf{ch}(e) = \mathsf{ch}, \nexists f \text{ s.t. } (e,f) \in \mathsf{rf}\}, \text{ and}$$
$$\mathsf{Matched}_{\mathsf{ch}} = \{e \in \mathsf{S} \mid \mathsf{op}(e) = \mathsf{snd}, \mathsf{ch}(e) = \mathsf{ch}, \exists f \text{ s.t. } (e,f) \in \mathsf{rf}\}$$

denote the set of unmatched and matched send events, respectively. We have

$$\varphi_{\mathsf{unmatched}} \equiv \bigwedge_{\substack{\mathsf{ch}\in\mathsf{Channels}(\mathcal{X}),\, e\in\mathsf{Matched}_{\mathsf{ch}},\\ f\in\mathsf{Unmatched}_{\mathsf{ch}}}} x_{e,f}$$

FIFO. This formula requires that the order of two receive events on the same channel matches the order of the corresponding send events.

$$\varphi_{\mathsf{FIFO}} \equiv \bigwedge_{\substack{(e,e')\in\mathsf{rf},\,(f,f')\in\mathsf{rf}\\ e\neq f,\,\mathsf{ch}(e)=\mathsf{ch}(f)}} \left(\left(x_{e,f} \implies x_{e',f'}\right) \land \left(x_{e',f'} \implies x_{e,f}\right)\right)$$

Transitivity. This formula requires that the ordering of events is transitive. Let $\mathsf{pred}(e)$ (resp. $\mathsf{succ}(e)$) be the unique event (if one exists) that precedes (resp. succeeds) e in po. If $\mathsf{pred}(e)$ (resp. $\mathsf{succ}(e)$) doesn't exist, then $\mathsf{pred}(e) = \bot$ (resp. $\mathsf{succ}(e) = \bot$) . We have $\varphi_{\mathsf{trans}} \equiv \varphi^{\mathsf{pred}}_{\mathsf{trans}} \land \varphi^{\mathsf{succ}}_{\mathsf{trans}}$, where

$$\varphi^{\mathsf{pred}}_{\mathsf{trans}} \equiv \bigwedge_{e,f\in\mathsf{S},\, e'=\mathsf{pred}(e)\neq\bot} \left(x_{e,f} \implies x_{e',f}\right) \qquad \varphi^{\mathsf{succ}}_{\mathsf{trans}} \equiv \bigwedge_{e,f\in\mathsf{S},\, f'=\mathsf{succ}(f)\neq\bot} \left(x_{e,f} \implies x_{e,f'}\right)$$

Capacity. This formula requires that the capacity constraints of channels ch with $\mathsf{cap}(\mathsf{ch}) \leq 1$ are met. In particular, for two different send events $\mathsf{snd}_1(\mathsf{ch}) \neq \mathsf{snd}_2(\mathsf{ch})$, the matching receive event of the earlier send event also precedes the other send event. For a synchronous channel, we encode the fact that send and receive events are consecutive. For asynchronous channels that are capacity-unbounded, we do not need any capacity constraint.

$$\varphi_{\mathsf{cap}=1} \equiv \bigwedge_{\substack{(e,f)\in\mathsf{rf},\, e'\in\mathsf{S},\,\mathsf{op}(e)=\mathsf{op}(e')=\mathsf{snd}\\ \mathsf{ch}(e)=\mathsf{ch}(e') \text{ is asynchronous}}} \left(x_{e,e'} \implies x_{f,e'}\right)$$
$$\varphi_{\mathsf{sync}} \equiv \bigwedge_{\substack{(e,f)\in\mathsf{rf},\,\mathsf{ch}(e) \text{ is synchronous}\\ e'=\mathsf{succ}(e),\, f'=\mathsf{pred}(f)}} \left(x_{f,e'} \land x_{f',e}\right)$$

The following lemma states the correctness of the encoding.

Lemma 3.5. *$\langle \mathcal{X}, \mathsf{cap}, rf\rangle$ is consistent iff $\varphi_{\langle \mathcal{X},\mathsf{cap},rf\rangle}$ is satisfiable.*

Finally, observe that the number of propositional variables $x_{e,f}$ is bounded by n^2, while the number of clauses is also $O(n^2)$. Since 2SAT is solvable in time that is linear in the size of the formula [12], together with Lemma 3.5, we arrive at an algorithm that solves VCh-rf for 2 threads in $O(n^2)$ time.

Acyclic topologies. We now have all the ingredients to solve VCh-rf on acyclic communication topologies. Given an input $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf}\rangle$, the algorithm iterates over all edges (τ_i, τ_j) of the communication topology of $\mathcal{X}$, and uses the 2SAT encoding to decide the consistency of $\langle \mathcal{X}\downharpoonright_{\tau_i,\tau_j}, \mathsf{cap}\downharpoonright_{\tau_i,\tau_j}, \mathsf{rf}\downharpoonright_{\tau_i,\tau_j}\rangle$.

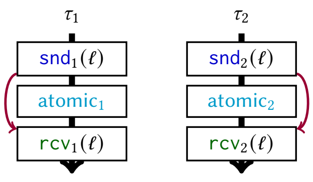
For analyzing the time complexity, observe that every two events $e, f \in \mathsf{S}$ appear in some propositional variable $x_{e,f}$ of at most one 2SAT instance. In particular, let $\tau_1 = \mathsf{th}(e)$ and $\tau_2 = \mathsf{th}(f)$. If $\tau_1 \neq \tau_2$, then $x_{e,f}$ appears in the 2SAT instance of the topology edge (τ_1, τ_2). On the other hand, if $\tau_1 = \tau_2 = \tau$, then $x_{e,f}$ appears in the 2SAT instance of the topology edge (τ, τ'), where τ' is the unique thread accessing the channels that e and f operate. We thus arrive at Theorem 1.7.

4 The Hardness of Verifying Channel Consistency with Reads From

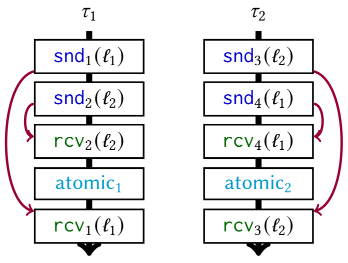
We now present some of the hardness results for VCh-rf. We first present how channels can be used to encode *atomicity gadgets* – that is, how to execute a sequence of events from the same thread without interleaving with other threads, as this encoding will be required in later reductions. We then show that the problem is intractable for case (i) and (iii) stated in Theorem 1.6 in Section 4.2 and Section 4.3). In Section 4.4, we prove the quadratic lower bound of VCh-rf on 2 threads, as stated in Theorem 1.8. The other lower bounds of VCh and VCh-rf stated in Theorem 1.1, Theorem 1.2, Theorem 1.3, Theorem 1.6, are proven with reductions of similar flavor, and appear in our companion technical report [81] due to space limits.

4.1 Atomicity Gadgets

Our reductions in later sections make use of *atomic blocks* of events as gadgets. An atomic block atomic in a thread is a sequence of events such that any two such blocks atomic_1, atomic_2 cannot overlap in any concretization. Here we show how to construct atomicity gadgets, both by using channels with capacity 1, and by using channels with unbounded capacity. The latter might sound counter-intuitive, in the sense that send operations to capacity-unbounded channels never block.



(a) Atomicity using capacity 1 channel.



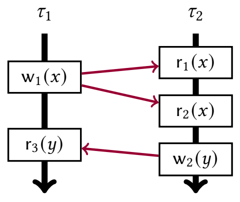
(b) Atomicity using two capacity-unbounded channels ℓ_1, ℓ_2

Fig. 6. Gadgets for implementing atomic blocks using capacity 1 (a) or unbounded-capacity channels (b). Reads-from edges are represented by arrows.

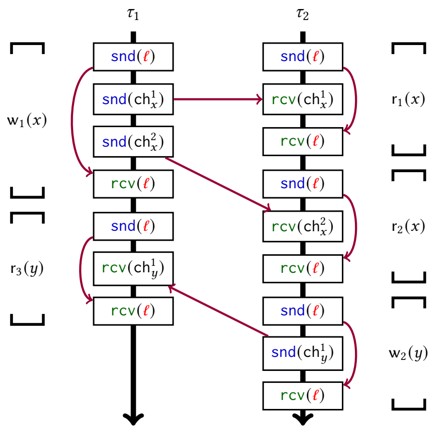
Atomicity with capacity 1. The atomicity gadget relying on channels of capacity 1 is shown in Figure 6a, using one channel ℓ, which resembles a lock. The thread that sends to ℓ first fills the channel capacity, and must execute the corresponding receive before the other thread can send to the channel. The events between the first send and receive are thus executed atomically.

Atomicity with unbounded capacity. The atomicity gadget using unbounded channels is shown in Figure 6b, relying on two channels ℓ_1 and ℓ_2. Its principle of operation is as follows. If $\mathsf{snd}_1(\ell_1)$ is executed before $\mathsf{snd}_4(\ell_1)$, then $\mathsf{rcv}_1(\ell_1)$ is also executed before $\mathsf{rcv}_4(\ell_1)$, making the atomic section of the first thread execute before the second. The inverse order is imposed if $\mathsf{snd}_4(\ell_1)$ is executed before $\mathsf{snd}_1(\ell_1)$, as this orders $\mathsf{snd}_3(\ell_2)$ before $\mathsf{snd}_2(\ell_2)$, and the argument repeats.

We note that the atomicity gadgets can be generalized to an arbitrary number of threads. For brevity, Figure 6 illustrates the case for two threads only.



(a) A VSC-read instance.



(b) The corresponding VCh-rf instance.

Fig. 7. A VSC-read instance (a) and the corresponding VCh-rf instance (b) with channel capacities of 1.

4.2 Hardness with Asynchronous Channels of Capacity 1

We establish a reduction from the VSC-read problem [42]. An instance of the VSC-read problem is a tuple $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}, \mathsf{rf} \rangle$, where S is a set of events of the form $\langle \tau, \mathsf{r}(x) \rangle$ or $\langle \tau, \mathsf{w}(x) \rangle$, in which τ is a thread identifier and x is a memory location, po is the per-thread total order (a.k.a program order) and rf maps each read event to a write event of the same register. Such an instance is sequentially consistent (SC) if there is a total order over S that respects po and rf, and ensures that for every $(e, f) \in \mathsf{rf}$ pair on register x, there is no other $\mathsf{w}(x)$ event ordered between e and f.

Overview. Let $\mathcal{X} = \langle \mathsf{S}, \mathsf{po}, \mathsf{rf} \rangle$ be an instance of VSC-read. We construct an instance $\langle \mathcal{X}', \mathsf{cap}', \mathsf{rf}' \rangle$ of VCh-rf, where $\mathcal{X}' = \langle \mathsf{S}', \mathsf{po}' \rangle$. At a high level, each write event (and each read event) in $\mathcal{X}$ is mapped to a sequence of send and receive instructions in $\mathcal{X}'$ that essentially appear atomically in every concretization. Further the reads-from relation of $\mathcal{X}$ is also accurately reflected in $\mathcal{X}'$ through reads-from on channels.

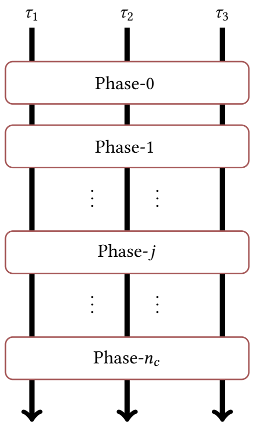
Reduction. Figure 7 illustrates the reduction on a small example. The set of threads in $\mathcal{X}'$ is the same as $\mathcal{X}$. The set of channels used in $\mathcal{X}'$ is $\{\mathsf{ch}_x^i \mid x \in \mathcal{R}, 1 \le i \le m_x\} \uplus \{\ell\}$, where $\mathcal{R}$ is the set of registers accessed in $\mathcal{X}$, $m_x = \max\{p_e \mid e \text{ is a write on } x\}$ and p_e is the number of read events f with $(e, f) \in \mathsf{rf}$. The capacity function cap assigns capacity 1 to every channel. At a high level, the thread-wise event sequences in $\mathcal{X}'$ are structurally similar to those in $\mathcal{X}$, and can be characterized using a map M that maps events in S to distinct atomic, thread-local sequences of events in S′, so that $\mathsf{S}' = \bigcup_{e \in \mathsf{S}} \{f \mid f \in M(e)\}$. Atomicity is guaranteed by channel ℓ with capacity 1. In Section 4.1, we have detailed an explanation about atomicity gadgets. We now describe the map M.

For a write event $e = \langle t, \mathsf{w}(x) \rangle$, $M(e)$ is a sequence of m_x-many snd events, followed by $m_x - p_x$ rcv events, all enclosed in a block of send-receive pair on channel ℓ; the thread identifier of each of the following event is τ, and we omit explicitly mentioning it.

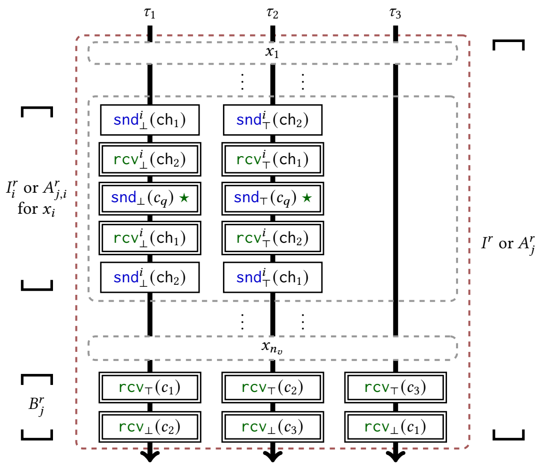
$$M(e) = \mathsf{snd}(\ell) \cdot \mathsf{snd}(\mathsf{ch}_x^1) \cdots \mathsf{snd}(\mathsf{ch}_x^{m_x}) \cdot \mathsf{rcv}(\mathsf{ch}_x^{p_e+1}) \cdots \mathsf{rcv}(\mathsf{ch}_x^{m_x}) \cdot \mathsf{rcv}(\ell)$$

Let us now discuss the encoding of read events. For this, we assume some arbitrary ordering $\{f_1, f_2, \ldots, f_{p_e}\}$ of the set of read events reading from some write event e. Then, the event sequence corresponding to the ith read event $e = \langle \tau, \mathsf{r}(x) \rangle$ of some write event is:

$$M(e) = \mathsf{snd}(\ell) \cdot \mathsf{rcv}(\mathsf{ch}_x^i) \cdot \mathsf{rcv}(\ell)$$



(a) Overall scheme


τ_1
τ_2
τ_3
x_1
I^r_i or $A^r_{j,i}$ for x_i
$\mathsf{snd}^i_\perp(\mathsf{ch}_1)$
$\mathsf{snd}^i_\top(\mathsf{ch}_2)$
$\mathsf{rcv}^i_\perp(\mathsf{ch}_2)$
$\mathsf{rcv}^i_\top(\mathsf{ch}_1)$
$\mathsf{snd}_\perp(c_q)$ ★
$\mathsf{snd}_\top(c_q)$ ★
$\mathsf{rcv}^i_\perp(\mathsf{ch}_1)$
$\mathsf{rcv}^i_\top(\mathsf{ch}_2)$
$\mathsf{snd}^i_\perp(\mathsf{ch}_2)$
$\mathsf{snd}^i_\top(\mathsf{ch}_1)$
I^r or A^r_j
x_{n_v}
B^r_j
$\mathsf{rcv}_\top(c_1)$
$\mathsf{rcv}_\top(c_2)$
$\mathsf{rcv}_\top(c_3)$
$\mathsf{rcv}_\perp(c_2)$
$\mathsf{rcv}_\perp(c_3)$
$\mathsf{rcv}_\perp(c_1)$

(b) Events in Phase-j

Fig. 8. Reduction from 3SAT to VCh-rf with capacity-unbounded channels. Events with double boundary do not appear in Phase-0. Events marked with ★ only appear when the q^{th} literal in clause C_j is over variable x_i

The program order po′ is then obtained by considering all pairs of events of the form (e_1, e_2) in S′ such that either they belong to $M(e)$ for some e and e_1 appears before e_2 in $M(e)$, or they belong to $M(e)$ and $M(e')$ respectively with $(e, e') \in$ po. The rf′ relation is also straightforward. For each event of the form $\mathsf{rcv}(\ell)$ in $M(e)$, its corresponding send event is the unique $\mathsf{snd}(\ell)$ event in $M(e)$. The send and receive events on channels of the form ch^i_x are paired as follows. Let $(e, f_i) \in$ rf be a pair of write and its i^{th} read event in S. Then the send event $e'_i = \mathsf{snd}(\mathsf{ch}^i_x)$ in $M(e)$ is paired to the event $f'_i = \mathsf{rcv}(\mathsf{snd}(\mathsf{ch}^i_x))$ in $M(f_i)$ (i.e., $(e'_i, f'_i) \in$ rf′). Further, the unmatched send event $e'_j = \mathsf{snd}(\mathsf{ch}^j_x)$ in $M(e)$, where $p_e + 1 \le j \le m_x$ is paired with the $(j - p_e)^{\text{th}}$ receive event $f'_j = \mathsf{rcv}(\mathsf{ch}^j_x)$ in $M(e)$, (i.e., $(e'_j, f'_j) \in$ rf′).

The correctness of the construction is relatively straightforward, and stated in the following lemma.

Lemma 4.1. *$\mathcal{X}$ is SC consistent iff $\langle \mathcal{X}', \mathsf{cap}', \mathit{rf}' \rangle$ is consistent.*

We now argue about the time taken to construct $\langle \mathcal{X}', \mathsf{cap}', \mathsf{rf}' \rangle$. Each write event in S can be observed by at most |S| different read events. Each $e \in$ S is thus mapped to a sequence consisting of $O(|\mathsf{S}|)$ events. Thus, $|\mathsf{S}'| \in O(|\mathsf{S}^2|)$, which concludes case (i) of Theorem 1.6.

4.3 Hardness with 3 Threads, 5 Channels and no Capacity Restrictions

We show that VCh-rf remains intractable when both the number of threads and channels are constant, and there are no restrictions on channel capacities.

Overview. Starting from a 3SAT instance ψ with n_c clauses $C_1, \dots, C_{n_c}$ over n_v propositional variables $\{x_1, \dots, x_{n_v}\}$, we construct a VCh-rf instance $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf} \rangle$ with 3 threads τ_1, τ_2, τ_3 and 5 channels $\mathsf{ch}_1, \mathsf{ch}_2, c_1, c_2, c_3$. Informally, $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf} \rangle$ consists of $n_c + 1$ phases, arranged sequentially. The first *initialization* phase ('Phase-0') picks an assignment of boolean values for each variable. The remaining n_c phases encode the requirement that at least one literal from each clause is set to true. Phase-j (with $j \ge 1$) duplicates the assignment to all variables from the previous phase and checks if the chosen assignment makes clause C_j true. Figure 8 depicts this scheme.

Reduction. The sequence σ_r corresponding to events of thread τ_r ($r \in \{1, 2, 3\}$) is of the form $\sigma_r = I^r \cdot A^r_1 \cdot A^r_2 \cdots A^r_{n_c}$. The sequence corresponding to Phase-0 is of the form $I^r = I^r_1 \cdots I^r_{n_v}$, where

I^r_p picks an assignment to variable x_p in thread τ_r:

$$I^1_p = \mathsf{snd}^p_\bot(\mathsf{ch}_1) \cdot \mathsf{snd}^p_\bot(\mathsf{ch}_2) \qquad I^2_p = \mathsf{snd}^p_\top(\mathsf{ch}_2) \cdot \mathsf{snd}^p_\top(\mathsf{ch}_1) \qquad I^3_p = \epsilon$$

Next, the sequence corresponding to thread τ_r and Phase-j ($j \geq 1$) is of the form $A^r_j = A^r_{j,1} \cdots A^r_{j,n_v} \cdot B^r_j$. where $A^r_{j,p}$ corresponds to variable x_p and B^r_j encodes the satisfaction of clause C_j (see Figure 8 for illustration). We describe these components next. $A^3_{j,p} = \epsilon$ for every $j \in \{1, \dots, n_c\}$, $p \in \{1, \dots, n_v\}$. When $r \in \{1, 2\}$, then $A^r_{j,p}$ is used to encode the variable x_p in clause C_j of thread τ_r. If x_p appears in clause C_j and it is the p^{th} literal of C_j (p $\in \{1, 2, 3\}$), then:

$$\begin{aligned} A^1_{j,p} &= \mathsf{snd}^p_\bot(\mathsf{ch}_1) \cdot \mathsf{rcv}^p_\bot(\mathsf{ch}_2) \cdot \mathsf{snd}_\bot(c_q) \cdot \mathsf{rcv}^p_\bot(\mathsf{ch}_1) \cdot \mathsf{snd}^p_\bot(\mathsf{ch}_2) \\ A^2_{j,p} &= \mathsf{snd}^p_\top(\mathsf{ch}_2) \cdot \mathsf{rcv}^p_\top(\mathsf{ch}_1) \cdot \mathsf{snd}_\top(c_q) \cdot \mathsf{rcv}^p_\top(\mathsf{ch}_2) \cdot \mathsf{snd}^p_\top(\mathsf{ch}_1) \end{aligned}$$

If x_p is not in clause C_j, then:

$$\begin{aligned} A^1_{j,p} &= \mathsf{snd}^p_\bot(\mathsf{ch}_1) \cdot \mathsf{rcv}^p_\bot(\mathsf{ch}_2) \cdot \mathsf{rcv}^p_\bot(\mathsf{ch}_1) \cdot \mathsf{snd}^p_\bot(\mathsf{ch}_2) \\ A^2_{j,p} &= \mathsf{snd}^p_\top(\mathsf{ch}_2) \cdot \mathsf{rcv}^p_\top(\mathsf{ch}_1) \cdot \mathsf{rcv}^p_\top(\mathsf{ch}_2) \cdot \mathsf{snd}^p_\top(\mathsf{ch}_1) \end{aligned}$$

Finally,

$$B^1_j = \mathsf{rcv}_\top(c_1) \cdot \mathsf{rcv}_\bot(c_2) \qquad B^2_j = \mathsf{rcv}_\top(c_2) \cdot \mathsf{rcv}_\bot(c_3) \qquad B^3_j = \mathsf{rcv}_\top(c_3) \cdot \mathsf{rcv}_\bot(c_1)$$

We now discuss the reads-from mappings.

- The receive events $\mathsf{rcv}^p_\bot(\mathsf{ch}_2)$, $\mathsf{rcv}^p_\bot(\mathsf{ch}_1)$, $\mathsf{rcv}^p_\top(\mathsf{ch}_2)$ and $\mathsf{rcv}^p_\top(\mathsf{ch}_1)$ in $A^1_{j,p}, A^1_{j,p}, A^2_{j,p}, A^2_{j,p}$ are respectively mapped to the send events $\mathsf{snd}^p_\bot(\mathsf{ch}_2)$, $\mathsf{snd}^p_\bot(\mathsf{ch}_1)$, $\mathsf{snd}^p_\top(\mathsf{ch}_2)$, $\mathsf{snd}^p_\top(\mathsf{ch}_1)$ in $A^1_{j-1,p}, A^1_{j-1,p}, A^2_{j-1,p}, A^2_{j-1,p}$ (or in $I^1_p, I^1_p, I^2_p, I^2_p$ if $j = 1$).
- Let $C_j = \gamma_1 \vee \gamma_2 \vee \gamma_3$ such that γ_q is either x_{j_q} or $\neg x_{j_q}$. For each $q \in \{1, 2, 3\}$, we have the following. If $\gamma_q = x_{j_q}$, then we require that the receive event $\mathsf{rcv}_\top(c_q)$ reads from send $\mathsf{snd}_\top(c_q)$ in A^2_{j,j_q}, and $\mathsf{rcv}_\bot(c_q)$ reads from $\mathsf{snd}_\top(c_q)$ in A^1_{j,j_q}. Otherwise, we require that $\mathsf{rcv}_\top(c_q)$ reads from $\mathsf{snd}_\bot(c_q)$ in A^1_{j,j_q} and $\mathsf{rcv}_\top(c_q)$ reads from $\mathsf{snd}_\top(c_q)$ in A^2_{j,j_q}.

The following lemma states the correctness of the above construction.

Lemma 4.2. *ψ is satisfiable iff $\langle \mathcal{X}, \mathsf{cap}, rf \rangle$ is consistent.*

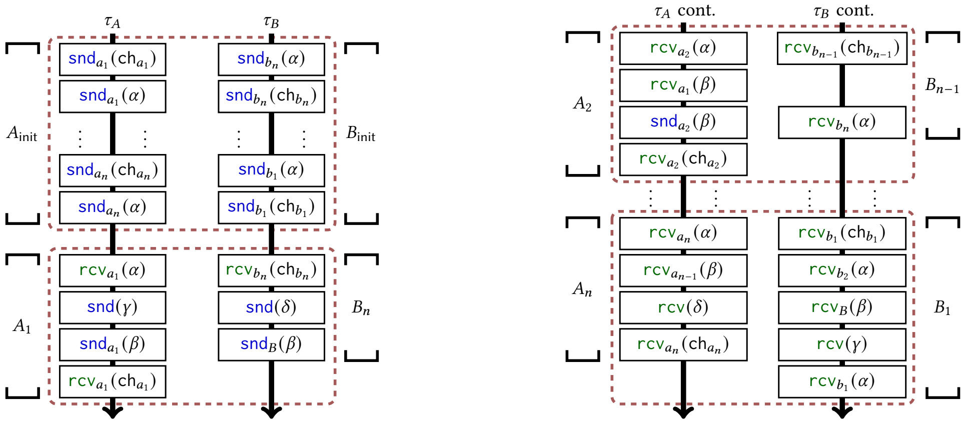
Finally, the number of events in $\langle \mathcal{X}, \mathsf{cap}, \mathrm{rf} \rangle$ is $O(n_v + n_c)$, which concludes case (iii) of Theorem 1.6.

4.4 Quadratic Hardness with 2 Threads

Finally, in this section we prove the quadratic hardness of VCh-rf over just 2 threads when either all channels have capacity 1 or have no capacity restrictions. We achieve this by establishing a fine-grained reduction from the Orthogonal Vectors problem (OV) [92].

The Orthogonal Vectors problem. The OV problem takes as input two sets $A = \{a_1, a_2 \dots, a_n\}, B = \{b_1, b_2 \dots, b_n\} \subseteq 2^{\{0,1\}^d}$, each containing n boolean vectors in d dimensions. The task is to determine whether there are two vectors $a \in A, b \in B$ such that a and b are orthgonal, i.e., $\langle a \cdot b \rangle = \sum_{i=1}^d a[i] \cdot b[i] = 0$. Under the SETH, OV cannot be solved in time $O(n^{2-\epsilon})$, for every fixed $\epsilon > 0$, as long as $d = \omega(\log n)$ [92].

Overview. We construct a VCh-rf instance $\langle \mathcal{X}, \mathsf{cap}, \mathrm{rf} \rangle$ which is consistent iff A and B contain an orthogonal vector pair. $\mathcal{X}$ comprises two threads τ_A and τ_B, respectively containing events encoding the vectors of A and B. Figure 9 illustrates the overall scheme. At a high level, the reads-from edge due to the pair $\langle \mathsf{snd}(\gamma), \mathsf{rcv}(\gamma) \rangle \in \mathrm{rf}$ triggers an orthogonality check between the vectors a_1 and b_1. The reduction is built in such a way that this process of inference, called *saturation* (formally


τ_A
τ_B
A_{init}
B_{init}
$\mathsf{snd}_{a_1}(\mathsf{ch}_{a_1})$
$\mathsf{snd}_{a_1}(\alpha)$
$\mathsf{snd}_{a_n}(\mathsf{ch}_{a_n})$
$\mathsf{snd}_{a_n}(\alpha)$
$\mathsf{snd}_{b_n}(\alpha)$
$\mathsf{snd}_{b_n}(\mathsf{ch}_{b_n})$
$\mathsf{snd}_{b_1}(\alpha)$
$\mathsf{snd}_{b_1}(\mathsf{ch}_{b_1})$
A_1
B_n
$\mathsf{rcv}_{a_1}(\alpha)$
$\mathsf{snd}(\gamma)$
$\mathsf{snd}_{a_1}(\beta)$
$\mathsf{rcv}_{a_1}(\mathsf{ch}_{a_1})$
$\mathsf{rcv}_{b_n}(\mathsf{ch}_{b_n})$
$\mathsf{snd}(\delta)$
$\mathsf{snd}_B(\beta)$
τ_A cont.
τ_B cont.
A_2
B_{n-1}
$\mathsf{rcv}_{a_2}(\alpha)$
$\mathsf{rcv}_{a_1}(\beta)$
$\mathsf{snd}_{a_2}(\beta)$
$\mathsf{rcv}_{a_2}(\mathsf{ch}_{a_2})$
$\mathsf{rcv}_{b_{n-1}}(\mathsf{ch}_{b_{n-1}})$
$\mathsf{rcv}_{b_n}(\alpha)$
A_n
B_1
$\mathsf{rcv}_{a_n}(\alpha)$
$\mathsf{rcv}_{a_{n-1}}(\beta)$
$\mathsf{rcv}(\delta)$
$\mathsf{rcv}_{a_n}(\mathsf{ch}_{a_n})$
$\mathsf{rcv}_{b_1}(\mathsf{ch}_{b_1})$
$\mathsf{rcv}_{b_2}(\alpha)$
$\mathsf{rcv}_B(\beta)$
$\mathsf{rcv}(\gamma)$
$\mathsf{rcv}_{b_1}(\alpha)$

(a) Events for $A_{\text{init}}, B_{\text{init}}, A_1, B_n$ (b) Events for A_i, B_{n-i} $(i \geq 2)$

Fig. 9. General scheme of the reduction from Orthogonal Vectors to VCh-rf with unbounded channels under two threads. Send/receive events on the same channel and with the same subscript are related by rf.

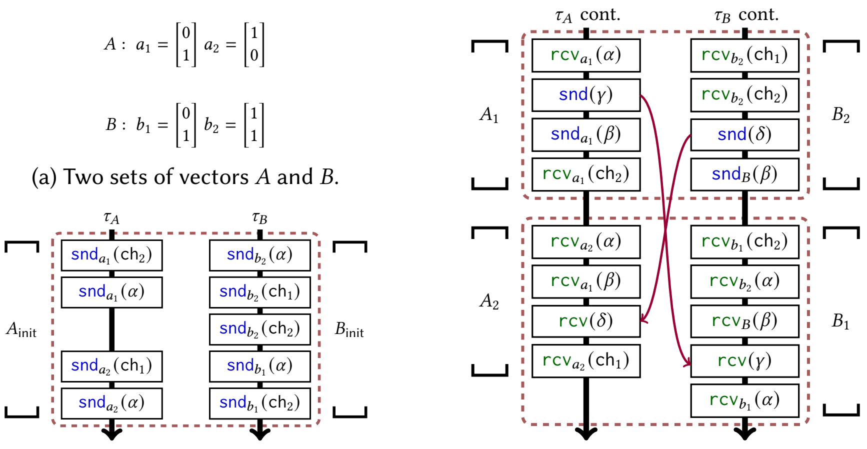
described in Section 5.1), simulates orthogonality comparisons of the vectors. If $a_1[i] = b_1[i] = 1$ for some i, witnessing that a_1 and b_1 are not orthogonal, the corresponding events encoding a_1 and b_1 will contain two sends on the same channel, which triggers an orthogonality check between a_1 and b_2. If a_1 and b_2 are also not orthogonal, then a_1 and b_3 are compared, and so on. If the check between a_1 and b_n fails, this triggers the check between a_2 and b_1, and the process continues, until an orthogonal pair is found, or the check between a_n and b_n does not identify an orthogonal pair. The fact that a_n, b_n are not orthogonal implies $\mathsf{rcv}(\delta)$ must be ordered before $\mathsf{snd}(\delta)$, which contradicts with $\langle \mathsf{snd}(\delta), \mathsf{rcv}(\delta)\rangle \in \mathsf{rf}$, implying that the constructed instance is not consistent. We now formally describe the reduction. For simplicity, for certain events e, we use subscripts to indicate the specific vector that e represents.

Reduction for capacity-unbounded channels. Given the OV instance A, B, we construct the corresponding VCh-rf instance using two threads τ_A and τ_B and channels $\{\mathsf{ch}_1, \mathsf{ch}_2, \ldots, \mathsf{ch}_d, \alpha, \beta, \gamma, \delta\}$, all having unbounded capacity. We describe the events next, while using subscripts in the event operations that ensure that the combination of the operation, the subscript and the channel uniquely identify each event. Send and receive events on the same channel that also have the same subscript are implicitly related by rf. The events of threads τ_A and τ_B are organized as follows:

$$\tau_A = A_{\text{init}} \cdot A_1 \cdot A_2 \cdots A_n \qquad \text{and} \qquad \tau_B = B_{\text{init}} \cdot B_n \cdot B_{n-1} \cdots B_1$$

Observe that the order of appearance of $A_1, \ldots, A_n$ is the reverse of that of $B_n, \ldots, B_1$. We next describe the contents of each block. We use the notation $\mathsf{snd}_{a_i}(\mathsf{ch}_{a_i})$ to denote the sequence $\mathsf{snd}_{a_i}(\mathsf{ch}_{j_1}) \cdot \mathsf{snd}_{a_i}(\mathsf{ch}_{j_2}) \cdots \mathsf{snd}_{a_i}(\mathsf{ch}_{j_k})$, where $j_1, j_2, \ldots, j_k$ is the unique increasing sequence of indices in $\{1, 2, \ldots, d\}$ corresponding to non-zero entries in the vector a_i. Likewise, $\mathsf{snd}_{b_i}(\mathsf{ch}_{b_i})$, $\mathsf{rcv}_{a_i}(\mathsf{ch}_{a_i})$ and $\mathsf{rcv}_{b_i}(\mathsf{ch}_{b_i})$ expand in a similar fashion. The init block in τ_A contains send events for each vector $a \in A$ (on all those channels ch_i such that $a[i] = 1$) with alternating send events on channel α, and likewise in τ_B (but in reverse order):

$$\begin{aligned} A_{\text{init}} &= \mathsf{snd}_{a_1}(\mathsf{ch}_{a_1}) \cdot \mathsf{snd}_{a_1}(\alpha) \cdots \mathsf{snd}_{a_n}(\mathsf{ch}_{a_n}) \cdot \mathsf{snd}_{a_n}(\alpha) \\ B_{\text{init}} &= \mathsf{snd}_{b_n}(\alpha) \cdot \mathsf{snd}_{b_n}(\mathsf{ch}_{b_n}) \cdots \mathsf{snd}_{b_1}(\alpha) \cdot \mathsf{snd}_{b_1}(\mathsf{ch}_{b_1}) \end{aligned}$$


$A: a_1 = \begin{bmatrix}0\\1\end{bmatrix} a_2 = \begin{bmatrix}1\\0\end{bmatrix}$
$B: b_1 = \begin{bmatrix}0\\1\end{bmatrix} b_2 = \begin{bmatrix}1\\1\end{bmatrix}$
τ_A
τ_B
A_{init}
B_{init}
$\mathsf{snd}_{a_1}(\mathsf{ch}_2)$
$\mathsf{snd}_{a_1}(\alpha)$
$\mathsf{snd}_{a_2}(\mathsf{ch}_1)$
$\mathsf{snd}_{a_2}(\alpha)$
$\mathsf{snd}_{b_2}(\alpha)$
$\mathsf{snd}_{b_2}(\mathsf{ch}_1)$
$\mathsf{snd}_{b_2}(\mathsf{ch}_2)$
$\mathsf{snd}_{b_1}(\alpha)$
$\mathsf{snd}_{b_1}(\mathsf{ch}_2)$
τ_A cont.
τ_B cont.
A_1
A_2
B_2
B_1
$\mathsf{rcv}_{a_1}(\alpha)$
$\mathsf{snd}(\gamma)$
$\mathsf{snd}_{a_1}(\beta)$
$\mathsf{rcv}_{a_1}(\mathsf{ch}_2)$
$\mathsf{rcv}_{a_2}(\alpha)$
$\mathsf{rcv}_{a_1}(\beta)$
$\mathsf{rcv}(\delta)$
$\mathsf{rcv}_{a_2}(\mathsf{ch}_1)$
$\mathsf{rcv}_{b_2}(\mathsf{ch}_1)$
$\mathsf{rcv}_{b_2}(\mathsf{ch}_2)$
$\mathsf{snd}(\delta)$
$\mathsf{snd}_B(\beta)$
$\mathsf{rcv}_{b_1}(\mathsf{ch}_2)$
$\mathsf{rcv}_{b_2}(\alpha)$
$\mathsf{rcv}_B(\beta)$
$\mathsf{rcv}(\gamma)$
$\mathsf{rcv}_{b_1}(\alpha)$

(a) Two sets of vectors A and B.

(b) Events for A_{init} and B_{init}.

(c) Events for A_1, A_2, B_1, and B_2.

Fig. 10. An example for the reduction from the Orthogonal Vectors problem to VCh-rf with unbounded channels under two threads. Only cross-thread reads-from edges are shown.

The extremal blocks (A_1, B_1, A_n, B_n) and internal blocks (A_i, B_i where $2 \le i \le n-1$) are as follows.

$$\begin{array}{rcl}
A_1 & = & \mathsf{rcv}_{a_1}(\alpha) \cdot \mathsf{snd}(\gamma) \cdot \mathsf{snd}_{a_1}(\beta) \cdot \mathsf{rcv}_{a_1}(\mathsf{ch}_{a_1}) \\
A_n & = & \mathsf{rcv}_{a_n}(\alpha) \cdot \mathsf{rcv}_{a_{n-1}}(\beta) \cdot \mathsf{rcv}(\delta) \cdot \mathsf{rcv}_{a_n}(\mathsf{ch}_{a_n}) \\
B_n & = & \mathsf{rcv}_{b_n}(\mathsf{ch}_{b_n}) \cdot \mathsf{snd}(\delta) \cdot \mathsf{snd}_B(\beta) \\
B_1 & = & \mathsf{rcv}_{b_1}(\mathsf{ch}_{b_1}) \cdot \mathsf{rcv}_{b_2}(\alpha) \cdot \mathsf{rcv}_B(\beta) \cdot \mathsf{rcv}(\gamma) \cdot \mathsf{rcv}_{b_1}(\alpha) \\
A_i & = & \mathsf{rcv}_{a_i}(\alpha) \cdot \mathsf{rcv}_{a_{i-1}}(\beta) \cdot \mathsf{snd}_{a_i}(\beta) \cdot \mathsf{rcv}_{a_i}(\mathsf{ch}_{a_i}) \\
B_i & = & \mathsf{rcv}_{b_i}(\mathsf{ch}_{b_i}) \cdot \mathsf{rcv}_{b_{i+1}}(\alpha)
\end{array}$$

The following lemma states the correctness of the construction.

Lemma 4.3. *$\langle \mathcal{X}, \mathsf{cap}, rf \rangle$ is consistent iff A and B contain an orthogonal vector pair.*

Regarding the time complexity, the reduction takes time proportional to $|A| + |B|$ i.e., $O(n \cdot d)$. Hence, a subquadratic algorithm for deciding the consistency of $\langle \mathcal{X}, \mathsf{cap}, \mathsf{rf} \rangle$ would imply a subquadratic algorithm for solving OV, thereby contradicting SETH. We thus arrive at case (i) of Theorem 1.8.

Example 4. *Figure 10 illustrates an example with $n = d = 2$. The OV instance consists of the two sets $A = \{a_1 = \langle 0, 1 \rangle, a_2 = \langle 1, 0 \rangle\}$ and $B = \{b_1 = \langle 0, 1 \rangle, b_2 = \langle 1, 1 \rangle\}$. Since a_2 and b_1 are orthogonal, the constructed instance $\Gamma = \langle \mathcal{X}, \mathsf{cap}, rf \rangle$ is consistent.*

We now explain how Γ encodes orthogonality checks between the vectors of A and B via saturation. Let us use $<_{\text{sat}}$ to denote the saturated order of Γ; its formal definition is presented in Section 5.1, though it can be intuitively understood to be a set of orderings between events of Γ that must hold in every consistent concretization of Γ, if one exists (Lemma 5.1). Since $(po \cup rf)^+ \subseteq <_{\text{sat}}$, we have $\mathsf{rcv}_{a_1}(\alpha) <_{\text{sat}} \mathsf{rcv}_{b_1}(\alpha)$, which means $\mathsf{snd}_{a_1}(\alpha) <_{\text{sat}} \mathsf{snd}_{b_1}(\alpha)$. This signifies that a_1 and b_1 are compared for orthogonality. Since a_1 and b_2 both have value 1 in dimension 2, they are not orthogonal. This is witnessed by $\mathsf{snd}_{a_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{snd}_{b_1}(\mathsf{ch}_2)$, further leading to $\mathsf{rcv}_{a_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{rcv}_{b_1}(\mathsf{ch}_2)$. Due to po, we also have $\mathsf{rcv}_{a_1}(\alpha) <_{\text{sat}} \mathsf{rcv}_{a_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{rcv}_{b_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{rcv}_{b_2}(\alpha)$, and therefore $\mathsf{snd}_{a_1}(\alpha) <_{\text{sat}} \mathsf{snd}_{b_2}(\alpha)$, which means that a_1 and b_2 are now compared for orthogonality. As before, a_1 and b_2 are not orthogonal, so we get the following sequence of inferences:

1. $\mathsf{snd}_{a_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{snd}_{a_1}(\alpha) <_{\text{sat}} \mathsf{snd}_{b_2}(\alpha) <_{\text{sat}} \mathsf{snd}_{b_2}(\mathsf{ch}_2) \implies \mathsf{rcv}_{a_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{rcv}_{b_2}(\mathsf{ch}_2)$

2. $\mathsf{snd}_{a_1}(\beta) <_{\text{sat}} \mathsf{rcv}_{a_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{rcv}_{b_2}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{snd}_B(\beta) \implies \mathsf{rcv}_{a_1}(\beta) <_{\text{sat}} \mathsf{rcv}_B(\beta)$
3. $\mathsf{rcv}_{a_2}(\alpha) <_{\text{sat}} \mathsf{rcv}_{a_1}(\beta) <_{\text{sat}} \mathsf{rcv}_b(\beta) <_{\text{sat}} \mathsf{rcv}_{b_1}(\alpha) \implies \mathsf{snd}_{a_2}(\alpha) <_{\text{sat}} \mathsf{snd}_{b_1}(\alpha)$

Notice that we now start to check orthogonality between a_2 and b_1.

As a_2 and b_1 are orthogonal, the sequences $\mathsf{ch}_{a_2}(\mathsf{ch}_{a_2})$ and $\mathsf{ch}_{b_1}(\mathsf{ch}_{b_1})$ do not share any channels. Therefore, saturation stops inferring orderings at this point. However, the receive on δ also implies orderings via saturation. In fact, this also leads to orthogonality comparisons, but in a reversed order: b_2 is compared with a_2, then b_1 with a_2, and so on. The following sequence of inferences illustrates this:

1. $\mathsf{rcv}_{b_2}(\mathsf{ch}_1) <_{\text{sat}} \mathsf{snd}(\delta) <_{\text{sat}} \mathsf{rcv}(\delta) <_{\text{sat}} \mathsf{rcv}_{a_2}(\mathsf{ch}_1) \implies \mathsf{snd}_{b_2}(\mathsf{ch}_1) <_{\text{sat}} \mathsf{snd}_{a_2}(\mathsf{ch}_1)$
2. $\mathsf{snd}_{b_2}(\alpha) <_{\text{sat}} \mathsf{snd}_{b_2}(\mathsf{ch}_1) <_{\text{sat}} \mathsf{snd}_{a_2}(\mathsf{ch}_1) <_{\text{sat}} \mathsf{snd}_{a_2}(\alpha) \implies \mathsf{rcv}_{b_2}(\alpha) <_{\text{sat}} \mathsf{rcv}_{a_2}(\alpha)$

The ordering of $\mathsf{rcv}_{b_2}(\alpha)$ before $\mathsf{rcv}_{a_2}(\alpha)$ is then what compares b_1 to a_2 (since $\mathsf{rcv}_{b_1}(\mathsf{ch}_2) <_{\text{sat}} \mathsf{rcv}_{b_2}(\alpha)$). Again, the orthogonality of a_1 and b_2 stops the saturation process.

We claim that the resulting $<_{\text{sat}}$ contains no cycle and is strong enough to fully sequentialize $\mathcal{X}$.

Channels with capacity 1. Finally, we argue about the quadratic hardness of VCh-rf when every channel has capacity 1. This result follows a recent result that verifying sequential consistency with a reads-from mapping (VSC-read problem) is OV-hard for 2 threads [66]. In Section 4.2, we have shown for any VSC-read instance with n events, we can construct an equivalent VCh-rf instance with $O(m_{\mathcal{R}} \cdot n)$ events, where $m_{\mathcal{R}}$ is the maximal number of read events that observe the same write event. Fortunately, in the reduction developed in [66], $m_{\mathcal{R}}$ is a constant, and therefore our VCh-rf instance is of linear size as the input VSC-read problem. Since VSC-read under two threads is OV-hard, VCh-rf cannot be solved in $O(n^{2-\epsilon})$ time for any $\epsilon > 0$, when there are 2 threads and every channel has capacity 1. Item (ii) of Theorem 1.8 is thus proven.

5 Evaluation

We have implemented our frontier graph algorithm for VCh-rf and evaluated its performance on 103 VCh-rf instances and compare against a constraint-solving approach for VCh-rf. Before we discuss our experimental setup (Section 5.2) and our evaluation results (Section 5.3 and Section 5.4), we describe a crucial optimization, *saturation* in Section 5.1.

5.1 Saturation

Saturation, widely used as a pre-processing step in consistency checking [96] and dynamic race detection [72], for shared register based concurrency, can be adapted to channel based concurrency. At a high level, saturation infers additional event orderings in polynomial time before the core consistency-checking algorithm executes. For a VCh-rf instance, saturation infers orderings beyond the input program order and reads-from relations. For example, if two send events on the same channel are ordered by po, then their corresponding receive events must also be ordered.

Formally, let $\Gamma = \langle \mathcal{X}, \mathsf{cap}, \mathsf{rf} \rangle$ be an input VCh-rf instance, where $\mathcal{X} = \langle \mathsf{S}, \mathsf{po} \rangle$. The saturated order $<_{\text{sat}}$ of Γ is defined to be the smallest transitive relation on events such that $(\mathsf{po} \cup \mathsf{rf})^+ \subseteq <_{\text{sat}}$ and:

1. For each channel ch, and pairs $(\mathsf{snd}_1(\mathsf{ch}), \mathsf{rcv}_1(\mathsf{ch})) \in \mathsf{rf}$, $(\mathsf{snd}_2(\mathsf{ch}), \mathsf{rcv}_2(\mathsf{ch})) \in \mathsf{rf}$, we have $\mathsf{snd}_1(\mathsf{ch}) <_{\text{sat}} \mathsf{snd}_2(\mathsf{ch})$ iff $\mathsf{rcv}_1(\mathsf{ch}) <_{\text{sat}} \mathsf{rcv}_2(\mathsf{ch})$.
2. For each channel ch, if $\mathsf{snd}_1(\mathsf{ch})$ is a matched send event and $\mathsf{snd}_2(\mathsf{ch})$ is an unmatched send event, then $\mathsf{snd}_1(\mathsf{ch}) <_{\text{sat}} \mathsf{snd}_2(\mathsf{ch})$.
3. For each synchronous channel ch, pair $(\mathsf{snd}(\mathsf{ch}), \mathsf{rcv}(\mathsf{ch})) \in \mathsf{rf}$, and event $e \in \mathsf{S}$, we have $e <_{\text{sat}} \mathsf{rcv}(\mathsf{ch})$ iff $e <_{\text{sat}} \mathsf{snd}(\mathsf{ch})$ and $\mathsf{snd}(\mathsf{ch}) <_{\text{sat}} e$ iff $\mathsf{rcv}(\mathsf{ch}) <_{\text{sat}} e$.
4. For each channel ch with $\mathsf{cap}(\mathsf{ch}) = 1$, pair $(\mathsf{snd}_1(\mathsf{ch}), \mathsf{rcv}_1(\mathsf{ch})) \in \mathsf{rf}$, and event $\mathsf{snd}_2(\mathsf{ch}) \neq \mathsf{snd}_1(\mathsf{ch})$, we have $\mathsf{snd}_1(\mathsf{ch}) <_{\text{sat}} \mathsf{snd}_2(\mathsf{ch})$ iff $\mathsf{rcv}_1(\mathsf{ch}) <_{\text{sat}} \mathsf{snd}_2(\mathsf{ch})$.

The saturated order $<_{\text{sat}}$ preserves the consistency of $\mathcal{X}$, as formalized in Lemma 5.1.

LEMMA 5.1. *Let $\Gamma = \langle \mathcal{X}, \text{cap}, rf \rangle$ be a VCh-rf instance and let $<_{\text{sat}}$ be the saturated order of Γ. If Γ is consistent, then $<_{\text{sat}}$ is a partial order (i.e., does not contain cycles), and further, each concretization σ of Γ (if any) must respect $<_{\text{sat}}$.*

Constructing the saturated order. We remark that $<_{\text{sat}}$ can be constructed in at most $O(n^3)$ using a straightforward fix point algorithm. Our actual implementation is optimized and crucially makes use of the recently proposed data structure Collective Sparse Segment Trees (CSSTs) [90] designed for saturation-like fix point computations.

Using the saturated partial order. Based on Lemma 5.1, saturation enhances the decision procedure for VCh-rf in two key ways. First, if the constructed order $<_{\text{sat}}$ is cyclic, then we return NO directly, i.e., without explicitly running any further consistency checking (such as our frontier graph-based or a constraint solving based) procedure. Second, when $<_{\text{sat}}$ is acyclic, then it can be used for a more aggressive on-the-fly pruning of the frontier graph. More specifically, in our frontier graph algorithm, when exploring outgoing edges of a node $u = \langle Y, Q, I \rangle$, we can prune all edges $u \xrightarrow{e} v$ for which e is not enabled in u according to the saturated order, i.e., when $\exists e'$, s.t. $(e', e) \in <_{\text{sat}}$ but e' is not in the set Y of events executed so far. This optimization significantly reduces the number of paths the algorithm must explore. We remark that we also augment the constraints (corresponding to the VCh-rf problem) with additional constraints induced by $<_{\text{sat}}$. However, as we observed in our evaluation, solver-based algorithms cannot benefit from this acceleration, as saturation increases the number of such constraints, slowing down performance.

Discussion. A saturation-style algorithm for consistency checking in *Message Sequence Charts* (MSCs) is proposed in [30]. Our setting generalizes this approach to a more expressive model. In MSCs, channels are assumed to be unbounded, and thus no saturation rules are needed to capture capacity constraints. In contrast, our model supports both bounded and unbounded channels, requiring valid interleavings to respect the capacity limits. This is reflected in our saturation rules – specifically, rules (3) and (4) correspond to capacities 0 and 1. While additional rules could model higher capacities, we omit them to preserve the efficiency of the heuristic saturation procedure.

5.2 Experimental Setup

Given that the problem of consistency checking arises in practical program analyses, the goal of our evaluation is to gauge how our algorithms perform in practice and how they compare against generic solutions such as the use of SMT solvers.

Compared methods and implementations. For our evaluation we focus on the VCh-rf problem and skip evaluation for VCh for two reasons — VCh-rf is more prominent in applications such as model checking and predictive analyses, and further, in our experimental setup, accurate and fast logging of values in executions is challenging. Recall that VCh-rf is an NP-complete problem in general and can alternatively be solved using a constraint solving approach such as the use of SMT solvers, and we also implement such a solution for our evaluation. We compare the performance between two approaches: (1) the frontier graph algorithm FG and (2) SMT-based solvers SMT, along with their respective saturated variants (FG-Sat and SMT-Sat). In our SMT encoding, for each event e, we use an integer variable $0 \leq x_e \leq n-1$, representing the position of e in a potential concretization. For each channel ch, we introduce $2n+2$ auxiliary integer variables to model (1) the cumulative count of send events and (2) the cumulative count of receive events across all prefixes in a potential concretization (encoding details can be found in our technical report [81]). Recall that the saturation-augmented versions SMT-Sat and FG-Sat pre-emptively reject if saturation produces a cycle. If saturation succeeds, then in SMT-Sat, we augment the SMT formula with the constraint $x_e < x_{e'}$, where $(e, e') \in <_{\mathsf{sat}}$ and e' is the earliest event in $\mathsf{th}(e')$ with this property. All algorithms are implemented in Java 23 and SMT/SMT-Sat use the Java bindings of Z3-4.12.2 [29].

Benchmark programs. Our evaluation subjects comprise two distinct groups. The first group is primarily derived from GoBench [95], a widely used Golang concurrency bug benchmark suite. GoBench includes 82 real-world bugs from 9 popular open-source projects (GoReal) and 103 bug kernels (GoKer) [50, 76]. From GoReal, we selected 6 projects for evaluation; the remaining 3 projects were excluded either because we could not log execution traces from them or the generated logs were too short. Similarly, we omitted GoKer benchmarks because they produce executions with too few channel operations to be meaningful for our analysis. The second group consists of additional prominent Golang open-source projects, namely `rpcx`, `raft`, `go-dsp`, `bigcache`, `telegraf`, `ccache`, and `v2ray`, selected to further validate our approach.

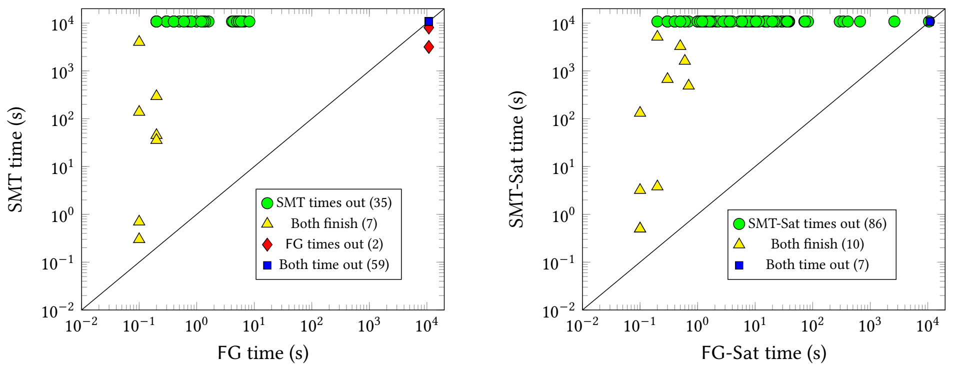
Generation of positive VCh-rf instances. For each benchmark, we first select 1–3 test cases and generate a totally ordered log of channel events in their runtime execution, by modifying ThreadSanitizer [78]. We performed a linear time sanity check that each recorded execution satisfies channel capacity constraints. Each execution log can now be translated to a VCh-rf instance by discarding the total order between events and only retaining program order and the inferred reads-from relations. As a result, the instances thus obtained are positive instances, since the original execution serves as a valid concretization of them. To evaluate algorithmic scalability, we additionally generate new VCh-rf instances by keeping varying length prefixes of existing instances corresponding to long executions (containing thousands to millions of channel accesses). Detailed statistics of these instances can be found in our companion technical report [81].

Generation of mutated instances. Recall from the previous paragraph that VCh-rf instances obtained from real executions are bound to be positive. To obtain negative instances, we mutate the previously obtained positive instances by performing targeted modifications to their reads-from relation as follows. We randomly select a reads-from pair $(\mathsf{snd}_1(\mathsf{ch}), \mathsf{rcv}_1(\mathsf{ch}))$ and another send event $\mathsf{snd}_2(\mathsf{ch})$ on the same channel. If $\mathsf{snd}_2(\mathsf{ch})$ has a matching receive event $\mathsf{rcv}_2(\mathsf{ch})$, then we swap the two reads-from mappings, i.e. enforce $(\mathsf{snd}_1(\mathsf{ch}), \mathsf{rcv}_2(\mathsf{ch})), (\mathsf{snd}_2(\mathsf{ch}), \mathsf{rcv}_1(\mathsf{ch})) \in \mathsf{rf}$. Otherwise, when $\mathsf{snd}_2(\mathsf{ch})$ is not received, we enforce $(\mathsf{snd}_2(\mathsf{ch}), \mathsf{rcv}_1(\mathsf{ch})) \in \mathsf{rf}$ and delete the reads-from pair $(\mathsf{snd}_1(\mathsf{ch}), \mathsf{rcv}_1(\mathsf{ch}))$. For each positive VCh-rf instance with n events, we mutate it $\max(5, 0.05n)$ times. While these mutations do not theoretically guarantee inconsistency, our experimental results show that 88.3% (91/103) of mutated instances become inconsistent, 8.7% (9/103) remain consistent, while the consistency status of the remaining 2.9% (3/103) instances could not be determined due to algorithm timeouts.

Machine configuration and metrics reported. The experiments are conducted on a 2.0GHz 64-bit Linux machine. We set the heap size of JVM to be 100GB and timeout to be 3 hours. For each VCh-rf instance, we report key parameters, such as number of events, threads, channels and maximal channel capacity, as well as the running time of each algorithm. Times reported denote average running time over 3 repeated runs.

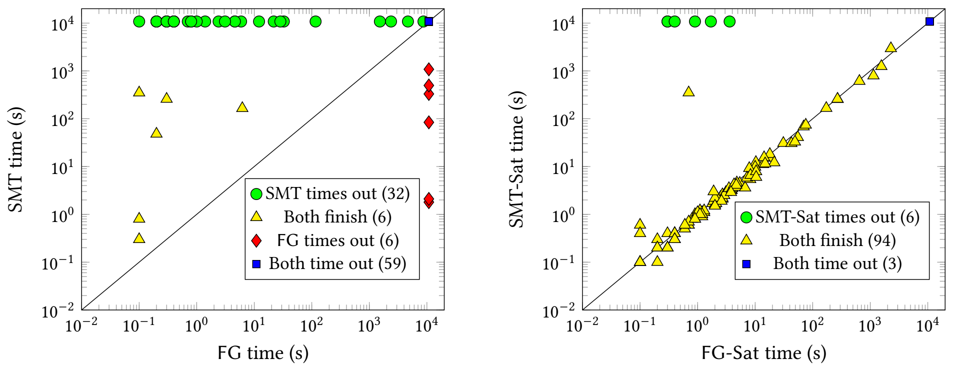
5.3 Evaluation Results for Consistent Instances

Comparison between FG and SMT (Figure 11a). SMT times out on 35 instances due to excessive memory consumption; FG solves all of these successfully. FG fails on only 2 instances that SMT completes. In addition, when both algorithms succeed, FG outperforms SMT in running time by a factor of 5–50,000× (full statistics can be found in our companion technical report [81]). These results demonstrate that FG scales significantly better than SMT on most benchmarks.



(a) FG speedup on consistent instances (b) FG-Sat speedup on consistent instances

Fig. 11. Running time of SMT, SMT-Sat, FG, FG-Sat on consistent instances. The legend indicates the number of instances in each class. The details of running time can be found in our companion technical report [81].



(a) FG speedup on mutated instances (b) FG-Sat speedup on mutated instances

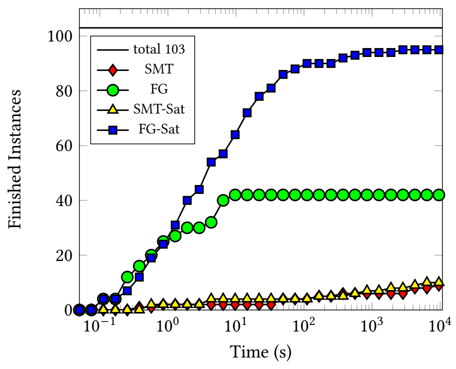
Fig. 12. Running time of SMT, SMT-Sat, FG, FG-Sat on mutated instances. The legend indicates the number of instances in each class. The details of running time can be found in our companion technical report [81].

Comparison between FG-Sat **and** SMT-Sat **(Figure 11b).** SMT-Sat times out on 90.3% (93/103) instances due to increased formula size, which leads to higher memory consumption compared to standard SMT. In contrast, FG-Sat successfully completes 93.2% (96/103) of instances and can often scale to instances with 50k events. These results demonstrate that FG-Sat achieves significantly better scalability than SMT-Sat on consistent benchmarks.

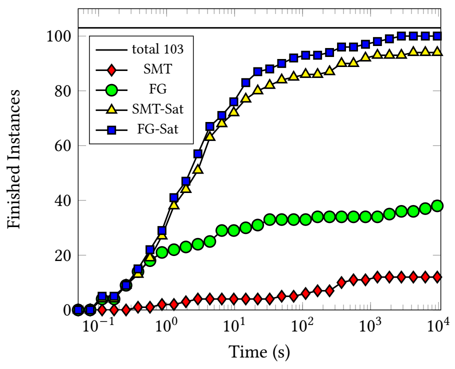
Impact of saturation. Saturation substantially enhances the performance of our frontier graph algorithm — FG-Sat solves 54 more instances than FG. Saturation induces a slight slowdown on some instances (due to the overhead of the fixpoint computation), but this is largely limited to smaller instances where FG already finishes very quickly. The impact of saturation on SMT solvers is limited. SMT-Sat successfully solves only 1 additional instance compared to SMT, and demonstrates significant speed improvements on just 3 benchmarks. We hypothesize that this marginal gain occurs because saturation increases the SMT formula size.

5.4 Evaluation Results for Mutated Instances

Comparison between FG **and** SMT **(Figure 12a).** FG successfully solves 26 more instances than SMT. For instances where both algorithms complete, FG achieves a speedup ranging from 3× to 3000× (full statistics can be found in our companion technical report [81]). Compared to its



(a) Survival plot of consistent instances



(b) Survival plot of mutated instances

Fig. 13. Cumulative number of finished instances over time for each algorithm.

performance on consistent instances, FG solves 4 fewer cases. This happens for inconsistent VCh-rf instances where FG performs a complete traversal of the frontier graph, resulting in increased computational time.

Comparison between FG-Sat **and** SMT-Sat **(Figure 12b).** Both algorithms demonstrate strong performance, successfully completing most instances, with SMT-Sat solving 91.3% (94/103) instances and FG-Sat solving 97.1% (100/103). The superior performance can be attributed to saturation's ability to efficiently reject nearly all inconsistent instances before initiating the core consistency checking algorithm. Notably, among the 6 instances where SMT-Sat times out but FG-Sat succeeds, all are consistent instances (recall that mutated instances are not guaranteed to be inconsistent). In these cases, saturation not only fails to benefit SMT solvers but actually degrades their performance due to the increased formula size.

Survival analysis. In Figure 13, we present the cumulative number of finished instances over time for each algorithm. All algorithms reach saturation after approximately 100 seconds, illustrating the NP-hard nature of the VCh-rf problem – additional time does not lead to solving more instances. For consistent instances, FG-Sat initially completes fewer instances than FG due to the overhead introduced by the saturation process. However, as time progresses, FG-Sat overtakes FG by efficiently pruning infeasible execution paths. For mutated instances, FG-Sat behaves almost identically to SMT-Sat, benefiting from the saturation phase, which filters out inconsistent instances before the main solving procedure begins. Overall, the survival analysis demonstrates that FG-Sat consistently solves the largest number of instances within the same time budget.

In summary, the frontier graph algorithm demonstrates better performance over SMT solving based approach, both with or without saturation. Frontier graph algorithm successfully completes more instances across all benchmarks. When both approaches terminate, the frontier graph algorithm achieves significant speedups ranging from 3× to 50,000×. Further, we observe that saturation can significantly enhance the effectiveness of consistency checking in practice.

6 Other Related Work

Verifying linearizability. Verifying channel consistency resembles verifying linearizability (VL) [4, 16, 17, 32, 40, 42, 46, 59], which checks whether a concurrent history over a (queue) object is equivalent to some sequential one. Unlike VCh, VL enjoys locality, enabling a linear-time decomposition into per-object histories. Moreover, VL typically operates on interval partial orders, making it simpler than VCh. Finally, VL over queues reduces to VCh in linear time.

Message sequence charts. A closely related problem is that of Message Sequence Charts (MSCs) [8, 30, 39, 63], where threads communicate via peer-to-peer channels. Among these, the work of Di Giusto et al. [30] is most relevant. However, our problem strictly generalizes MSCs: (i) send and receive events in VCh need not be paired a priori; (ii) both VCh and VCh-rf allow channels shared by more than two threads; (iii) the same thread pair may communicate over multiple channels; and (iv) channels may have bounded capacities. Our communication model, inspired by languages like Go, assumes channel-based FIFO semantics where all threads can access any channel, with operations on each channel totally ordered. Consequently, their results do not directly extend to our setting. Indeed, in [30], it is shown that the consistency checking problem for these communication models can be solved in polynomial time, because the consistency predicate is MSO definable. In contrast, the consistency checking problem for our setting is NP-hard.

Model checking. The SPIN model checker [47] provides support for programs with message passing, and the GOMELA [31] bounded model checker extends it to support Go programs and uses SPIN as its backend. Given an LTL formula ψ and a program P, SPIN constructs an automaton for $\neg\psi$ and P, and then searches for traces accepted by their product. If such a trace exists, the property ψ does not hold for P. That is, SPIN does not explicitly address the consistency-checking problem. Moreover, MUST [33] is another model checker that operates under the message-passing semantics of MSCs. While MUST implicitly performs consistency checks as part of its model checking algorithm, [33] does not investigate the complexity-theoretic aspects of these checks.

Register consistency checking. The consistency checking problem for registers has been extensively studied in prior work [20, 26, 41, 42]. As demonstrated in this paper, channel consistency checking is strictly harder than register consistency checking due to a key difference in their semantics: registers can only retain the most recent write event, whereas channels can remember up to capacity send events. Related algorithms have also been developed for consistency checking under weak memory models, including TSO [27, 48, 64] and C11 [23, 87].

Predictive analysis. Predictive analysis is a dynamic analysis technique that takes a program execution as input and reorders it to expose potential concurrency bugs. Recent work has developed predictive algorithms for detecting data races [9, 35, 36, 57, 65, 67, 72, 79], deadlocks [51, 89], atomicity violations [37, 68] and more general properties [10, 11]. These algorithms typically compute a candidate set of events and attempt to serialize them into a witness execution, often using a consistency checking oracle. Thus, predictive analysis can be viewed as a downstream application of consistency checking. However, existing prediction algorithms almost exclusively target shared-memory concurrency, neglecting executions involving message-passing via channels. Our work bridges this gap by establishing theoretical foundations for channel-based predictive analysis.

7 Conclusion

Consistency testing is a fundamental task for analyses of concurrent programs such as model checking and predictive testing. We conducted a thorough complexity-theoretic investigation for this problem for the message-passing programming paradigm with FIFO channels. We have developed novel algorithms and established hardness results for a range of inputs parameters. We further implemented and empirically evaluated the performance of our algorithms. Together, our upper and lower bounds reveal an intricate complexity landscape and our evaluation of our algorithms demonstrate their effectiveness in practice. Future work includes applying these algorithms to partial order reduction based model checking and predictive testing for message-passing languages such as Go.

Data-Availability Statement

The artifact containing the VCh-rf instances and source code is available at [80].

Acknowledgments

Zheng Shi and Umang Mathur are partially supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore. Lasse Møldrup and Andreas Pavlogiannis were partially supported by a research grant (VIL42117) from VILLUM FONDEN, and by a research grant from STIBOFONDEN.

References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 373–384. doi:10.1145/2535838.2535845

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 150:1–150:29. doi:10.1145/3360576

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking under the Release-Acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (2018), 29 pages. doi:10.1145/3276505

[4] Parosh Aziz Abdulla, Samuel Grahn, Bengt Jonsson, Shankaranarayanan Krishna, and Om Swostik Mishra. 2025. Efficient Linearizability Monitoring. *Proc. ACM Program. Lang.* 9, PLDI, Article 225 (June 2025), 24 pages. doi:10.1145/3729328

[5] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. 2021. Stateless Model Checking Under a Reads-Value-From Equivalence. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 341–366. doi:10.1007/978-3-030-81685-8_16

[6] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(TACAS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 41–44. doi:10.1007/978-3-642-19835-9_5

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (jul 2014), 74 pages. doi:10.1145/2627752

[8] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. 2005. Realizability and Verification of MSC Graphs. *Theoretical Computer Science* 331, 1 (Feb. 2005), 97–114. doi:10.1016/j.tcs.2004.09.034

[9] Zhendong Ang, Azadeh Farzan, and Umang Mathur. 2025. Enhanced Data Race Prediction Through Modular Reasoning. arXiv:2504.10813 [cs.PL] https://arxiv.org/abs/2504.10813

[10] Zhendong Ang and Umang Mathur. 2024. Predictive Monitoring against Pattern Regular Languages. *Proc. ACM Program. Lang.* 8, POPL, Article 73 (Jan. 2024), 35 pages. doi:10.1145/3632915

[11] Zhendong Ang and Umang Mathur. 2024. Predictive Monitoring with Strong Trace Prefixes. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 182–204. doi:10.1007/978-3-031-65630-9_9

[12] Bengt Aspvall, Michael F Plass, and Robert Endre Tarjan. 1979. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information processing letters* 8, 3 (1979), 121–123. doi:10.1016/0020-0190(79)90002-4

[13] Ranadeep Biswas, Michael Emmi, and Constantin Enea. 2019. On the Complexity of Checking Consistency for Replicated Data Types. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 324–343. doi:10.1007/978-3-030-25543-5_19

[14] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28. doi:10.1145/3360591

[15] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 132:1–132:27. doi:10.1145/3485546

[16] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2018. On reducing linearizability to state reachability. *Information and Computation* 261 (2018), 383–400. doi:10.1016/j.ic.2018.02.014

[17] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 542–563. doi:10.1007/978-3-319-63390-9_28

[18] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. 2023. Dynamic partial order reduction for checking correctness against transaction isolation levels. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 565–590. doi:10.1145/3591243

[19] Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The reads-from equivalence for the TSO and PSO memory models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 164 (oct 2021), 30 pages. doi:10.1145/3485541

[20] Jason F Cantin, Mikko H Lipasti, and James E Smith. 2005. The complexity of verifying memory coherence and consistency. *IEEE Transactions on Parallel and Distributed Systems* 16, 7 (2005), 663–671. doi:10.1109/TPDS.2005.86

[21] Milind Chabbi and Murali Krishna Ramanathan. 2022. A study of real-world data races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 474–489. doi:10.1145/3519939.3523720

[22] Milind Chabbi and Murali Krishna Ramanathan. 2022. A study of real-world data races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 474–489. doi:10.1145/3519939.3523720

[23] Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2024. How Hard Is Weak-Memory Testing? *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1978–2009. doi:10.1145/3632908

[24] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-Centric Dynamic Partial Order Reduction. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–30. doi:10.1145/3158119

[25] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. 2019. Value-centric dynamic partial order reduction. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 124:1–124:29. doi:10.1145/3360550

[26] Yunji Chen, Lei Li, Tianshi Chen, Ling Li, Lei Wang, Xiaoxue Feng, and Weiwu Hu. 2012. Program regularization in memory consistency verification. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (2012), 2163–2174. doi:10.1109/TPDS.2012.44

[27] Peter Chini and Prakash Saivasan. 2020. A Framework for Consistency Algorithms. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 182)*, Nitin Saxena and Sunil Simon (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 42:1–42:17. doi:10.4230/LIPIcs.FSTTCS.2020.42

[28] Ugo Dal Lago and Alexis Ghyselen. 2024. On Model-Checking Higher-Order Effectful Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2610–2638. doi:10.1145/3632929

[29] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[30] Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. 2023. A Partial Order View of Message-Passing Communication Models. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 55:1601–55:1627. doi:10.1145/3571248

[31] Nicolas Dilley and Julien Lange. 2022. Automated verification of go programs via bounded model checking. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 1016–1027. doi:10.1109/ASE51524.2021.9678571

[32] Michael Emmi and Constantin Enea. 2019. Violat: Generating Tests of Observational Refinement for Concurrent Objects. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 534–546. doi:10.1007/978-3-030-25543-5_30

[33] Constantin Enea, Dimitra Giannakopoulou, Michalis Kokologiannakis, and Rupak Majumdar. 2024. Model Checking Distributed Protocols in Must. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 338 (Oct. 2024), 28 pages. doi:10.1145/3689778

[34] Erlang developing team Erlang developers. 2024. Erlang documentations. https://www.erlang.org/doc/system/conc_prog.html.

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

[36] 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. doi:10.1145/1542476.1542490

[37] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 293–303. doi:10.1145/1375581.1375618

[38] Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. 2014. Memory Model-Aware Testing - A Unified Complexity Analysis. In *2014 14th International Conference on Application of Concurrency to System Design*. IEEE, 92–101. doi:10.1109/ACSD.2014.27

[39] B. Genest and A. Muscholl. 2005. Message sequence charts: a survey. In *Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*. IEEE, 2–4. doi:10.1109/ACSD.2005.25

[40] Phillip B Gibbons, John L Bruno, and Steven Phillips. 2002. Black-Box Correctness Tests for Basic Parallel Data Structures. *Theory of Computing Systems* 35, 4 (2002), 391–432. doi:10.1007/s00224-002-1046-6

[41] Phillip B. Gibbons and Ephraim Korach. 1994. On testing cache-coherent shared memories. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures* (Cape May, New Jersey, USA) *(SPAA '94)*. Association for Computing Machinery, New York, NY, USA, 177–188. doi:10.1145/181014.181328

[42] Phillip B Gibbons and Ephraim Korach. 1997. Testing shared memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. doi:10.1137/S0097539794279614

[43] Go developing team Go developers. 2024. channel features in Go. https://github.com/golang/go/blob/master/src/runtime/chan.go.

[44] Go developing team Go developers. 2024. channel features in Go. https://go.dev/tour/concurrency/2.

[45] Go developing team Go developers. 2024. Effective Go. https://golang.org/doc/effective_go.html.

[46] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492. doi:10.1145/78969.78972

[47] G.J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295. doi:10.1109/32.588521

[48] Weiwu Hu, Yunji Chen, Tianshi Chen, Cheng Qian, and Lei Li. 2011. Linear time memory consistency verification. *IEEE Trans. Comput.* 61, 4 (2011), 502–516. doi:10.1109/TC.2011.41

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

[50] Zongze Jiang, Ming Wen, Yixin Yang, Chao Peng, Ping Yang, and Hai Jin. 2023. Effective Concurrency Testing for Go via Directional Primitive-Constrained Interleaving Exploration. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1364–1376. doi:10.1109/ASE56229.2023.00086

[51] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. doi:10.1145/3276516

[52] 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. doi:10.1145/3062341.3062374

[53] 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. doi:10.1016/0020-0190(79)90002-4

[54] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. 2022. Truly Stateless, Optimal Dynamic Partial Order Reduction. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 49:1–49:28. doi:10.1145/3498711

[55] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *CAV 2021*. Springer-Verlag, Berlin, Heidelberg, 427–440. doi:10.1007/978-3-030-81685-8_20

[56] Kotlin developing team Kotlin developers. 2024. Kotlin documentations. https://kotlinlang.org/docs/channels.html.

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

[58] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323. doi:10.1007/978-3-662-47666-6_25

[59] Zheng Han Lee and Umang Mathur. 2025. Efficient Decrease-and-Conquer Linearizability Monitoring. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 345 (Oct. 2025), 28 pages. doi:10.1145/3763123

[60] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When threads meet events: efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International*

Conference on Programming Language Design and Implementation (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 725–739. doi:10.1145/3453483.3454073

[61] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. 2022. Who goes first? detecting go concurrency bugs via message reordering. 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, 888–902. doi:10.1145/3503222.3507753

[62] Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 630–646. doi:10.1145/3445814.3446711

[63] P. Madhusudan. 2001. Reasoning about Sequential and Branching Behaviours of Message Sequence Graphs. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming, (ICALP '01)*. Springer-Verlag, Berlin, Heidelberg, 809–820. doi:10.1007/3-540-48224-5_66

[64] C. Manovit and S. Hangal. 2006. Completely verifying memory consistency of test program executions. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* IEEE, 166–175. doi:10.1109/HPCA.2006.1598123

[65] 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. doi:10.1145/3276515

[66] 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*. ACM, Saarbrücken Germany, 713–727. doi:10.1145/3373718.3394783

[67] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal prediction of synchronization-preserving races. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434317

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

[69] Lasse Møldrup and Andreas Pavlogiannis. 2025. AWDIT: An Optimal Weak Database Isolation Tester. *Artifact for "AWDIT: An Optimal Weak Database Isolation Tester"* 9, PLDI (June 2025), 209:1540–209:1564. doi:10.1145/3742465

[70] Stefan K Muller. 2024. Language-Agnostic Static Deadlock Detection for Futures. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 68–79. doi:10.1145/3627535.3638487

[71] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/osdi-08/finding-and-reproducing-heisenbugs-concurrent-programs

[72] Andreas Pavlogiannis. 2020. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–29. doi:10.1145/3371085

[73] Hernán Ponce-de León, Thomas Haas, and Roland Meyer. 2022. Dartagnan: SMT-based Violation Witness Validation (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 418–423. doi:10.1007/978-3-030-99527-0_24

[74] Rust developing team Rust developers. 2024. Rust documentations. https://doc.rust-lang.org/std/index.html.

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

[76] Georgian-Vlad Saioc, I-Ting Angelina Lee, Anders Møller, and Milind Chabbi. 2025. *Dynamic Partial Deadlock Detection and Recovery via Garbage Collection.* Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/3676641.3715990

[77] Scala developing team Scala developers. 2024. Scala documentations. https://www.scala-lang.org/api/3.4.2/docs/index.html.

[78] 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. doi:10.1145/1791194.1791203

[79] Zheng Shi, Umang Mathur, and Andreas Pavlogiannis. 2024. Optimistic Prediction of Synchronization-Reversal Data Races. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 134, 13 pages. doi:10.1145/3597503.3639099

[80] Zheng Shi, Lasse Møldrup, Umang Mathur, and Andreas Pavlogiannis. 2025. *Artifact.* doi:10.5281/zenodo.18091491

[81] Zheng Shi, Lasse Møldrup, Umang Mathur, and Andreas Pavlogiannis. 2025. Technical report. *arXiv preprint arXiv:2505.05162* (2025). https://arxiv.org/abs/2505.05162

[82] Kyle Storey, Eric Mercer, and Pavel Parizek. 2021. A Sound Dynamic Partial Order Reduction Engine for Java Pathfinder. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2021), 15–15. doi:10.1145/3364452.3364457

[83] Martin Sulzmann and Kai Stadtmüller. 2017. Trace-Based Run-Time Analysis of Message-Passing Go Programs. In *Hardware and Software: Verification and Testing*, Ofer Strichman and Rachel Tzoref-Brill (Eds.). Springer International Publishing, Cham, 83–98. doi:10.1007/978-3-319-70389-3_6

[84] Martin Sulzmann and Kai Stadtmüller. 2018. Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming* (Frankfurt am Main, Germany) *(PPDP '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 13 pages. doi:10.1145/3236950.3236959

[85] Martin Sulzmann and Kai Stadtmüller. 2018. Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming* (Frankfurt am Main, Germany) *(PPDP '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 13 pages. doi:10.1145/3236950.3236959

[86] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878. doi:10.1145/3297858.3304069

[87] Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023. Optimal Reads-From Consistency Checking for C11-Style Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 761–785. doi:10.1145/3591251

[88] Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Cirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis *(ASPLOS '24, Vol. 3)*. Association for Computing Machinery, New York, NY, USA, 223–238. doi:10.1145/3620666.3651358

[89] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound dynamic deadlock prediction in linear time. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1733–1758. doi:10.1145/3591291

[90] Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Cirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 223–238. doi:10.1145/3620666.3651358

[91] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 474–486. doi:10.1145/3510003.3510178

[92] Ryan Williams. 2005. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science* 348, 2 (2005), 357–365. doi:10.1016/j.tcs.2005.09.023 Automata, Languages and Programming: Algorithms and Complexity (ICALP-A 2004).

[93] Dylan Wolff, Zheng Shi, Gregory J. Duck, Umang Mathur, and Abhik Roychoudhury. 2024. Greybox Fuzzing for Concurrency Testing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (, La Jolla, CA, USA,) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 482–498. doi:10.1145/3620665.3640389

[94] Wenhao Wu, Jan Hückelheim, Paul D. Hovland, Ziqing Luo, and Stephen F. Siegel. 2023. Model Checking Race-Freedom When "Sequential Consistency for Data-Race-Free Programs" is Guaranteed. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 265–287. doi:10.1007/978-3-031-37703-7_13

[95] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 187–199. doi:10.1109/CGO51591.2021.9370317

[96] Rachid Zennou, Mohamed Faouzi Atig, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2020. Boosting Sequential Consistency Checking Using Saturation. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 360–376. doi:10.1007/978-3-030-59152-6_20

Received 2025-06-30; accepted 2025-11-06