Model Checking Distributed Protocols in Must

CONSTANTIN ENEA, AWS and Ecole Polytechnique, France
DIMITRA GIANNAKOPOULOU, AWS, USA
MICHALIS KOKOLOGIANNAKIS*, ETH Zurich, Switzerland
RUPAK MAJUMDAR, AWS and MPI-SWS, Germany

We describe the design and implementation of Must, a framework for modeling and automatically verifying distributed systems. Must provides a concurrency API that supports multiple communication models, on top of a mainstream programming language, such as Rust. Given a program using this API, Must verifies it by means of a novel, optimal dynamic partial order reduction algorithm that maintains completeness and optimality for all communication models supported by the API.

We use Must to design and verify models of distributed systems in an industrial context. We demonstrate the usability of Must's API by modeling high-level system idioms (e.g., timeouts, leader election, versioning) as abstractions over the core API, and demonstrate Must's scalability by verifying systems employed in production (e.g., replicated logs, distributed transaction management protocols), the verification of which lies beyond the capacity of previous model checkers.

CCS Concepts: • **Theory of computation → Distributed computing models**; **Verification by model checking**.

Additional Key Words and Phrases: Distributed Systems, Model Checking

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

1 Introduction

Despite significant research advances in the past years, formal methods are yet to be integrated in the development process of distributed systems. A major hindrance in their adoption is that existing techniques fail to adequately balance the conflicting desiderata of researchers and programmers: researchers typically design powerful (yet expensive) reasoning techniques for programs written in domain-specific languages, while programmers prefer mainstream programming languages, and do not tolerate non-scalable tools that interfere with their development. As such, while domain-specific languages for distributed systems [Holzmann 1997; Lamport 2002; Padon et al. 2016] offer abstract modeling and systematic exploration for finite-state abstractions, they do not integrate well with the rest of the software engineering process. On the other hand, tools for systematic concurrency testing built on mainstream languages [Deligiannis et al. 2021; Meng et al. 2023; Musuvathi et al. 2008; Ozkan et al. 2018] do not provide adequate interfaces for abstract modeling (e.g., no support

*Work done while at AWS

Authors' Contact Information: Constantin Enea, AWS and Ecole Polytechnique, France, cenea@amazon.com; Dimitra Giannakopoulou, AWS, USA, dimgia@amazon.com; Michalis Kokologiannakis, ETH Zurich, Switzerland, michalis.kokologiannakis@inf.ethz.ch; Rupak Majumdar, AWS and MPI-SWS, Germany, rumajumd@amazon.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/10-ART338
https://doi.org/10.1145/3689778

for data non-determinism, multiple communication primitives, etc), and/or do not provide any verification guarantees.

In this paper, we present Must, a framework for modeling and verifying distributed systems that reconciles the above desiderata, thereby lowering the adoption barrier for systems developers. Must advocates the combination of a **mainstream programming language** for describing local computation, and of a lean **concurrency API** for describing communication among processes. Must's concurrency API comes with a semantics that is tightly-coupled with a novel **dynamic partial order reduction (DPOR)** algorithm [Flanagan and Godefroid 2005; Kokologiannakis et al. 2022], which exhaustively enumerates all possible program executions modulo an equivalence relation. This design renders Must expressive enough to model complex distributed systems in an industrial context, yet still amenable to **scalable** model checking techniques.

Programming Language: The basic observation behind Must is that large distributed-system implementations mostly consist of complex sequential code, and only use a handful of primitives for inter-process communication. As such, Must can be instantiated for any language, so long as that language can enforce that all concurrent communication is performed through Must's API. In this work, we use Rust as the underlying programming language, and enforce that all inter-process communication uses Must's API via the Rust type system (see §5).

Must API: In the Must API, distributed system processes are modeled as threads that communicate through implicitly-created channels using *send* and *receive* operations. In addition to these basic operations, Must's API comes with built-in semantics for several features, including: (a) *non-blocking receives* (i.e., receives that return immediately if there is no pending message), (b) *selective receives* (i.e., receives that can process messages out of order based on a user predicate), (c) *non-deterministic choice*, and (d) messages being sent/received under *different semantics* (e.g., certain messages being sent with TCP semantics and others with UDP semantics). To the best of our knowledge, Must's API is the first API that provides a semantics supporting all the above features.

Must DPOR: The Must API is supported by a novel DPOR algorithm that is directly applicable to its multiple message-passing primitives, and natively handles all of its features. The resulting algorithm is novel in a number of different aspects.

(1) It is *parametric* w.r.t. the choice of the underlying model, and has *polynomial memory consumption* in the size of the input program. Even though such parametric algorithms have been proposed for the shared-memory setting (e.g., [Kokologiannakis et al. 2022]), they do not naturally carry over to message-passing concurrency[1]. One issue is that they assume that a given execution can always be extended by executing another instruction, which is not always the case in message-passing concurrency: blocking receives can only be executed if there is a corresponding message to read. Another issue is that shared-memory DPORs use a "revisiting condition" [Bouajjani et al. 2023; Kokologiannakis et al. 2022] to generate new executions from a given one, but that revisiting condition does not work for message-passing concurrency.
(2) It is *optimal*, i.e., explores each program behavior (equivalence class of executions) exactly once and does not initiate redundant explorations. Must is the first optimal DPOR algorithm to operate under the "reads-from" equivalence relation [Abdulla et al. 2018; Chalupa et al. 2017] in message-passing concurrency (i.e., it never explores two executions where all receives read precisely the same messages), while having polynomial memory consumption.
(3) It handles messages being sent/received with different semantics, as dictated by the Must API.
(4) It can deal with *global safety properties*. Traditionally, DPOR algorithms could only verify properties expressed via `assert` statements, which can only reference the state of a given

[1]Although one could in principle "compile" all message-passing features to shared-memory, as we show in §6, such an approach unnecessarily blows up the state space.

```
loop {
  match (recv^p2p()) {
    Read(client) =>
      { if is_tail { send^p2p(client, log); } }
    Write(val, client) => {
      if is_head {
        append(log, val);
        if is_tail {
          send^mbox(storage, val);
          send^p2p(client, ok);
          continue; }
        send^p2p(next, Update(val, client));
    }}
    Update(val, client) => { append(log, val);
      if is_tail { send^p2p(client, ok); }
      else {  send^p2p(next, Update(client, val));
    }}
    ...
}}
```

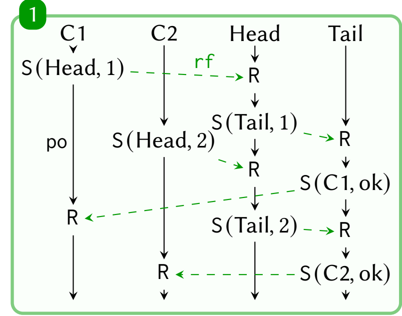


Fig. 1. Toy replication log in Rust-like syntax (left); an execution graph for two nodes and two writes (right). For readability, we omit from the execution graph the sends to the storage and all communication models.

process. As such, reasoning about the global state of the system (e.g., whether all processes agree on a value) was difficult. Must provides a way of reasoning about global properties via dedicated processes called *monitors*, which track the global state by "snooping" messages sent by the rest of the system. This is achieved by having monitors subject to a different communication model than the rest of the system, in turn made possible by Must's capability to handle combinations of communication primitives.

Scalability: To demonstrate the effectiveness of our framework, we used Must to model and verify complex distributed systems employed in production (namely, two replicated log protocols and a distributed commit protocol). In doing so, we found all features of both the Must API and the Must DPOR extremely useful. Must's API made it possible to easily express many patterns commonly found in distributed systems: we used TCP semantics to model a state machine replication protocol, together with UDP semantics for heartbeat messages related to fault detection; we emulated timeouts using non-deterministic choice; we modeled versioning using selective receives, etc. The polynomial memory requirements of Must's DPOR allowed us to run multiple Must instances on servers for multiple days, without having to worry about crashes and out-of-memory errors. In all our experiments, the execution time and memory overhead for Must was orders of magnitude lower than existing tools for systematic state-space exploration of distributed systems.

Our contributions can be summarized as follows:

- §2 We propose a new design for modeling distributed systems on top of mainstream languages.
- §3 We describe the Must API in detail, and formalize its underpinning partial-order semantics.
- §4 We describe the Must DPOR algorithm, and prove it sound, complete and optimal.
- §6 We implement Must as a tool for Rust programs, and use it to model and verify complex distributed protocols used in cloud applications in an industrial context.

2 A Tour of Must

We start with a quick overview of the Must API (§ 2.1) and DPOR (§ 2.2), and then demonstrate the tight coupling between the two by modeling common patterns in distributed systems (§ 2.3).

2.1 Must API: The Basics

2.1.1 Programming Model. Given a full-fledged programming language, Must extends it with a concurrency API that undertakes all message-passing communication. The Must API models each node in a distributed system as a thread which, in addition to all sequential programming language constructs, also has access to the following primitives:

$$\mathtt{nondet}(S) \quad | \quad \mathtt{send}^{\mathsf{M}}(t, v) \quad | \quad \mathtt{recv}^{\mathsf{M},\beta}(\lambda x : e)$$

$\mathtt{nondet}(S)$ is *nondeterministic choice* operator that chooses a value v from a finite set $S \triangleq \{v_1, \ldots, v_n\}$; $\mathtt{send}^{\mathsf{M}}(t, v)$ sends a value v to the thread with identifier t; and the (selective) receive $\mathtt{recv}^{\mathsf{M},\beta}(\lambda x : e)$ waits for a message matching the predicate $\lambda x : e$, and returns the received message.

The receive attribute $\beta \in \{\mathsf{b}, \mathsf{nb}\}$ denotes whether a receive is blocking or non-blocking, while the communication model M determines the semantics of message passing. Example models include: (a) *FIFO peer-to-peer* (p2p), modeling TCP streams where a receiver receives the messages of a given sender in FIFO order, (b) *mailbox* (mbox), modeling local channels where messages are received in the global order in which they were sent, and (c) *full asynchrony* (asyn), modeling UDP streams where any previously sent message can be received. In fact, Must can accommodate multiple communication models used in a single execution graph; for more details, see §2.3.5 and §3.

In the next subsections, we motivate these choices for the Must API using the example below.

Example 2.1 (Must API: Motivation) Consider the chain replication protocol [Renesse and Schneider 2004] in Fig. 1 (left). The protocol comprises a chain of nodes, delimited by Head and Tail, each one replicating the log. Client read requests are handled by Tail, which processes them locally using its own replica. Client write requests, on the other hand, are sent to the Head, which appends the written value to its log using `append`, and then forwards the update to the next node in the chain using an `Update` message. When a write request reaches Tail, the latter sends the value to a durable storage (see §2.3.5) and informs the client of the successful request.

As can already be seen, the Must API is expressive enough to model inter-process communication (`send()` and `recv()` are provided by the Must API), while all sequential computation (e.g., control flow) can be readily modeled using an existing language like Rust.

2.1.2 Partial Order Semantics: Execution Graphs. To formalize the semantics of Must's concurrency API, we use declarative semantics and *execution graphs* [Alglave et al. 2014].

Informally, an execution graph captures the flow of communication in an execution. It contains: (i) a set of events (nodes), modeling the operations captured by the Must API for a given program (i.e., send, receive, and nondeterministic choice), and (ii) certain relations on these events (edges), representing various constraints on the communication. Two relations included in all communication models are the *program order* (po), which orders the events within a given thread, and the *reads-from* relation (`rf`), which specifies the send from which each receive gets its value.

The semantics of communication are determined by a communication model M, which constrains `rf` so that the resulting graph is consistent under M. For instance, under p2p, `rf` is restricted so that messages from a given thread are received in FIFO order, while under mbox, messages are received based on a global ordering on their corresponding sends. More generally, we assume that M provides a consistency predicate, and say that a given execution graph G is consistent, written $\mathsf{consistent}_{\mathsf{M}}(G)$, if M's consistency predicate holds in G. The semantics of a program P is then given by the set of consistent execution graphs corresponding to P.

Example 2.2 (Execution Graphs) Assuming a system with two nodes and two concurrent write requests, the semantics of the replicated log in Fig. 1 comprises two execution graphs. Each of these

graphs represents a distinct behavior: it captures a set of equivalent interleavings, in all of which the receives of the program read the messages indicated by the graph's rf relation. In other words, an execution graph captures an *equivalence class*.

The graph where the log processes the request of C1 before that of C2 can be seen in Fig. 1 (right). This graph captures 106 interleavings, obtainable by linearizing $(\mathsf{po} \cup \mathtt{rf})^{+}$. The graph where the requests are received in the other order is symmetric, and captures 106 further interleavings.

2.2 Must DPOR: An Overview

Given a program that uses Must's API, Must verifies it by means of DPOR. In this section, we present an overview of the DPOR algorithm accompanying the Must API.

The Must DPOR verifies a given program automatically, by exploring all of its consistent execution graphs in a *complete* and *optimal* way, while using polynomial space overhead. Completeness means that Must explores all consistent execution graphs, while optimality means that each execution graph is explored exactly once, and no fruitless explorations are initiated. Even though we only show a few communication models in this paper, Must is sound for any communication model that satisfies certain conditions (see §3).

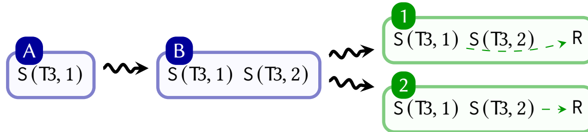
At a high level, the Must DPOR builds the execution graphs of a given program incrementally, in a depth-first manner, while ensuring that the graph at each exploration step remains consistent. In doing so, when it encounters a potential choice point, it recursively considers all exploration options. For instance, when a receive can read one of several unread messages, Must will separately consider each of these scenarios. Conversely, when a newly-found send operation may become the source of a previously-executed receive operation, Must will explore that scenario as well.

We illustrate the basic ideas behind the Must DPOR with the following example.

Example 2.3 (Must DPOR: A First Example) Consider the S+S+R program below, an abstraction of our chain replication example, where two concurrent clients (threads T1 and T2) communicate with the replicated log (thread T3).

$$\text{T1: } \mathsf{send}(\text{T3}, 1) \;\Big\|\; \text{T2: } \mathsf{send}(\text{T3}, 2) \;\Big\|\; \text{T3: } \mathsf{recv}() \qquad (\text{S+S+R})$$

S+S+R has two consistent execution graphs, 1 and 2, since the receive can either read 1 or 2.



An exploration of the Must DPOR closely follows a depth-first search. Assuming that Must runs T1 before T2, it explores graphs A and B, each of which contains the events of the threads explored so far. When Must encounters the receive in T3, it realizes that either of the two sends can be received, and therefore recursively explores both options in graphs 1 and 2. These graphs are complete, and Must concludes the exploration.

Although the example above was straightforward, things may not always be so simple. To see why, let us instead assume that Must performs the exploration in a different order, and schedules T3 before the other two threads. Even though such an exploration should also produce two consistent execution graphs, if Must first runs T3, then the receive will have no message to read from. To make matters worse, Must cannot follow existing DPORs and simply add the receive, block the

Conventions

In execution graphs, we treat `rf` as a relation from sends to receives. In programs, we write $\texttt{recv}^{\mathsf{M}}()$ for the special case when the receive predicate is true, omit the receive attribute for blocking receives, and omit the communication model when all communication is performed under p2p. In explorations, we use $\rightsquigarrow$ to delineate exploration steps, letters to refer to intermediate executions, and numbers to refer to full executions.

corresponding thread, and then change its `rf` when a send is encountered, because, in general, a send to the receiving thread might never appear (in such a case the graph would be inconsistent, as it would contain a blocking receive reading from nowhere).

Must deals with the above issues by making two contributions. First, it only adds a blocking receive if there is some message the receive can read. Note that this is in contrast to graph-based DPORs for shared memory, where events can always be added to the graph. As we show in §3.2.1, such an assumption is unnecessary to obtain an optimal DPOR for message-passing concurrency.

Second, as it is still possible that only a subset of possible messages are available when Must encounters a receive r, when a new message s to r appears, the Must DPOR performs a careful "backward revisit", and examines the case where r reads from s. While necessary for completeness, backward revisits have to be handled with care, since they might violate optimality: two different backward revisits can lead to the same execution graph. As such, Must's algorithm maintains a careful ordering and a novel "tie-breaking" condition to ensure that such duplicate exploration does not arise[2]. We defer the full presentation of Must's DPOR to §4.

2.3 Modeling Patterns: Using the Must API and DPOR

Key to Must's scalability is the tight coupling between its API and its DPOR. We now show how the API handles common distributed system patterns, and describe the changes induced in the DPOR.

2.3.1 Nondeterminism. A first feature of Must is that it supports "higher level" patterns and data-non-determinism. Must achieves this by incorporating a nondeterministic choice operator in its API and DPOR. A usage example of this operator can be seen below.

Example 2.4 (Must API: Modeling Failures) A replicated log like the one in our motivating example (Fig. 1) needs to be *fault tolerant*: a chain with N nodes should tolerate up to $N-1$ node failures. Fault tolerance is typically achieved by relying on a fault-free coordinator who is in charge of monitoring the chain and appropriately initiates updates if any node fails. In an implementation, the coordinator is a Paxos cluster, and failure detection uses heartbeat messages and timeouts. In a model, we can abstract away the cluster to a single non-failing process, and model failure detection by explicitly failing a particular process in a non-deterministic manner.

To explicitly fail processes in Must, we can use its nondeterministic choice operator:

```
fn environment {
  for i in 0..NUM_FAULTS {
    let who = nondet(0..NUM_NODES);
    send^p2p(coord, Fail(who));
}}
```

```
fn coordinator {
  loop { match (recv^p2p()) {
      Fail(nd) => recover(...),
      ...
}}}
```

Given a fault budget `NUM_FAULTS`, an environment process sends `NUM_FAULTS` failure messages to the coordinator, with each message carrying the id of the failing node (picked non-deterministically).

[2] Shared-memory DPORs also employ revisiting conditions, but such conditions cannot be utilized by Must (see §4).

(We write `nondet(0..NUM_NODES)` to denote choice from the interval `[0..NUM_NODES)`.) The failure messages are interleaved with other system messages, thereby exercising all possible failure points.

To support data-nondeterminism in the Must DPOR, we treat nondeterministic choice similarly to how we treat multiple `rf` choices for receives: Must collects all possible outcomes of the nondeterministic choice, and tries each one of them separately, by means of backtracking.

2.3.2 Timeouts. A commonly occurring pattern in distributed systems is a receive operation that may time out. For instance, a failure detector may initiate recovery if it does not receive a special "heartbeat" message within a predetermined amount of time. Such treatment of timeouts is so common in distributed computing that Erlang provides first-class primitives to specify timeouts [Erlang 2023].

While Must's API does not have an explicit notion of time, one could model timeouts by using the nondeterministic choice operator. A naive encoding might look like the following:

$$\texttt{recv_timeout}^{\mathsf{M}}(\lambda x : e) \triangleq \texttt{if (nondet(}\{0,1\}\texttt{)) \{ recv}^{\mathsf{M}}(\lambda x : e) \texttt{ \} else \{ } \bot \texttt{ \}}$$

Above, if the (binary) non-deterministic choice operator returns 1, `recv_timeout()` boils down to a blocking receive, while if it returns 0, `recv_timeout()` returns $\bot$.

Unfortunately, as the example below demonstrates, this encoding leads to state-space explosion.

Example 2.5 (Must API: Naive Timeout Modeling) In the NNR program below, N threads concurrently perform a receive that might timeout:

$$\text{T1: } \texttt{recv_timeout()} \;\|\; \dots \;\|\; \text{TN: } \texttt{recv_timeout()} \qquad (\text{NNR})$$

The problem with the naive encoding above is that the Must DPOR would examine 2^N different executions (corresponding to the all the possibilities for the nondeterministic choices), despite the fact that no message is ever sent (e.g., because the senders failed/terminated).

As such, the Must API supports timeouts by providing a *non-blocking* receive $r := \texttt{recv}^{\mathsf{M,nb}}(\lambda x : e)$ (marked using the nb attribute), that mimics the usual receive instruction, but can return either some value or $\bot$ (nondeterministically).

In turn, the Must DPOR accounts for non-blocking receives by also considering $\bot$ as a possible `rf` for such receives, in addition to all other consistent sends they can read from. (Note that, in contrast to sends, $\bot$ can be read by multiple non-blocking receives). With this treatment, Must is exponentially faster than the naive encoding: it explores a single execution in the case of NNR (where all receives read $\bot$).

2.3.3 Selective Receives. So far, we have considered receives whose predicates are trivially true. Must, however, additionally provides the ability to constrain the values a receive can read via a predicate. As the following example demonstrates, receive predicates may exponentially decrease the state space of a program.

Example 2.6 (Must API: Modeling Versioning) A common issue in fault-tolerant distributed systems like the one in Fig. 1 is a *split brain* situation: the coordinator believes a node has failed and expels it from the chain, but the node itself does not realize this and continues to accept requests. In such a scenario, client requests should not be handled by the expelled node, as the latter might have a stale view of the system.

One way to ensure clients do not get stale values is by maintaining a *version number*. When a client queries the coordinator for the current head or tail, it receives their IP addresses as well as the current version number, which it then includes in its future requests. The nodes on the chain

match the client number with their own, and reject "stale" messages. Any configuration change by the coordinator increases the version number, which is then propagated to the chain.

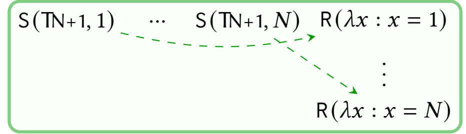
As far as DPOR is concerned, receiving such "stale" messages introduces additional combinatorial explosion even if the messages have no effect on the receiver's state (DPOR has to still explore all possible `rf`s for all the program receives). One way such explosion can be mitigated is by using `assume` statements: in our example above, one could block stale messages using an `assume`(version = client.version) statement, which could wake the model checker block the executing thread if the check fails. The downside of this approach is that DPOR has to still examine all `rf`s for a given receive, even if the executing thread blocks for only some of these `rf`s.

The selective receives provided by the Must API are a superior strategy: nodes declare predicates in their selective receive operations, and the Must DPOR will only consider the `rf` options that satisfy the predicates. A usage example can be seen below.

Example 2.7 (Must API: Selective receives) In NS+RN-SEL, T_{N+1} requires that N messages be received in a particular order.

$$\mathrm{T}_1\colon \mathtt{send}(\mathrm{T}_{N+1}, 1) \;\Big\|\; \ldots \;\Big\|\; \mathrm{T}_N\colon \mathtt{send}(\mathrm{T}_{N+1}, N) \;\Big\|\; \begin{array}{l}\mathrm{T}_{N+1}\colon \mathbf{for}\ (i \in [1 \ldots N])\\ \qquad\quad \mathtt{recv}(\lambda x : x = i)\end{array} \qquad \text{(NS+RN-SEL)}$$

In programs like NS+RN-SEL, we only care about executions where receives read values that satisfy the receive predicates. The single graph that satisfies the receiver requirements can be seen on the right.

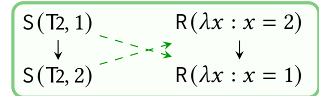


Had we used an `assume` in the last thread to restrict the received messages, Must would have explored N^2 executions. In general, selective receives can lead to exponentially fewer executions, compared to `assume` statements.

As the following example demonstrates, however, selective receives not only lead to exponential reductions compared to `assume`s, but they are also strictly more expressive: they allow node to process messages in a given order, even if that order contradicts the order in which the messages were sent.

Example 2.8 (Must API: Receiving Messages Out-of-Order) Consider the program below:

$$\begin{array}{l}\mathtt{send}(\mathrm{T}_2, 1)\\ \mathtt{send}(\mathrm{T}_2, 2)\end{array} \;\Big\|\; \begin{array}{l}\mathtt{recv}(\lambda x : x = 2)\\ \mathtt{recv}(\lambda x : x = 1)\end{array}$$



This graph is p2p-consistent, despite the messages received in the opposite order they were sent.

To support selective receives in the Must DPOR, one could opt for a "modeling" solution, and implement selective receives with the help of a save queue: a process scans a FIFO buffer, and messages that do not match the predicate are temporarily put into a save queue. When a message that satisfies the predicate is found, all messages in the save queue are re-inserted in the FIFO buffer in the order they arrived.

Of course, such an implementation is not suitable for a model checker, as it introduces additional state. In response, Must supports selective receives not via an algorithmic solution, but rather by incorporating them into the underlying communication model. Then, by proving the Must DPOR correct for any communication model (see §3), we are able to get support for selective receives "for free".

2.3.4 Ordering Messages Lazily. Owing to message losses and timeouts, many distributed protocols have many more messages sent than received. Must's DPOR contains an optimization to take advantage of this observation; the optimization tracks the ordering among sent messages *lazily*.

Let us consider a variation of S+S+R where N threads send a message to thread N+1:

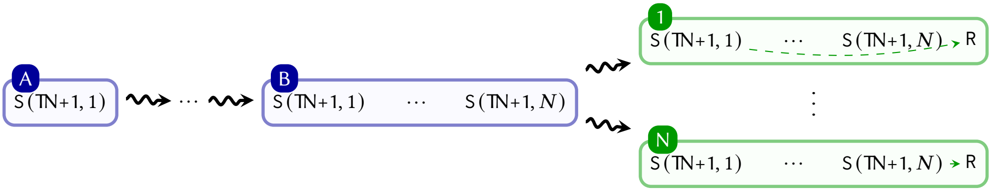
$$\text{T1: send(TN+1, 1)} \;\|\; \dots \;\|\; \text{TN: send(TN+1, } n\text{)} \;\|\; \text{TN+1: recv()} \qquad \text{(NS+R)}$$

An *eager* approach would immediately impose a total ordering on all the messages sent to thread $N + 1$, and ensure that a receive reads the earliest message. (In the shared memory setting, this order is called coherence [Alglave et al. 2014].) In this example, there are $N!$ orderings of the sent messages, and the receive would read the first message in each ordering. Clearly, however, this is wasteful: what matters is which message arrives first, and not the order of the later messages.

As we have already hinted in our examples, the Must DPOR orders messages to a given thread *lazily*. Sends remain unordered when they are added to an execution graph, while subsequent receives consider all pending sends as possible rfs, thereby implicitly ordering them.

An example how lazy ordering helps decrease the state space of a program can be seen below.

Example 2.9 (Must DPOR: Lazy ordering) Consider the exploration of NS+R below, assuming that Must runs the threads in a left-to-right schedule.



The Must DPOR adds the sends of the first N threads one by one, leading to graphs A,...,B. As messages are ordered lazily, Must does not know the order in which these messages reach TN+1.

Instead, when Must adds the R of TN+1, it then considers the N different rf options for the receive, leading to graphs 1,...,N. By reading from a different message at each exploration, a different receipt order is modeled: the message read by TN+1 is the first one to arrive there, while the order among the other messages is irrelevant.

That way Must brings down the number of explored executions for NS+R from $N!$ to N.

Of course, ordering messages lazily does not mean that receives can read any message in the graph; the Must DPOR still has to ensure that all explored graphs are consistent. For instance, consider a program where T1 sends two messages to T2, which only receives one message. Given a p2p model of communication, it is inconsistent for T2 to only receive the second message, and the Must DPOR will prune such an exploration.

The biggest challenge in supporting lazy message ordering is that they complicate the condition used when backward revisiting. In the absence of eager ordering, Must's revisiting condition is rather subtle, and requires a "tiebreaker" among possible rfs; see §4 for details.

Remark. If all messages are received, ordering them lazily yields no benefit, as all possible receipt orderings have to be examined anyway. However, as we show in §6, ordering messages lazily does aid scalability in real-world implementations, e.g., because a thread prematurely terminates, and a lot of messages to it remain unread.

2.3.5 Multiple Communication Models. Another feature commonly found in real-world implementations is having multiple different communication models in the same system. For instance, a chain-replication implementation may use different protocols in various situations, e.g., to distinguish between data-replication messages and heartbeat messages (used in fault detection): a node needs to receive data-replication messages in the order in which they are sent (hence TCP is appropriate), but missing/reordered heartbeat messages does not affect correctness (hence UDP is a better choice due to its performance).

Example 2.10 (Must API: Multiple Communication Models) To further motivate support for multiple communication models, let us revisit our motivating example of Fig. 1, and more specifically the messages sent by the tail node to durable storage. In a real implementation, the durable storage might be a Paxos cluster, but here it is modeled here as a single non-failing process `storage`. Messages to `storage` are sent under a stronger communication model, namely mbox. Using mbox is important: since failures are possible, multiple nodes can play the role of the tail and it is important that the durable storage receives those messages in the global order in which they were sent. Using a weaker model like p2p provides no guarantees on the receipt order of messages coming from different nodes.

Another novelty of the Must API is that is supports such a coexistence between different models, thereby making modeling distributed systems more intuitive. We extend this support to the Must DPOR by decoupling reasoning about the algorithm correctness from a particular communication model, and rather only assume that the underlying communication model satisfies some basic assumptions (see §3). Importantly, supporting multiple communication models allows Must to lift one of DPOR's major limitations: the inability to reason about *global properties* (see below).

2.3.6 Monitoring for Temporal Properties. While DPOR is sound for safety properties that concern local states (i.e., program assertions) or "end" states reached upon program termination, previous approaches could not reason about temporal properties such as the ordering among events of different threads.

Reasoning about such orderings, however, is necessary when dealing with realistic distributed system protocols. For instance, consider again the chain replication protocol of Fig. 1. One property that we would like to verify for this protocol is that it is *strongly consistent*: the log of any node is a prefix of any log preceding in the chain, and all write requests serialize at Tail.

To understand why previous approaches cannot reason about such properties, let us consider the simpler variant of Fig. 1, S+S+R, and suppose we want to check whether the send of T1 is always executed before that of T2 (a property that clearly does not hold). The issue is that DPOR might only examine a particular scheduling between T1 and T2, because exploring them in the opposite order would result in the same execution graph. For instance, assuming DPOR schedules threads from left to right, then the send of T1 is always executed before that of T2, and DPOR will not examine the opposite ordering in order to not get any duplicate graphs. Thus, it is unsound to conclude anything about the order in which these sends execute.

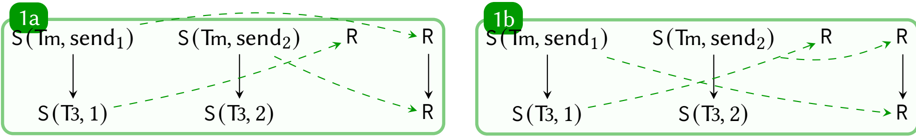
Must overcomes this issue and reasons about temporal properties via *monitors*. Monitors are special threads that receive notifications about events of interest, and throw assertion violations whenever they observe unexpected behavior. Monitors are best understood with an example.

Example 2.11 (Must DPOR: Temporal Properties) Must reasons about the ordering between the sends of T1 and T2 in S+S+R by adding a monitor thread Tm as follows:

T1: $\mathsf{send}^{cd}(\mathrm{Tm}, \mathsf{send}_1)$ $\mathsf{send}^{p2p}(\mathrm{T3}, 1)$	T2: $\mathsf{send}^{cd}(\mathrm{Tm}, \mathsf{send}_2)$ $\mathsf{send}^{p2p}(\mathrm{T3}, 2)$	T3: $\mathsf{recv}^{p2p}()$	Tm: $m_1 := \mathsf{recv}^{cd}()$ $m_2 := \mathsf{recv}^{cd}()$ $\mathsf{assert}(m_1 = \mathsf{send}_1 \wedge m_2 = \mathsf{send}_2)$

In the code above, T1 and T2 notify Tm about the messages they are about to send just before they send them, and the assertion in the monitor checks whether send_1 always arrives before send_2.

The Must DPOR explores two graphs for each execution graph in the original S+S+R program; e.g., 1a and 1b below, correspond to graph 1 of S+S+R. As can be seen, the assertion in Tm will



fail in graph 1b, from which Must concludes that the sends can be received in either order.

We can make several interesting observations about the above example. First, observe that T1 and T2 send notifications to the monitor *just before* their "original" sends (since sends are executed unconditionally). In the case of a receive, notifications would have to be sent just after it (as receives may be blocking).

More generally, we can define the functions $\mathsf{send}_{not}()$ and $\mathsf{recv}_{not}()$ that automatically notify a monitor on a given event as follows:

T: $\mathsf{send}^{M}_{not}(\mathrm{T}', m, \mathrm{Tm}) \triangleq$
 $\mathsf{send}^{cd}(\mathrm{Tm}, \mathsf{Send}(\mathrm{T}, \mathrm{T}', m))$
 $\mathsf{send}^{M}(\mathrm{T}', m)$

T: $\mathsf{recv}^{M}_{not}(\lambda x : e, \mathrm{Tm}) \triangleq$
 $m := \mathsf{recv}^{M}(\lambda x : e)$
 $\mathsf{send}^{cd}(\mathrm{Tm}, \mathsf{Recv}(\mathrm{T}, m))$

Second, also observe that in the example and definitions above, multiple communication models are used: monitor operations always take place under *causal delivery*. Intuitively, causal delivery dictates that if two send operations with the same destination are causally ordered (i.e., there is a $(\mathsf{po} \cup \mathsf{rf})^+$ path between them), then their messages will also be received in the same order. (Recall that, under p2p, only sends originating from the same process are guaranteed to have their messages delivered in order.) To see why causal delivery is required for messages to monitors, consider the following example.

Example 2.12 (Must DPOR: Monitors and cd) In the following variation of our previous example, the sends to T3 are causally related, i.e., T2 sends its message to T3 after receiving a message from T1:

T1: $\mathsf{send}^{p2p}_{not}(\mathrm{T3}, 1, \mathrm{Tm})$ $\mathsf{send}^{p2p}(\mathrm{T2}, 1)$	T2: $\mathsf{recv}^{p2p}()$ $\mathsf{send}^{p2p}_{not}(\mathrm{T3}, 2, \mathrm{Tm})$	T3: $\mathsf{recv}^{p2p}()$	Tm: $m_1 := \mathsf{recv}^{cd}()$ $m_2 := \mathsf{recv}^{cd}()$ $\mathsf{assert}(m_1 = \mathsf{Send}(\mathrm{T1}, \mathrm{T3}, 1) \wedge m_2 = \mathsf{Send}(\mathrm{T2}, \mathrm{T3}, 2))$

If we *do not* use cd for the messages sent to the monitor, then Must will erroneously determine that the assertion is violated, since the sends to Tm are performed by different processes, and can thus be received in any order under p2p. Using cd for the notifications Tm, on the other hand, guarantees that since there is a $(\mathsf{po} \cup \mathsf{rf})^+$ path between the sends to Tm (i.e., they are causally related), the sends will be received according to their causal order.

Finally, observe that monitors do not increase the state space more than necessary. Users are free to (1) use multiple monitors (one per property of interest), (2) send only a subset of the program messages to a given monitor, and (3) employ selective receives in a monitor to minimize the induced overhead (if the property checked does not rely on causality). In fact, Must performs such optimizations, and generates monitors automatically, through procedural macros (see §5).

Remark. Must (and DPOR in general) only considers safety properties. Must could handle liveness properties by reducing them to safety properties (e.g., by providing a ranking function [Yao et al. 2024] or via heuristics [C. Killian et al. 2007], but such concerns are orthogonal to our main contribution. Note that we cannot perform a lasso check as in Spin [Holzmann 1997]: getting the state of a running program is difficult without precise instruction modeling, and there is no guarantee that a specific state will be revisited (the program state space can be unbounded).

3 The Must API: Partial Order Semantics

We now present execution graphs (§3.1) formally, define our notion of consistency (§3.2), and then show how programs are mapped to sets of consistent execution graphs (§3.3).

Notation. Given a relation r, we write r^+, $r^?$ and r^* for its transitive, reflexive, and reflexive-transitive closure. We write $r_1; r_2$ for relational composition, and assume that ; binds tighter than $\cap$. Given a set of elements A, we write $[A]$ for the identity relation on that set.

3.1 Execution Graphs

As explained in §2.1.2, an execution graph G comprises a set of events and some relations on these events. We begin by defining events as follows.

Definition 3.1. An *event*, $e \in \mathsf{Event}$ is a tuple $\langle t, i, l\rangle$ where $t \in \mathsf{Tid}$ is a thread identifier, $i \in \mathsf{Idx} \triangleq \mathbb{N}$ is an index within a thread, and $l \in \mathsf{Lab}$ is a label that takes one of the following forms:

- Receive label $\mathsf{R}^a(vals)$, where $a \in \mathsf{RAttr}$ denotes the receive attributes (explained below), and $vals \in \mathcal{P}(\mathsf{Val})$ denotes the possible values that match the receive predicate;
- Send label $\mathsf{S}^{\mathsf{M}}(t, v)$, where M is the used communication model, $t \in \mathsf{Tid}$ is the destination thread identifier, and $v \in \mathsf{Val}$ is the value sent;
- Nondeterministic choice label $\mathsf{ND}^S(v)$, where $v \in S \triangleq \{v_1, \dots, v_n\}$ is the value obtained;
- Block label B; and Error label `error`.

Receive attributes in RAttr are tuples of the form $\langle \mathsf{M}, \beta\rangle$ where M denotes the communication model of the receive, and $\beta \in \{\mathsf{b}, \mathsf{nb}\}$ denotes whether the receive is blocking or non-blocking.

When applicable, the functions `tid`, `idx`, `dst`, `val`, and `vals` return the thread identifier, index, destination thread identifier, value, and matching receive values of an event, respectively. We write $\mathsf{R} \triangleq \left\{\langle t, i, l\rangle \;\middle|\; l = \mathsf{R}^{\cdot}(_)\right\}$ to denote the set of all receives, $\mathsf{S} \triangleq \left\{\langle t, i, l\rangle \;\middle|\; l = \mathsf{S}^{\cdot}(_,_)\right\}$ for the set of all sends, $\mathsf{ND} \triangleq \left\{\langle t, i, l\rangle \;\middle|\; l = \mathsf{ND}^{\cdot}(_)\right\}$ for the set of all nondeterministic-choices, and B and `error` for the set of all block and error nodes, respectively. We use subscripts and superscripts to further restrict those sets; (e.g., $\mathsf{S}_t \triangleq \left\{s \in \mathsf{S} \;\middle|\; \mathtt{dst}(s) = t\right\}$ and $\mathsf{S}^{\mathsf{p2p}} \triangleq \left\{s \in \mathsf{S} \;\middle|\; l = S^{\mathsf{p2p}}(_,_)\right\}$).

Definition 3.2. An *execution graph* G is a tuple $\langle E, po, rf\rangle$ where:

(1) E is a set of events that does not contain multiple events with the same serial number.
(2) $po \subseteq E \times E$, called the *program order*, is a strict partial order on events.
(3) $rf \subseteq \mathsf{S}_\perp \times \mathsf{R}$, called the *reads-from* relation, maps each receive event to the corresponding send from where it gets its value (or $\perp$, if no such send exists).

We write $G.\mathsf{E}$, $G.\mathsf{po}$ and $G.\mathtt{rf}$ to project graph components, $G.\mathsf{po}\mathtt{rf}$ for $(G.\mathsf{po} \cup G.\mathtt{rf})^+$, $G.\mathsf{R}$ for the set $G.\mathsf{E} \cap \mathsf{R}$ and similarly for other sets. We use $G.\mathsf{US} \triangleq \left\{s \in G.\mathsf{S} \;\middle|\; \nexists r.\, \langle s, r\rangle \in G.\mathtt{rf}\right\}$ for the set

of sends that are not read. Given a set of events E, we write $G|_E$ for the restriction of G to E, and $G \backslash E$ for the graph obtained by removing the events in E.

To ensure that receives read values that match their predicates and attributes, we introduce graph well-formedness.

Definition 3.3 (Well-formedness). Execution graph G is *well-formed* if the following hold for $G.\mathtt{rf}$:

- only non-blocking receives can read $\bot$: for all $r \in G.\mathsf{R}$ such that $\langle \bot, r \rangle \in G.\mathtt{rf}$, it is $r \in G.\mathsf{R}^{\mathsf{nb}}$
- receives read a single value: $G.\mathtt{rf}$ is functional on its range
- sends are read only once: $G.\mathtt{rf}^{-1}$ is functional on $G.\mathsf{S}$
- $G.\mathtt{rf}$ only relates sends and receives of matching destination, value, and model: for each $\langle s, r \rangle \in G.\mathtt{rf}$, if $s \neq \bot$ then $\mathtt{tid}(r) = \mathtt{dst}(s)$, $\mathtt{val}(s) \in \mathtt{vals}(r)$ and $s, r \in G.\mathsf{E}^{\mathsf{M}}$
- there are no causal cycles: $G.\mathsf{po}\mathtt{rf}$ is irreflexive

Abusing notation, we sometimes write $G.\mathtt{rf}(r)$ to get the (unique) event a receive is reading from. We also define the auxiliary *matching values* relation, $G.\mathtt{mval}$, that relates a send with receives that are satisfied by the corresponding value as $G.\mathtt{mval} \triangleq \left\{ \langle s, r \rangle \in G.\mathsf{S} \times G.\mathsf{R} \;\middle|\; \mathtt{val}(s) \in \mathtt{vals}(r) \right\}$.

Observe that, in addition to the conventions of §2, when drawing graphs, we omit (a) transitive $G.\mathsf{po}$ edges, (b) thread identifiers (as they are inferred from the context), and (c) serial numbers.

3.2 Message-Passing Consistency

We now define what it means for an execution graph to be consistent. As a graph might contain events that operate under different semantics, a graph G is consistent if for every communication model M, the restriction of G to events of M satisfies the corresponding consistency predicate.

Thus, we begin by defining consistency predicates for the usual models of fully asynchronous communication (asyn), peer-to-peer (p2p), causal delivery (cd), and mailbox communication (mbox).

Fully asynchronous communication only requires that there are no causal cycles.

Definition 3.4. A graph G is consistent under asyn, written $\mathsf{consistent}_{\mathsf{asyn}}(G)$, if it is well-formed.

Moving on to p2p consistency, whenever a process sends two messages to the same process, these have to be received in order.

Definition 3.5. A graph G is consistent under p2p, written $\mathsf{consistent}_{\mathsf{p2p}}(G)$, if (a) it is well-formed, (b) $[\mathsf{US}]; G.\mathtt{so}|_{\mathtt{dst}}; G.\mathtt{rf} \cap \mathtt{mval}$ is empty, and (c) $G.\mathsf{po}; G.\mathtt{rf}^{-1}; (G.\mathtt{so}|_{\mathtt{dst}}; G.\mathtt{rf} \cap \mathtt{mval})$ is irreflexive, assuming that $\mathtt{so} \triangleq [\mathsf{S}^{\mathsf{p2p}}]; \mathsf{po}; [\mathsf{S}^{\mathsf{p2p}}]$, and $\mathtt{so}|_{\mathtt{dst}}$ denotes $\mathtt{so}$ edges with the same destination.

Intuitively, condition (b) precludes graphs where a receive r reads from a send s and there exists an unread send s' po-before s that matches r's predicate. Condition (c) precludes graphs where s' is read by some po-successor of r (i.e., sends to the same process are received out of order).

Causal delivery dictates that messages that are causally related (i.e., there is a $\mathsf{po}\mathtt{rf}$ path between them) are received in the order they are sent. Its definition is easily obtained by modifying Def. 3.5 to use $\mathtt{so} \triangleq \mathsf{po}\mathtt{rf}$ (and restricting to cd events).

Definition 3.6. A graph G is consistent under cd, written $\mathsf{consistent}_{\mathsf{cd}}(G)$, if (a) it is well-formed (b) $[\mathsf{US}]; G.\mathtt{so}|_{\mathtt{dst}}; G.\mathtt{rf} \cap \mathtt{mval}$ is empty, and (c) $G.\mathsf{po}; G.\mathtt{rf}^{-1}; (G.\mathtt{so}|_{\mathtt{dst}}; G.\mathtt{rf} \cap \mathtt{mval})$ is irreflexive, assuming that $\mathtt{so} \triangleq [\mathsf{S}^{\mathsf{cd}}]; \mathsf{po}\mathtt{rf}; [\mathsf{S}^{\mathsf{cd}}]$.

In mbox, any two messages to the same process have to be received in the order they were sent.

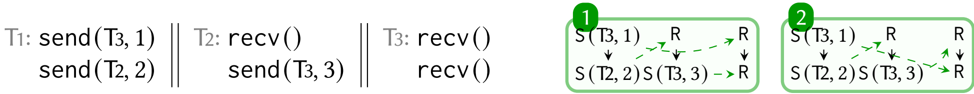
Definition 3.7. A graph G is consistent under mbox, written $\mathsf{consistent}_{\mathsf{mbox}}(G)$, if (a) it is well–formed, (b) $[\mathsf{US}]; G.\mathtt{so}|_{\mathtt{dst}}; G.\mathtt{rf} \cap \mathtt{mval}$ is empty, and (c) $G.\mathsf{po}; G.\mathtt{rf}^{-1}; (G.\mathtt{so}|_{\mathtt{dst}}; G.\mathtt{rf} \cap \mathtt{mval})$ is irreflexive, for some $\mathtt{so}$ total over $G.\mathsf{E}^{\mathsf{mbox}} \times G.\mathsf{E}^{\mathsf{mbox}}$.

Even though Def. 3.7 above assumes the existence of a total order on the events of an execution graph, our algorithm in §4 does not enumerate all such total orders. Instead, it adapts the equivalent reformulation of mailbox semantics using partial orders of Di Giusto et al. [2023, Def. 4.2].

Definition 3.8 (Consistency). An execution graph G is consistent, written consistent(G), if it is well-formed and consistent$_{\mathsf{M}}(G)$ holds for each model M used.

We write consistent$_{\mathsf{M}_1,\ldots,\mathsf{M}_n}(\cdot)$ for a consistency predicate parameterized by models $\mathsf{M}_1, \ldots, \mathsf{M}_n$. As the example below shows, the communication model determines the set of consistent graphs.

Example 3.1 Consider the program below along with its p2p-consistent graphs.



If we change the communication model to cd, the graph 2 becomes inconsistent: there is a porf path between the two sends to T3, but the messages are received in the opposite order.

3.2.1 Consistency Requirements. Must is parametric in the choice of the communication model (and different events can operate under different models), but its correctness depends on a number of assumptions on the overall model (i.e., on graph consistency). We now enumerate these assumptions:

No "Out-of-Thin-Air": There are no causal cycles. Formally, given a graph G with consistent(G), then G.porf is irreflexive.

Prefix-closedness: Restricting a consistent graph to a porf-prefix-closed subset of its events preserves consistency. Formally, given a graph G such that consistent(G), then for any $E \subseteq G.\mathsf{E}$ such that $dom(G.\mathsf{porf}; [E]) \subseteq E$, it holds that consistent$(G|_E)$.

Conditional Extensibility: A program should not get stuck if (a) a thread has more instructions to execute, and (b) there is some message to receive (if the next instruction is a receive). Formally, given a graph G and a G.po-maximal event $e \in G.\mathsf{E}$, if consistent$(G \setminus \{e\})$, either
(i) $e \notin \mathsf{R}$ and consistent(G),
(ii) $e \in \mathsf{R}$ and consistent(G'), where G' is equal to G apart from the rf of e, or
(iii) $e \in \mathsf{R}^{\mathsf{b}}$ and for all $s \in G.\mathsf{S}_{\mathtt{vals}(e)}$ there exists $r' \neq e$ such that $G.\mathsf{rf}(r') = s$.

Observe that the conditions above imply that all events apart from blocking receives can always be added in a consistent manner, and that the consistency predicates of well-known models like asyn, p2p, cd, and mbox all satisfy the above assumptions.

Moreover, it easily follows that if each model M used in a program satisfies Prefix-closedness and Conditional Extensibility, consistency in general (Def. 3.8) satisfies them as well (No "Out-of-Thin-Air" is satisfied by default).

3.3 Programs

As Must can be integrated into any mainstream programming language, we refrain from providing a concrete syntax here. Instead, we only assume that the following commands extend the underlying programming language, and are responsible for the production of ND, R and S labels, respectively[3]:

$$\mathsf{Cmd} \ni c ::= \ldots \mid \mathtt{nondet}(\{v_1, \ldots, v_n\}) \mid \mathtt{send}^{\mathsf{M}}(e_1, e_2) \mid \mathtt{recv}^{a}(\lambda x : e)$$

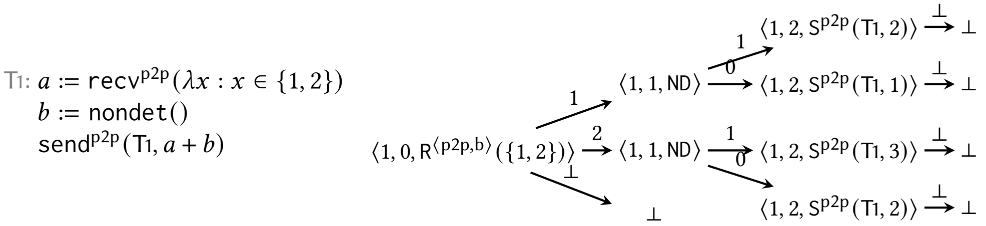
[3] B,`error` can be produced either by jumping to special addresses, or by providing new primitives; we assume the former.

We assume that $v \in \mathsf{Val}$ ranges over values, M over communication models, a over receive attributes (see §3.1), $t \in \mathsf{Tid} \triangleq \{1, \dots, N\}$ over a finite set of thread identifiers, and $e \in \mathsf{Expr}$ over the underlying language of arithmetic expressions.

For simplicity, we further assume that, programs are of the form $\|_{i \in \mathsf{Tid}}\ i : P_i$, and each P_i is a sequential bounded program[4], in turn represented as a function that, given a (partial) program trace, returns the event corresponding to the next action to be executed. Formally, a trace $tr \in \mathsf{Trace}$ is a finite sequence of values $v \in \mathsf{Val}_\bot$, and P_i is a partial function $P_i : \mathsf{Trace} \rightharpoonup \mathsf{Event}_\bot$.

Intuitively, a trace is a sequence of values that the Must API primitives return: the values read by receives (including $\bot$), and the values returned by non-deterministic choice (`send()` returns $\bot$). We assume that error, block, and blocking receive labels that read $\bot$ block the execution: if $P_i(tr) \in \mathtt{error} \cup \mathsf{B} \cup \mathsf{R}^{\mathsf{b}}$, then $P_i(tr \cdot \langle \bot \rangle) = \bot$.

Example 3.2 (Semantics of sequential programs) Let us demonstrate the semantics of a sequential program P_i, assuming P_i is represented as labeled transition system.



Above, each transition is labeled with the values that the corresponding state may return, and terminal states are denoted with $\bot$. Receive and non-deterministic states have multiple edges, while all other states have a single outgoing edge. Selective receives only have edges corresponding to the values that satisfy the receiver's predicate, and all receives also have $\bot$ as an edge. Observe that the transition system above can be equivalently be seen as a function from traces to events.

To map programs to execution graphs, we first have to define how traces are extracted from graphs. We define the function $\mathtt{trace}_G(i) \triangleq \langle G.\mathtt{val}(e_1), \dots, G.\mathtt{val}(e_n) \rangle$ to return the trace corresponding to process i: $e_k = \langle i, k, _ \rangle$ is the k-th event in thread i (according to po).

Then, we say that G is an *execution* of $P = \|_{i \in \mathsf{Tid}}\ i : P_i$ if, for every $i \in \mathsf{Tid}$ and proper prefix tr_n of length n of $\mathtt{trace}_G(i)$, it is $P_i(tr_n) = e_{n+1}$, where $e_{n+1} = \langle i, n+1, _ \rangle$ is the $(n+1)$-th event of thread i in G. We also say that G is a *full execution* of P if $P_i(\mathtt{trace}_G(i)) = \bot$ for every $i \in \mathsf{Tid}$.

The semantics of a program P, written $[\![P]\!]$, is then the set of consistent execution graphs (Def. 3.8) S such that each $G \in S$ is either a full execution of P, or there exists a full execution G' of P such that $G = G'|_{G.\mathsf{E}}$, and extending G with any event of G' renders G inconsistent. (The last case is necessary as the full executions of a given program might contain blocking receives that read $\bot$, which renders them inconsistent.) In turn, a program P is *erroneous* if $[\![P]\!]$ contains an execution graph with an `error` label.

4 The Must DPOR Algorithm

We now formally describe the Must DPOR and prove it sound, complete and optimal.

Algorithm. Given a concurrent program P together with the consistency predicates for all communication models used, Must explores all consistent execution graphs of P.

[4]A program is bounded either by statically unrolling it, or by dynamically bounding the number of steps of each thread.

Algorithm 1 Must: Optimal Dynamic Partial Order Reduction for Message-Passing Concurrency

1: **procedure** $\textsc{Verify}(P) = \textsc{Visit}_P(G_\emptyset)$

2: **procedure** $\textsc{Visit}_P(G)$
3:     **switch** $e \leftarrow \mathsf{next}_P(G)$ **do**
4:         **case** $e = \bot$ **return** "Visited full execution graph G"
5:         **case** $e \in \mathtt{error}$ **exit**("error")
6:         **case** $e \in \mathsf{ND}^S$ **for** $v \in S$ **do** $\textsc{Visit}_P(\mathsf{SetND}^S(G, e, v))$
7:         **case** $e \in \mathsf{R}$ **for** $s \in G.\mathsf{S} \cup \{\bot\}$ **do** $\textsc{VisitIfConsistent}_P(\mathsf{SetRF}(G, e, s))$
8:         **case** $e \in \mathsf{S}$
9:             $\textsc{VisitIfConsistent}_P(G)$
10:             **for** $r \in \left\{e' \in G.\mathsf{R}_{\mathtt{dst}(e)} \;\middle|\; \mathtt{val}(e) \in \mathtt{vals}(e') \land \langle e', e\rangle \notin G.\mathsf{porf}\right\}$ **do**
11:                 $\mathit{Deleted} \leftarrow \{e' \in G.\mathsf{E} \mid r <_G e' \land \langle e', e\rangle \notin G.\mathsf{porf}\}$
12:                 **if** $\forall e' \in \mathit{Deleted} \cup \{r\}.\ \textsc{RevisitCondition}(G, e', e)$ **then**
13:                     $\textsc{VisitIfConsistent}_P(\mathsf{SetRF}(G|_{G.\mathsf{E}\setminus \mathit{Deleted}}, r, e))$
14:         **case** _ $\textsc{Visit}_P(G)$

15: **procedure** $\textsc{VisitIfConsistent}(G)$
16:     **if** $\mathsf{consistent}(G)$ **then** $\textsc{Visit}_P(G)$

17: **procedure** $\textsc{RevisitCondition}(G, e, s)$
18:     **if** $e \in \mathsf{R}^{\mathsf{nb}}$ **then return** $G.\mathsf{rf}(e) = \bot$
19:     **if** $e \in \mathsf{ND}^S$ **then return** $\mathtt{val}(e) = \mathsf{min}(S)$
20:     $\mathit{Previous} \leftarrow \{e' \in G.\mathsf{E} \mid e' \leq_G e \lor \langle e', s\rangle \in G.\mathsf{porf}\}$
21:     **if** $e \notin \mathsf{R}$ **then return** $\forall r \in \mathit{Previous}$ it is $G.\mathsf{rf}(r) \neq e$
22:     **return** $G.\mathsf{rf}(e) = \textsc{GetConsTiebreaker}(G|_{\mathit{Previous}}, e)$

To do so, Must endows the definition of an execution graph G with an *insertion order* $\leq$, a total order on $G.\mathsf{E}$ representing the order in which events were incrementally added to the graph. In what follows, we write $\leq_G$ to project to the insertion order of a given graph G; given two events $e_1, e_2 \in G.\mathsf{E}$, we write $e_1 <_G e_2$ if $e_1 \leq_G e_2$ and $e_1 \neq e_2$.

The entry point of Must is the function Verify (algorithm 1). Verify visits all graphs of P in a depth-first manner by calling Visit on the empty execution graph $G_\emptyset$ (line 1).

Visit is integral to the verification procedure. As long as the current graph G remains consistent, Visit inspects P, appropriately extends G with the next program event by calling $\mathsf{next}_P(G)$ (line 3), and returns the added event e. We assume that $\mathsf{next}_P(G)$ (i) does not pick events from threads that are blocked (denoted by a B event in G), (ii) does not add a receive if there are no messages to read, and (iii) returns $\bot$ when there are no more events to be added.

The type of e determines the next step of Visit.

If e is $\bot$ (line 4), Must has visited a full execution and returns.

If e denotes an error such as an assertion violation (line 5), Must exits and reports an error.

If e represents a nondeterministic choice (line 6), Must recursively explores both options for the returned value (line 6). Formally, assuming $e = \langle t, i, _\rangle$, $\mathsf{SetND}^S(G, e, v)$ returns a new graph G' where the value of the nondeterministic choice event e has been set to v as follows:

$$G'.\mathsf{E} = G.\mathsf{E} \setminus \{e\} \cup \{\langle t, i, \mathsf{ND}^S(v)\rangle\}$$

If e is a receive (line 7), Must has to examine all possible rf options for it. As such, for each send s in G (line 7), Must recursively explores the case where e reads from s (line 7). Must also considers a $\perp$ option for e (line 7), without first ensuring that e is a non-blocking receive. That is because such checks are part of the consistency checks: it is inconsistent for a blocking receive to read $\perp$ (due to well-formedness; see §3.2), and thus such executions will be ruled out by the consistency checks of line 16. (Consistency checks also ensure that s is a send with the right destination, its value matches the receive predicate, etc.) Formally, $\mathsf{SetRF}(G, e, s)$ returns a new graph G' that is equal to G except that e is reading from s:

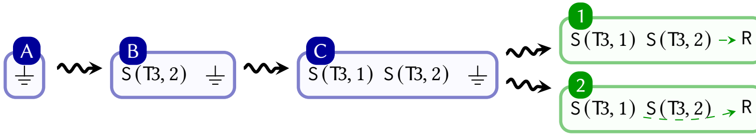
$$G'.\mathsf{rf} = G.\mathsf{rf} \setminus (\mathsf{E}_\perp \times \{e\}) \cup \{\langle s, e\rangle\}$$

Finally, in the last two cases of Visit (line 8, line 14), Must handles the case where e is a send or some other event, respectively. The case where e has any other type (e.g., block) is simple; Visit simply recursively calls itself (line 14).

The case when e is a send, however, requires some care, since Must has to explore both the scenario where e is merely added in the graph (line 9), as well as the one where e backward-revisits some previously added receive.

Let us now focus on the revisiting case. Must has to ensure that it explores all graphs (for completeness) but does not end up exploring the same graph twice (for optimality). The reason why backward revisits make this challenging is best understood with some concrete examples.

Example 4.1 (Rescheduling receives) Consider an exploration of s+s+r where T3 is scheduled before T1 and T2.



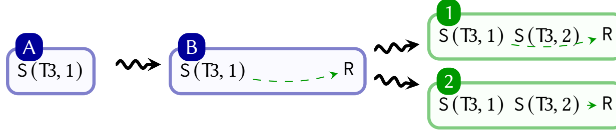
In the exploration above, when Must tries to add the receive of T3, no send to T3 has been added to the graph yet. To deal with this issue, instead of inserting a receive to the graph reading from nowhere, Must blocks T3 and schedules a different thread (graph A). (The ⏚ symbol denotes that T3 is waiting on a blocking receive.) With T3 blocked, Must has no other option but to continue the exploration with the other threads, and adds the labels S(T3, 1) and S(T3, 2) in some order.

At this point, Must has explored all instructions of the program (graph C). Before terminating, however, it checks whether some thread is blocked on a receive that can now be unblocked (due to new sends appearing). Indeed, Must sees that there are two new sends for T3, adds the corresponding R, and then recursively explores both rf options for it (graphs 1 and 2).

As already hinted at in §2, such rescheduling of receives is necessary for soundness: adding a receive reading $\perp$ would lead to the exploration of inconsistent graphs e.g., in the case where the receive is blocking and no send ever appears.

Unfortunately, however, rescheduling is insufficient to guarantee completeness. The problem is that when Must adds a receive event, only a subset of the possible sends to the receiving thread may have been added to the graph. The example below demonstrates this issue.

Example 4.2 (Backward revisiting) Consider an exploration of s+s+r where T1 and T3 are scheduled before T2.



As can be seen from graphs A and B, Must first adds S(T3, 1) and then proceeds by adding the R of T3, which can only read 1 (the only message available in the graph). If Must takes no further action, when S(T3, 2) is added, it will not explore the case where T3 reads 2.

To address this issue, after adding S(T3, 2), Must *backward-revisits* the existing receive node. Concretely, when S(T3, 2) is added, Must considers both the case where it does not revisit any event (leading to graph 1), as well as the case where it revisits the R event.

In the latter case, following recent DPOR approaches (e.g., [Kokologiannakis et al. 2022]), Must *restricts* the execution graph so that it only contains the events that were added before R, as well as the porf-prefix of S(T3, 2), yielding graph 2 of S+S+R. (Even though the graph obtained by revisiting contains the same events, as we will shortly see, this is not generally the case.)

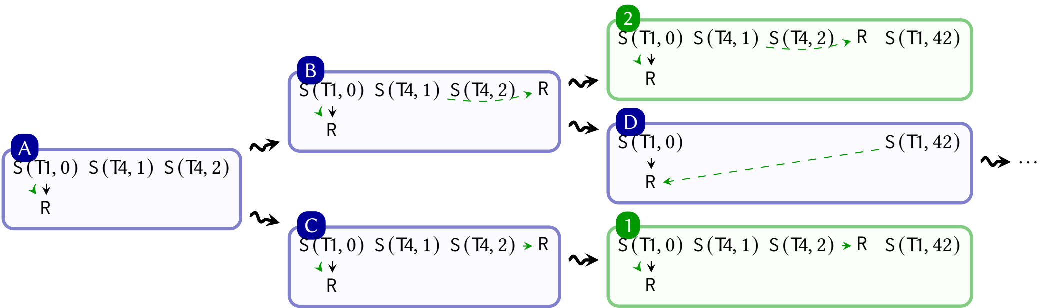
Backward revisiting suffices to recover completeness. However, one has to take extra care to ensure optimality. The problem here is that backward revisits from different subexplorations might lead to the same graph; we demonstrate this issue with an example.

Example 4.3 (Revisiting condition) Consider the following variant of S+S+R, with the newly-added leftmost and rightmost threads (T1 and T5) communicating among themselves:

T1: send(T1, 0) recv()	T2: send(T4, 1)	T3: send(T4, 2)	T4: recv()	T5: send(T1, 42)	(S+S+R-BR)

Note that the receive of T1 will receive one of two messages: the one from itself, or the one from T5.

The problem with the above variant is that, assuming a left-to-right exploration order, Must will encounter two *different* explorations where send(T1, 42) can backward-revisit the receive of T1 (graphs B and C below).



In graph B the receive of T4 is reading 1, while in graph C it is reading 2. In both cases, however, when the send of T5 is added, it will have the chance to revisit the receive of T1, and in both cases the graph resulting from the backward revisit will be graph D. (Recall that the graph resulting from a backward revisit contains the events that were added before the revisited receive, as well as the porf-prefix of the revisiting send.)

This poses a problem: if Must backward-revisits R in both explorations, it will explore graph D twice. In order to avoid storing all previously explored graphs, Must constrains which backward revisits are performed using a *revisiting condition*. In the case of the example above, Must's revisiting condition ensures that the revisit will only take place from graph B.

Let us now examine the revisit condition in more detail. Consider the graph G' that occurs if a send s revisits a receive r. From prefix-closedness (see §3.2.1), $G_{rs} \triangleq G' \setminus \{r, s\}$ is also consistent. Now, if we start from G_{rs} and try to add the remaining events using the insertion order prescribed by $\mathsf{next}_P()$ until s is encountered, we can of course arrive in many different graphs depending on the values read by the receives. Must's revisiting condition ensures that the revisit will only take place from one of these graphs, namely the one constructed when no backward revisit is performed, and where each receive reads some from some consistent send (if there are many such sends Must deterministically picks a "tiebreaker"; see below). Observe that the graph from which Must performs the revisit is unique (there is a single `rf` for each receive that satisfies the condition) and always exists (guaranteed by extensibility; see §3.2.1).

Returning to algorithm 1, Must first collects all receives that can be revisited by e without creating a `porf` cycle (line 10), and then ensures that each event that will be deleted by the revisit (line 11) satisfies the condition described above.

The last check is performed with the help of RevisitCondition (line 12), which distinguishes four cases. Non-blocking receives satisfy the condition if they are reading bottom (line 18). Nondeterministic choices satisfy the condition if they return the minimum value among all possible options (line 19). All other non-receive events trivially satisfy the condition as long as they are not sends that have revisited some receive that was added earlier starting from G_{rs} (line 21). Receive events satisfy the condition if they are reading a "tiebreaker" value decided by the GetConsTiebreaker oracle (line 22). In terms of implementation, GetConsTiebreaker(G, e) can simply return e.g., the `tid`-minimal send for which it is consistent for e to read from. For the well-known models asyn, p2p, cd, and mbox, we define a partial order between unread sends to e that match its predicate such that e can consistently read any of the minimal ones. Then, we use the `tid` to break the tie in case there are multiple minimal ones.

Soundness, Completeness, Optimality. Given a program P that uses a set of communication models which satisfy the requirements of §3.2.1, algorithm 1 is sound (i.e., does not explore inconsistent executions), complete (i.e., explores all consistent executions of P), and optimal (i.e., explores each consistent execution once, and does not start wasteful explorations). We now formally state our results. (Full proofs can be found in our supplementary material.)

Must is trivially sound: it only constructs consistent executions, since inconsistent executions are dropped as soon as they are reached (algorithm 1, line 16). The following theorem summarizes completeness and optimality; we write $G_1 \approx G_2$ if the two graphs are equal up to the $\leq_G$ component (if present), i.e., they agree on all other components.

Theorem 4.1. *Let G_f be some graph in $[\![P]\!]$.*
Completeness: Verify(P) calls Visit$_P(G'_f)$ for some $G'_f \approx G_f$.
Optimality: Verify(P) will never call Visit (a) for graphs G_1, G_2 such that $G_1 \approx G_2$, or (b) for a graph G that cannot lead to an execution $G_f \in [\![P]\!]$.

5 Implementation

We implemented Must as a tool for Rust programs. While in principle we could have used any language to implement Must, we chose Rust because (a) using Rust we could easily enforce Must's

requirement that all inter-process communication takes place via the Must API (see below), and (b) our goal was to verify existing distributed-system protocols written in Rust (see §6).

The overall architecture of Must closely follows that of other stateless model checkers such as CDSChecker [Norris and Demsky 2013], Loom [*Loom* 2023], and Shuttle [*Shuttle* 2023], where the runtime and the model checker can be seen as coroutines. The model checker invokes the runtime so that the program is executed, and the runtime informs the model checker about communication events (e.g., message sends, receives). When a full execution is obtained, the model checker checks whether there remain behaviors to be explored, and if so, it invokes the runtime again.

We aimed to make our implementation usable in several different ways.

First, to make interfacing with Rust programs smoother, we implemented Must's API as a library. In turn, this library is implemented using Rust generics so that it works for arbitrary user datatypes: if a datatype satisfies certain traits, it automatically qualifies for inter-process communication.

Second, to enforce that all inter-process communication takes place via the Must API, Must provides its own version of `std::sync::atomic`, which forbids communication using shared-memory operations. Of course, one could still use `unsafe` to enable shared-memory communication (unless `unsafe` is statically prohibited), but the code we verified did not make any use of unsafe.

Third, to allow for easier integration with existing Rust code, Must provides its own version of concurrency primitives commonly used in Rust, including `std::thread` and `mpsc::channel`.

Fourth, even though we did not need any shared-memory communication to verify distributed systems (hosts over networks cannot share state in memory), we found it useful to model a limited amount of shared-memory synchronization, as such synchronization shows up in test harnesses and local processing. As such, we equipped Must with abstractions for a shared register and a lock, which we implemented using message passing primitives. In this scheme, a register is a process that stores the current value, while other processes manipulate the register by sending read/write requests (the lock is implemented in a similar fashion). Using these abstractions, we also provided implementations of common Rust data structures such as `Arc`, `Mutex` and `oneshot`[5].

Finally, to facilitate specification writing and avoid verbosity, we implemented monitors using Rust procedural macros. The attributes of a macro define the message types for which the monitor is notified automatically in case of sends. The macro itself declares two traits: (a) a trait for notification handlers the implementation of which must be defined by the user, and (b) a "filter" trait whose purpose is to restrict the messages that are observed by the monitor. The latter is important for efficiency: as explained in §2.3.6, Must has to enumerate all possible orders in which a monitor receives notifications, and so filtering some messages out aids scalability.

6 Evaluation

We now set out to evaluate the performance of Must, by answering the following questions:

§6.1: Does Must scale better than tools with less expressive APIs or exponential memory use?
§6.2-§6.3: Does Must scale up to the verification of industrial-strength protocols?

To answer the first question, we evaluate Must against the following state-of-the-art DPOR-based tools on a set of synthetic benchmarks, and the Chord protocol for distributed hash tables [Stoica et al. 2003]: (a) TruSt [Kokologiannakis et al. 2022], an optimal DPOR for C/C++ programs under shared-memory concurrency, and (b) Concuerror an optimal DPOR for Erlang programs under message-passing concurrency. We compare against TruSt to show the importance of native message-passing support (as opposed to encoding message-passing in shared memory), and against

[5]For programs making heavy use of these features, we could extend the Must DPOR to directly handle shared reads/writes without losing soundness/completeness/optimality, in a fashion similar to TruSt [Kokologiannakis et al. 2022]. As the code we verified did not make heavy use of these features, we were satisfied with the above abstractions.

Table 1. Must vs state-of-the-art (synthetic benchmarks)

	TruSt				Concuerror		Must
	Execs	+	*Blocked*	*Mem*	*Execs*	*Mem*	*Mem*
`ns+r(2)`	4	+	2	68MB	2	90MB	19MB
`ns+r(5)`	600	+	120	68MB	5	90MB	19MB
`ns+r(8)`	322 560	+	40 320	68MB	8	90MB	19MB
`ns+nr-sel(2)`	2	+	5	60MB	1	90MB	19MB
`ns+nr-sel(5)`	42	+	1066	68MB	1	90MB	19MB
`ns+nr-sel(8)`	1430	+	590 251	68MB	1	86MB	19MB
`ns+nr(2)`	4	+	8	68MB	2	84MB	19MB
`ns+nr(5)`	5040	+	25 200	68MB	120	91MB	19MB
`ns+nr(8)`	🕒	+	🕒	🕒	40 320	91MB	19MB
`nworkers(7)`	🕒	+	🕒	🕒	10 080	195MB	19MB
`nworkers(8)`	🕒	+	🕒	🕒	80 640	550MB	19MB
`nworkers(9)`	🕒	+	🕒	🕒	725 760	OOM	19MB

Concuerror to demonstrate the importance of polynomial memory consumption (Concuerror's algorithm may consume an exponential amount of memory). For this part of our evaluation, as Concuerror does not support messages with different semantics in the same program, we ran Must and Concuerror under p2p semantics.

To answer the second question, we conduct two case studies. Along with development teams, we used the Must API to model two systems employed in production, and then verified them using the Must DPOR[6].

Overall, we answer both questions positively: Must is the only tool that consistently performs well in terms of both memory and time, and is able to verify complex systems designed by the Must API. Moreover, all features of the Must API are crucial when verifying systems of larger scale.

Experimental Setup We conducted all experiments on a MacBook Pro with 2GHz Quad-Core Intel Core i5 (4 cores) and 16GB RAM. Times are in seconds. Timeout: 30 minutes; memory limit: 1GB.

6.1 Must vs State-of-the-Art

We answer the first question using synthetic benchmarks so that we can better evaluate the different aspects of the tools (e.g., memory consumption). As each tool verifies programs in a different language, we do not provide their running times, but rather the executions explored. For TruSt, we used lock-based queues to model communication among threads (receives and sends are modeled as dequeues and enqueues, respectively). We also used `assume()`s to restrict the shared-memory encoding to valid outcomes, leading to blocked executions (reported separately).

Our results are summarized in table 1: in a nutshell, Must is the only tool that consistently performs well across the board. We now focus on individual benchmarks.

The first three benchmarks demonstrate why a message-passing encoding is superior to a shared-memory one. Starting from `ns+r(N)` of §2.3.4, TruSt explores $N \cdot N!$ executions: as the dequeue conflicts with all enqueues in the shared-memory encoding (due to lock interference), TruSt has to order it w.r.t. all enqueues (N possibilities), as well as w.r.t. each possible item enqueue order ($N!$ possibilities). In fact, TruSt has to do this despite the fact that the dequeue will always return the same value (i.e., the first one enqueued). What is even worse, however, is that TruSt also explores $N!$ blocked executions for `ns+r(N)`, representing the cases where the dequeue is executed on an empty queue. The benchmarks `ns+rn-sel(N)` and `ns+rn(N)` of §2 demonstrate how "harmful" blocked executions can be, as TruSt explores exponentially more blocked executions than normal ones. Must and Concuerror, on the other hand, explore N, 1 and $N!$ executions for these three benchmarks, respectively (i.e., the minimum number required to cover all program behaviors).

[6]We also ran Must on other classic distributed protocols such as Two-Phase-Commit and Paxos and obtained similar results.

Table 2. Case study #1: A chain replication protocol (~520 LoC) and an industrial replicated log (~2250 LoC)

	Must/asyn		Must/p2p-eager		Must/p2p		Must/cd		Must/mbox	
	Execs	*Time*	*Execs*	*Time*	*Execs*	*Time*	*Execs*	*Time*	*Execs*	*Time*
`chain(0,3)`	5340	3	719	1	540	4	288	1	288	2
`chain(0,10)`	1 957 020	699	719	2	540	4	288	1	288	47
`chain(1,3)`	212 732	27	168 001	21	22 956	4	8910	2	8910	60
`chain(1,4)`	2 275 258	354	163 222	25	31 620	6	13 916	3	13 916	209
`chain(2,3)`	4 722 338	531	5 274 403	551	367 174	40	104 936	13	104 936	857
`chain(2,4)`	128 591 662	18 895	46 074 076	5962	980 218	129	349 238	48	349 238	6139
`rlog(1,3)`	594	1	607	2	477	1	244	1	210	33
`rlog(1,4)`	7453	3	10 040	4	5862	3	3053	2	2462	709
`rlog(2,3)`	12 588	5	8911	4	5723	2	2483	1	1890	668
`rlog(2,4)`	168 758	65	143 096	58	74 173	30	28 581	12	19 204	15 077
`rlog-nonsel(1,3)`	18 899	7	1932	1	1389	1	325	1	264	76
`rlog-nonsel(2,3)`	41 027 836	13 530	94 513	39	56 188	21	5116	2	3403	1861

`chain(F,N)`: `N` nodes serve three concurrent client requests with `F` faults. (The protocol is buggy under asyn.)
`rlog(F,N)`: `N` nodes serve one client request with `F` faults. (The `nonsel` version does not employ selective receives.)

The last benchmark demonstrates the importance of polynomial memory consumption. In `nworkers(N)`, a main thread spawns N workers who send messages to a coordinator. Upon receiving all messages, the coordinator sends a message to the main thread, which in turn can receive either a message it previously sent to itself, or the message of the coordinator. Concuerror exceeds the memory limit for 8 workers, while Must maintains a stable memory consumption. Although TruSt also maintains a stable memory consumption, it times out due to its shared-memory encoding.

At this point, one might wonder whether such exponential behaviors occur in practice. To answer this question, we modeled the Chord protocol for distributed hash tables [Stoica et al. 2003] in Must, and compared the performance of Must with that of Concuerror on an independent implementation of Chord in Erlang. As the implementations were different, we focused on the memory consumption of the two tools for one client request, and $N \in \{3, 4, 5\}$ processes.

Our results confirm that memory consumption does matter in practice: Concuerror's memory grew from 224M for 3 processes, to 1.4G for 4 processes, and to 3.1G for 5 processes before timing out. By contrast, Must only used 24M for all configurations, and this number did not increase even after increasing the timeout to 72 hours (at which point Must had explored over 150M executions).

6.2 Case Study #1: Model Checking Replicated Log Protocols

In our first case study, we use Must to verify models of two replicated logs: a chain replication protocol by Renesse and Schneider [2004] (the toy version of which we described in §2), and a proprietary, industrial-scale replicated log, which serves as the backend for a number of highly scalable, customer-facing applications. The latter model was designed along with its original developers, and both models extensively use all features of the Must API.[7]

Running Must on both protocols under different communication models yields the results depicted in table 2. As expected, with the exception of `chain(0,N)` and mbox (see below), as the communication model becomes stronger, the number of executions drops and Must becomes more efficient. For `chain(0,N)` the performance of Must does not change for p2p and stronger models, as the number of executions is independent of the chain length when there are 0 faults. For the asyn case, Must explores more executions since the protocol is buggy: in contrast to `rlog` (which is designed to be correct under asyn), Must reports a violation under asyn within a few seconds, and hence the numbers we report are with all safety checks removed.

[7]All non-proprietary code used in our case studies can be found in our supplementary material.

Table 3. Case study #2: A leader election protocol (~600 LoC) and a distributed commit protocol (~900 LoC)

	Must/asyn		Must/p2p-eager		Must/p2p		Must/cd		Must/mbox	
	Execs	*Time*	*Execs*	*Time*	*Execs*	*Time*	*Execs*	*Time*	*Execs*	*Time*
`leader(2,2)`	20 256	3	13 600	4	10 128	2	10 128	2	10 128	12
`leader(2,3)`	408 120	42	105 912	12	68 020	8	68 020	8	68 020	94
`leader(3,2)`	16 240 896	1619	20 167 641	2966	8 120 448	761	8 120 448	910	◷	◷
`leader(3,3)`	459 884 790	44 620	◷	◷	76 647 465	7290	76 647 465	10 138	◷	◷
`leader(2,2)+mon`	22 506	7	16 858	5	11 253	3	11 253	3	11 253	19
`leader(2,3)+mon`	481 890	59	132 119	18	80 315	11	80 315	11	80 315	177
`leader(3,2)+mon`	24 048 748	2988	35 066 847	4296	12 024 374	1502	12 024 374	1512	◷	◷
`commit(2,3)`	157 057	3	14 107	2	14 107	1	5989	1	5989	2
`commit(2,3)`	3 856 393	60	334 171	7	334 171	6	138 601	3	138 601	14
`commit(3,2)`	737 518 375	15 684	2 445 031	53	2 445 031	46	736 957	15	736 957	167
`commit(3,3)`	◷	◷	283 732 579	8050	283 732 579	6008	83 377 729	1723	◷	◷

`leader(N,I)`: `N` nodes increment a global clock `I` times with values non-deterministically chosen from {1,2,3}.
`leader(N,I)+mon`: leader election with a monitor that tracks leader announcements and checks uniqueness.
`commit(T,K)`: Two committers serve `T` transactions over `K` keys.

As far as Must under mbox is concerned, Must is slower due to much more complex consistency checking: while for models weaker than mbox consistency checking can be done very efficiently by computing transitive closures via vector clocks, it is unclear whether this is possible for mbox.

We conclude this case study with two observations underlining the importance of lazy message ordering and selective receives in Must, respectively. To demonstrate the benefits of lazy message ordering (see §2.3.4), we added support in Must for an eager version of p2p (namely p2p-eager), which tracks the order in which messages arrive to a given node, irrespective of whether the messages are read or not. As can be seen by comparing the columns p2p and p2p-eager, lazy message ordering yields a significant improvement. To demonstrate the benefits of selective receives, we wrote a variation of `rlog` (namely `rlog-nonsel`), which does not use selective receives to avoid receiving "stale" messages (as e.g., in example 6). As it can be seen, the reduction when using selective receives is exponential, especially for weaker models such as asyn and p2p.

6.3 Case Study #2: Model Checking a Distributed Commit Protocol

Moving on to our second case study, we now use Must to verify a model of the distributed commit logic in an industrial distributed transaction management system. In the protocol, client transactions are routed to a number of committer nodes, which are in charge of a sharded keyspace. The committers coordinate on whether to commit or abort the transaction.

Our model has two parts, and was also developed in close collaboration with the respective development teams. The first part models a leader election which ensures that a cluster of N committer nodes are tolerant to up to $N - 1$ node failures. The election uses the `rlog` protocol of table 2 as an "atomic writer" to elect a temporary leader. The second part abstracts each cluster of committers to a single fail-safe "abstract" committer, and then performs a distributed version of two-phase commit. The two parts work hand-in-hand: if the "abstract" committers determine that a transaction can be committed, the commit is implemented as a write to a replicated log.

We use compositional reasoning (at the meta level) to explore each part separately; the results for leader election and the distributed commit protocols can be seen in table 3. The observations for these benchmarks are similar to the ones for table 2: lazy message ordering significantly reduces the state space (e.g., for `leader(3,3)`, lazy ordering is required to finish within the time limit), and Must becomes faster as the model becomes stronger, as stronger models often imply fewer executions. (For mbox, Must sometimes times out, due to its naive transitive-closure calculation.)

More interestingly, however, we also observe that using monitors does not introduce a lot of overhead thanks to our filtering mechanism (see §5): `leader-mon` explores less than 20% more executions compared to its non-monitored counterpart, thereby indicating that monitors can reason about temporal properties efficiently, despite the additional messages they introduce.

7 Related Work

Domain Specific Languages. Domain-specific languages such as Promela, TLA, and others [Holzmann 1997; Jaber et al. 2021; Lamport 2002; Padon et al. 2016; Sergey et al. 2018] allow abstract modeling of protocols, backed by a verifier. However, there is a gap between the models and the (imperative) code that engineers write in their implementations. Languages such as Mace [C. E. Killian et al. 2007], P and P# [Deligiannis et al. 2015; Desai et al. 2013], or data-centric ones such as Dedalus or Bloom [Alvaro 2015], are closer to mainstream programming models, but do not attempt to combine abstract features with efficient model checking algorithms. Must provides the "sweet spot" of providing domain-specific abstractions within a mainstream implementation language, together with a partial order semantics that is exploited by an efficient model checker.

Partial Order Semantics. Research in partial order semantics has focused on defining semantics of various communication models [Di Giusto et al. 2023] and coarsening the notion of equivalence among executions. Early work focused on Mazurkiewicz traces [Godefroid 1996]; more recent work looks at coarser equivalences such as reads-from equivalence or identifies conditions under which better reduction can be achieved [Abdulla et al. 2019; Agarwal et al. 2021; Albert et al. 2017, 2018; Aronis et al. 2018; Chalupa et al. 2017; Chatterjee et al. 2019].

Model Checking and DPOR. Algorithmic verification of distributed systems at the level of executions is a classical topic in verification and model checking. There has been extensive research on the verification of concurrent programs by means of stateless model checking [Godefroid 1997] with dynamic partial order reduction (DPOR) [Flanagan and Godefroid 2005]. While there are significant tools that focused on distributed message passing (e.g., [Godefroid 1997; Gotovos et al. 2011; C. E. Killian et al. 2007]), most existing work exclusively focuses on shared-memory concurrency.

DPOR-based algorithms have been extended to various memory consistency models [Abdulla et al. 2015, 2016, 2018, 2023; Kokologiannakis et al. 2019; Kokologiannakis and Vafeiadis 2020; Norris and Demsky 2013; Yi and Huang 2018] in the shared-memory setting or to message-passing/actor models [Lauterburg et al. 2009; Maiya et al. 2016; Sen and Agha 2006; Tasharofi et al. 2012],

Finally, research in DPOR has focused on *optimality*: ensuring that a DPOR-based search considers exactly one execution from each equivalence class [Abdulla et al. 2019; Kokologiannakis et al. 2022, 2019; Kokologiannakis and Vafeiadis 2020; Rodríguez et al. 2015], even for general memory consistency models. A culmination of this line of work is [Kokologiannakis et al. 2022], an optimal DPOR algorithm that is agnostic to the memory model, and which has linear memory overhead.

To the best of our knowledge, Must is the first DPOR for message-passing concurrency that is parameterized in the concurrency model, optimal, and has polynomial memory overhead.

TruSt [Kokologiannakis et al. 2022] presents a memory-model parametric DPOR framework with polynomial memory consumption for the shared memory setting. While Must's algorithm seems similar to TruSt, the underlying conditions on the consistency model as well as proofs of correctness are different in many subtle ways. For example, TruSt assumes that consistent executions can always be extended in a maximal way: writes can always be added co-maximally (co stands for "coherence"), and reads can always read the co-latest write. This property, however, does not hold in the message-passing setting: in p2p, for example, receives cannot always be added, and when they can, they need to read the co-*earliest* unread send that satisfies the receive predicate. Thus, Must's proof of correctness requires a different approach.

Concuerror [Abdulla et al. 2014; Aronis et al. 2018; Gotovos et al. 2011] is an optimal DPOR for programs written in Erlang. Concuerror also lazily orders messages using the concept of *observers*, but it is not clear whether this results in optimality w.r.t. reads-from equivalence. Concuerror supports timeouts and predicated receives, but does not support coexisting, *parametric*, communication models. (The paper describes mbox but the implementation also supports p2p.) Unlike Must, Concuerror's DPOR may consume exponential memory, leading to poor scalability (see §6).

Concurrency Testing. Automated systematic exploration tools for distributed systems implementations [Deligiannis et al. 2021, 2023; Desai et al. 2018; Meng et al. 2023; Musuvathi et al. 2008; Ozkan et al. 2018; Winter et al. 2023] attempt to systematically explore the state space of unmodified code. We expect that Must's modeling abstractions can be used to reduce the state space, and that combinations of DPOR with other (deterministic or randomized) state-space traversal techniques will yield model checkers that can explore even larger instances of industrial-scale protocols.

8 Conclusion

We presented Must, a framework for designing and verifying distributed systems. Must extends a mainstream programming language (e.g., Rust) with a concurrency API that (a) encapsulates common distributed systems abstractions, and (b) supports multiple coexisting communication models. This API is formalized by a partial-order semantics, and backed up by an optimal DPOR algorithm that incorporates a number of state-space-reduction techniques (e.g., lazy message ordering, selective receives). We demonstrated Must's effectiveness by verifying various configurations of challenging distributed protocols in an industrial setting.

References

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. "Stateless model checking for TSO and PSO." In: *TACAS 2015* (LNCS). Vol. 9035. Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. "Optimal dynamic partial order reduction." In: *POPL 2014*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. Oct. 10, 2019. "Optimal stateless model checking for reads-from equivalence under sequential consistency." *Proc. ACM Program. Lang.*, 3, (Oct. 10, 2019), 150:1–150:29, OOPSLA, (Oct. 10, 2019). https://doi.org/10.1145/3360576.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. "Stateless model checking for POWER." In: *CAV 2016* (LNCS). Vol. 9780. Springer, Berlin, Heidelberg, 134–156. https://doi.org/10.1007/978-3-319-41540-6_8.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Oct. 2018. "Optimal stateless model checking under the release-acquire semantics." *Proc. ACM Program. Lang.*, 2, OOPSLA, (Oct. 2018), 135:1–135:29. https://doi.org/10.1145/3276505.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, S. Krishna, Ashutosh Gupta, and Omkar Tuppe. 2023. "Optimal Stateless Model Checking for Causal Consistency." In: *TACAS 2023* (LNCS). Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13993. Springer, 105–125. https://doi.org/10.1007/978-3-031-30823-9_6.

Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. July 2021. "Stateless Model Checking Under a Reads-Value-From Equivalence." In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Springer International Publishing, Cham, (July 2021), 341–366. isbn: 978-3-030-81685-8. https://doi.org/10.1007/978-3-030-81685-8_16.

Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. 2017. "Context-sensitive dynamic partial order reduction." In: *CAV 2017*. Ed. by Rupak Majumdar and Viktor Kunčak. Springer International Publishing, Cham, 526–543. isbn: 978-3-319-63387-9. https://doi.org/10.1007/978-3-319-63387-9_26.

Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. 2018. "Constrained dynamic partial order reduction." In: *CAV 2018*. Ed. by Hana Chockler and Georg Weissenbacher. Springer International Publishing, Cham, 392–410. isbn: 978-3-319-96142-2. https://doi.org/10.1007/978-3-319-96142-2_24.

Jade Alglave, Luc Maranget, and Michael Tautschnig. July 2014. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." *ACM Trans. Program. Lang. Syst.*, 36, 2, (July 2014), 7:1–7:74. https://doi.org/10.1145/2627752.

Peter Alvaro. 2015. "Data-centric Programming for Distributed Systems." Ph.D. Dissertation. University of California, Santa Cruz, USA. http://www.escholarship.org/uc/item/2296w4q3.

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. "Optimal dynamic partial order reduction with observers." In: *TACAS 2018* (LNCS). Vol. 10806. Springer, 229–248. https://doi.org/10.1007/978-3-319-89963-3_14.

Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. June 2023. "Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels." *Proc. ACM Program. Lang.*, 7, PLDI, (June 2023). https://doi.org/10.1145/3591243.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Dec. 2017. "Data-centric dynamic partial order reduction." *Proc. ACM Program. Lang.*, 2, POPL, (Dec. 2017), 31:1–31:30. https://doi.org/10.1145/3158119.

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

Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. "Asynchronous programming, analysis and testing with state machines." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* Ed. by David Grove and Stephen M. Blackburn. ACM, 154–164. https://doi.org/10.1145/2737924.2737996.

Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. 2021. "Building Reliable Cloud Services Using Coyote Actors." In: *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021.* Ed. by Carlo Curino, Georgia Koutrika, and Ravi Netravali. ACM, 108–121. https://doi.org/10.1145/3472883.3486983.

Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. 2023. "Industrial-Strength Controlled Concurrency Testing for sc C tt # Programs with sc Coyote." In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Springer, 433–452. https://doi.org/10.1007/978-3-031-30820-8_26.

Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. "P: safe asynchronous event-driven programming." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 321–332. https://doi.org/10.1145/2491956.2462184.

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. "Compositional programming and testing of dynamic distributed systems." *Proc. ACM Program. Lang.*, 2, OOPSLA, 159:1–159:30. https://doi.org/10.1145/3276529.

Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. 2023. "A Partial Order View of Message-Passing Communication Models." *Proc. ACM Program. Lang.*, 7, POPL. https://doi.org/10.1145/3571248.

Erlang. 2023. *Concurrent Programming.* (). Retrieved July 12, 2023 from https://www.erlang.org/doc/getting_started/conc_prog.html.

Cormac Flanagan and Patrice Godefroid. 2005. "Dynamic partial-order reduction for model checking software." In: *POPL 2005.* ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315.

Patrice Godefroid. 1997. "Model checking for programming languages using VeriSoft." In: *POPL 1997.* ACM, Paris, France, 174–186. https://doi.org/10.1145/263699.263717.

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science. Vol. 1032. Springer. ISBN: 3-540-60761-7. https://doi.org/10.1007/3-540-60761-7.

Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. 2011. "Test-Driven Development of Concurrent Programs Using Concuerror." In: *Erlang 2011.* ACM, Tokyo, Japan, 51–61. ISBN: 9781450308595. https://doi.org/10.1145/2034654.2034664.

G.J. Holzmann. 1997. "The model checker SPIN." *IEEE Trans. Software Eng.*, 23, 5, 279–295. https://doi.org/10.1109/32.588521.

Nouraldin Jaber, Christopher Wagner, Swen Jacobs, Milind Kulkarni, and Roopsha Samanta. 2021. "QuickSilver: modeling and parameterized verification for distributed agreement-based systems." *Proc. ACM Program. Lang.*, 5, OOPSLA, 1–31. https://doi.org/10.1145/3485534.

Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Apr. 2007. "Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code." In: *NSDI 2007.* USENIX Association, Cambridge, MA, (Apr. 2007). https://www.usenix.org/conference/nsdi-07/life-death-and-critical-transition-finding-liveness-bugs-systems-code.

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. 2007. "Mace: Language Support for Building Distributed Systems." In: *PLDI 2007.* ACM, San Diego, California, USA, 179–188. ISBN: 9781595936332. https://doi.org/10.1145/1250734.1250755.

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Jan. 2022. "Truly stateless, optimal dynamic partial order reduction." *Proc. ACM Program. Lang.*, 6, POPL, (Jan. 2022). https://doi.org/10.1145/3498711.

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. 2023. "Unblocking Dynamic Partial Order Reduction." In: *CAV 2023*.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. "Model checking for weakly consistent libraries." In: *PLDI 2019*. ACM, New York, NY, USA. https://doi.org/10.1145/3314221.3314609.

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. "HMC: Model checking for hardware memory models." In: *ASPLOS 2020* (ASPLOS '20). ACM, Lausanne, Switzerland, 1157–1171. ISBN: 9781450371025. https://doi.org/10.1145/3373376.3378480.

Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. ISBN: 0-3211-4306-X. http://research.microsoft.com/users/lamport/tla/book.html.

Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. 2009. "A Framework for State-Space Exploration of Java-Based Actor Programs." In: *ASE 2009*, 468–479. https://doi.org/10.1109/ASE.2009.88.

Loom. (). Retrieved July 6, 2023 from https://github.com/tokio-rs/loom.

Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. 2016. "Partial Order Reduction for Event-Driven Multi-threaded Programs." In: *TACAS 2016*. Springer Berlin Heidelberg, Berlin, Heidelberg, 680–697. ISBN: 978-3-662-49674-9. https://doi.org/10.1007/978-3-662-49674-9_44.

Ruijie Meng, George Pirlea, Abhik Roychoudhury, and Ilya Sergey. 2023. "Distributed System Fuzzing." *CoRR*, abs/2305.02601. arXiv: 2305.02601. https://doi.org/10.48550/ARXIV.2305.02601.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard 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 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 267–280. http://www.usenix.org/events/osdi08/tech/full%5C_papers/musuvathi/musuvathi.pdf.

Brian Norris and Brian Demsky. 2013. "CDSChecker: Checking concurrent data structures written with C/C++ atomics." In: *OOPSLA 2013*. ACM, 131–150. https://doi.org/10.1145/2509136.2509514.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. "Randomized testing of distributed systems with probabilistic guarantees." *Proc. ACM Program. Lang.*, 2, OOPSLA, 160:1–160:28. https://doi.org/10.1145/3276530.

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. "Ivy: safety verification by interactive generalization." In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 614–630. https://doi.org/10.1145/2908080.2908118.

Robbert van Renesse and Fred B. Schneider. 2004. "Chain Replication for Supporting High Throughput and Availability." In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 91–104. http://www.usenix.org/events/osdi04/tech/renesse.html.

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. "Unfolding-based Partial Order Reduction." In: *CONCUR 2015* (LIPIcs). Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456.

Koushik Sen and Gul Agha. 2006. "Automated Systematic Testing of Open Distributed Programs." In: *FASE 2006*. Ed. by Luciano Baresi and Reiko Heckel. Springer Berlin Heidelberg, Berlin, Heidelberg, 339–356. ISBN: 978-3-540-33094-3. https://doi.org/10.1007/11693017_25.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. "Programming and proving with distributed protocols." *Proc. ACM Program. Lang.*, 2, POPL, 28:1–28:30. https://doi.org/10.1145/3158116.

Shuttle. (). Retrieved July 6, 2023 from https://github.com/awslabs/shuttle.

Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. "Chord: a scalable peer-to-peer lookup protocol for internet applications." *IEEE/ACM Trans. Netw.*, 11, 1, 17–32. https://doi.org/10.1109/TNET.2002.808407.

Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. "TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs." In: *FORTE 2012*. Ed. by Holger Giese and Grigore Rosu. Springer Berlin Heidelberg, Berlin, Heidelberg, 219–234. https://doi.org/10.1007/978-3-642-30793-5_14.

Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. "Randomized Testing of Byzantine Fault Tolerant Algorithms." *Proc. ACM Program. Lang.*, 7, OOPSLA1, 757–788. https://doi.org/10.1145/3586053.

Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. Jan. 2024. "Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions." *Proc. ACM Program. Lang.*, 8, POPL, (Jan. 2024). https://doi.org/10.1145/3632877.

Qiuping Yi and Jeff Huang. 2018. "Concurrency verification with maximal path causality." In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,*

ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 366–376. https://doi.org/10.1145/3236024.3236048.