Optimal Concolic Dynamic Partial Order Reduction

Mohammad Hossein Khoshechin Jorshari
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Michalis Kokologiannakis
ETH Zurich, Switzerland

Rupak Majumdar
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Srinidhi Nagendra
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

Stateless model checking (SMC) software implementations requires exploring both concurrency- and data nondeterminism. Unfortunately, most SMC algorithms focus on efficient exploration of concurrency nondeterminism, thereby neglecting an important source of bugs.

We present ConDpor, an SMC algorithm for unmodified Java programs that combines optimal dynamic partial order reduction (DPOR) for concurrency nondeterminism, with concolic execution for data nondeterminism. ConDpor is sound, complete, optimal, and parametric w.r.t. the memory consistency model. Our experiments confirm that ConDpor is exponentially faster than DPOR with small-domain enumeration. Overall, ConDpor opens the door for efficient exploration of concurrent programs with data nondeterminism.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Verification by model checking

Keywords and phrases Stateless model checking, dynamic symbolic execution

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.26

Related Version *Extended Version*: `https://doi.org/10.5281/zenodo.15649583` [44]

Supplementary Material *Software (Source Code)*: `https://github.com/mpi-sws-rse/jmc/tree/old_main`; archived at `swh:1:dir:b3b4445b195ebf730735a84dba1a442816e296a3`

Funding This research was funded in part by the DFG project 389792660 TRR 248–CPEC.

Acknowledgements We thank the anonymous reviewers for their detailed and constructive feedback, and Iason Marmanis for helpful feedback and suggestions.

1 Introduction

Systematic state space exploration for concurrent software involves exploring both *concurrency nondeterminism* and *data nondeterminism*. Concurrency nondeterminism arises out of the possible orders in which threads or processes can be scheduled or messages delivered; data nondeterminism arises out of the possible values that variables may take. Both sources are important in finding bugs.

There are orthogonal techniques to handle each source in the context of dynamic (or stateless) model checking. *Dynamic partial order reduction* (DPOR) [29, 30, 6, 48, 45] handles concurrency nondeterminism efficiently by maintaining scheduling constraints along an execution, and exploring distinct schedules if and only if the corresponding constraints lead to non-equivalent executions. Modern DPOR tools explore the concurrency space optimally [6, 45] and also account for errors caused by compiler/CPU reorderings [8, 5, 48, 56].

Symbolic execution handles data nondeterminism by maintaining symbolic constraints on data along an execution, and exploring distinct execution paths if and only if the symbolic constraints along these paths are mutually disjoint. Tools based on *dynamic* symbolic execution (also called concolic testing, CT) [31, 61, 19, 23], maintain the symbolic constraints dynamically, and efficiently manage the backtracking symbolic search.

Despite their many individual successes, few tools handle both concurrency and data nondeterminism *simultaneously and optimally.* Current approaches ignore either partial orderings [59, 19, 23] or data nondeterminism [55, 6, 56, 40, 45] or optimality [58, 60].

In this paper, we present ConDpor, a systematic exploration algorithm that combines DPOR with CT. ConDpor does so by maintaining symbolic constraints in the trace constructed by a DPOR algorithm (specifically, TruSt [45]), and backtracking both on scheduling constraints and on symbolic constraints.

But why tackle both issues at the same time in the first place? At a first glance, data- and concurrency bugs seem orthogonal, as many concurrent programs are otherwise deterministic. However, popular concurrent data structures argue otherwise: algorithms like timestamped stack [26] or elimination-backoff stack [37] use timestamps or random identifiers to achieve better efficiency, and thus fundamentally require reasoning about both concurrency- and data nondeterminism. Moreover, concurrency APIs also often require supporting data nondeterminism, as their primitives may fail with multiple error values, each one leading to a different control-flow path. In a nutshell, handling concurrency and data together amplifies stateless model checking to unbounded state spaces.

An interesting aspect of ConDpor lies in how symbolic constraints are generated and manipulated in the context of DPOR. Indeed, as we later show, ConDpor only needs to know when symbolic values are generated, and when symbolic constraints are evaluated. As such, instead of using a dedicated runtime to manipulate the symbolic path constraint according to the executed instructions, ConDpor offloads the manipulation of symbolic expressions to the underlying programming language (similarly to [20, 63, 60]). Specifically, ConDpor provides dedicated datatypes for symbolic types (available to users) that "carry" the symbolic path constraint of the execution. Arithmetic operations on symbolic variables manipulate the path constraint before performing the respective operation, while logical operations record an event in the current trace at which ConDpor can later backtrack.

ConDpor is sound (i.e., explores no false positives), complete (i.e., explores all program behaviors), optimal (i.e., does not explore traces that only differ in the execution of DPOR-independent instructions or in the model satisfying the symbolic constraints), and parametric in the choice of the memory model (i.e., can find bugs due to compiler/CPU reorderings). As our implementation of ConDpor for concurrent Java programs demonstrates, ConDpor is also efficient: its overhead over standard DPOR is proportional to the amount of data nondeterminism in a given program, and ConDpor is able to fully explore challenging concurrent data structures within seconds.

In summary, we make the following contributions.

- We present ConDpor, a combination of optimal DPOR and concolic execution, and prove it sound, complete, and optimal.
- We implement ConDpor in a tool for programs running on the JVM.
- We empirically show that ConDpor can systematically explore concurrent programs involving both data and concurrency nondeterminism efficiently, with only a small memory overhead.

2 Overview

In this section, we provide a motivating example, followed by an informal description of ConDpor, along with some algorithmic and engineering challenges we faced.

2.1 A Motivating Example

We motivate ConDpor using *timestamped stack* [26], a high-performance data structure where each thread maintains a portion of the stack. When a thread pushes an element, it adds it to a thread-local stack and marks it with a timestamp (e.g., by calling a hardware timer). When a thread pops an element, it goes over the top elements of all stacks, and returns the one with the largest timestamp.

Suppose we want to verify a client that makes n concurrent pushes to the stack and then pops an element:

$$\big(\texttt{push}(1) \;\big\|\; \texttt{push}(2) \;\big\|\; \ldots \;\big\|\; \texttt{push}(n)\big)\,;\texttt{pop}() \tag{NPUSH+POP}$$

Enumerative approaches like DPOR would use a shared atomic counter to model timestamps: each call that gets a timestamp will atomically fetch-and-increment the counter. These shared access to the counter make pure DPOR explore $n!$ orderings among the pushes.

Instead, a symbolic approach like CT would construct a symbolic counter value for each thread, only constraining that symbolic values in the same thread are linearly ordered. As the pop() operation compares timestamps to find the largest one, a pure concolic approach would first serialize the pushes and then solve the symbolic constraints to find which push had the largest timestamp, leading to $n! \cdot 2^{n-1}$ executions.

ConDpor combines both DPOR and CT: while it does use a symbolic counter for each thread, ConDpor also leverages DPOR and does not order the pushes, thereby only exploring 2^{n-1} executions (corresponding to solving the symbolic constraints). In fact, if each thread pushed k times the difference becomes even more prominent: ConDpor would still consider 2^{n-1} executions, whereas DPOR would explore $(kn)!/(k!)^n$.

While this example shows a “best case” – we have assumed no further contention – our experiments confirm that ConDpor does outperform purely enumerative approaches for many benchmarks (see §6).

2.2 Making Optimal DPOR Concolic

Let us now describe ConDpor. The core of ConDpor is a backtracking search over program events that explores all executions of a given program (up to equivalence). ConDpor explores exactly one execution from each equivalence class, and initiates no redundant explorations.

2.2.1 Equivalence Partitioning

Our notion of equivalence is a symbolic version of DPOR's underlying equivalence partitioning.[1] For example, assuming an underlying *reads-from* equivalence partitioning [22], ConDpor considers two interleavings equivalent if the reads obtain their values from the same writes, and every symbolic value satisfies the same constraints. As an example, consider

[1] ConDpor extends TruSt [45], which supports both Mazurkiewicz [54] and reads-from equivalence [22].

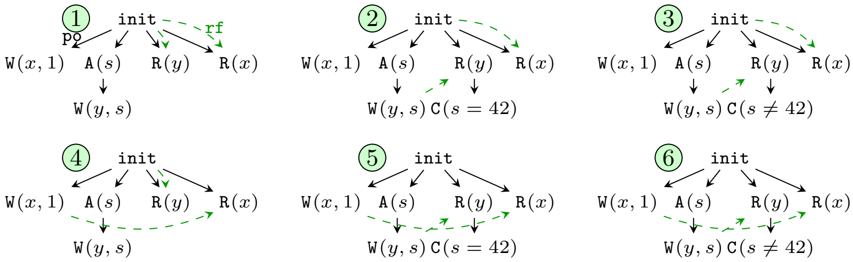

Figure 1 The six execution graphs of W+SW+R+R model its equivalence classes.

the program below[2] where nondet() is a source of symbolic values (e.g., program input):

$$x := 1 \parallel y := \texttt{nondet}() \parallel \texttt{assert}(y = 42) \parallel b := x \qquad (\text{W+SW+R+R})$$

Naively, this program has 36 behaviors. There are $4! = 24$ ways the accesses to x and y can be interleaved. For half of those, the read of y reads 0, whereas for the other half it reads the written symbolic value, which is evaluated in the assertion and may or may not be 42.

Observe, however, that the ordering between the writes to x and y does not matter, as these are different memory locations. As such, it suffices to only consider 6 equivalence classes: two cases for the read of x, and for each of them, one case where the read of y reads 0, and two cases where the read of y reads the written symbolic value (which may or may not be 42).

Formally, the equivalence classes of a given program are represented as a set of *execution graphs* [45]. In each of these graphs, the nodes (events) correspond to instructions of the program, while the edges denote various relations between the instructions. Examples of events are reads and writes to shared variables, while examples of relations include the *program order*, po, which orders the instructions of a given thread, and *reads-from*, rf, which connects a read to the write from which it obtains its value.

For ConDpor, we extend events to also include the generation of symbolic values and the evaluation of symbolic constraints. For instance, the execution graphs for W+SW+R+R can be seen in Fig. 1. In these graphs, the initializer event init models the initialization writes of all memory location (and is po-ordered before the first event of each thread), while R, W, A and C denote reads, writes, symbolic-value generation and constraint evaluation, respectively. Using these new events means that we can now check for satisfiability of all constraints simply by taking the conjunction of all constraint-evaluation events.

Execution graphs have to satisfy certain consistency constraints imposed by the underlying memory model. For example, under sequential consistency [51], given a program where thread I writes two variables ($x := 1; y := 1$) and thread II reads them in the opposite order ($a := y; b := x$), thread II cannot read $a = 1 \land b = 0$. Other models (known as weak memory models) allow this outcome, which may arise due to compiler and/or CPU optimizations.

Although below we assume sequential consistency to ease the presentation, ConDpor is parametric in the choice of the memory consistency model (see §3).

[2] We use x, y, z for shared variables (initialized to 0), and $a, b, \ldots$ for thread-local variables.

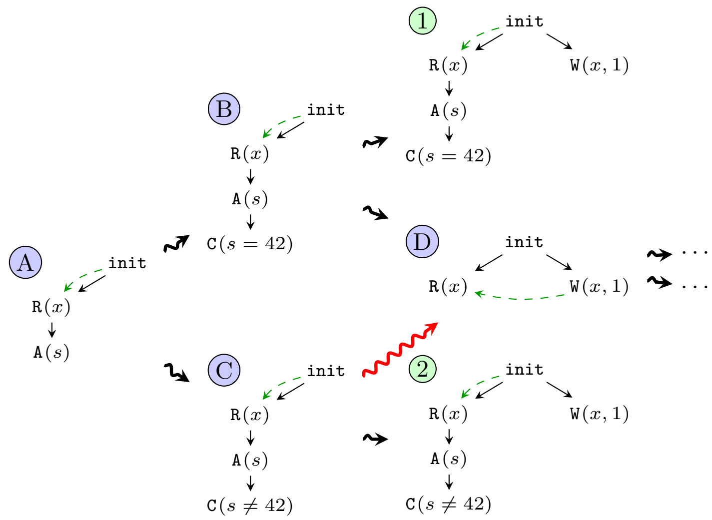

Figure 2 A CONDPOR exploration.

2.2.2 Enumerating Execution Graphs

Given the representation above, we can verify a concurrent program by enumerating all of its execution graphs. CONDPOR does so in a depth-first manner: starting from the empty graph, it extends it one event at a time (maintaining consistency), recording alternative exploration options when possible. When a read is added to the graph, CONDPOR explores all writes the read can (consistently) read from, and sets its `rf` accordingly. When a write is added, CONDPOR checks to see whether any of the previously added reads can be (consistently) revisited to read from the newly added write. Finally, when a constraint-evaluation event is added, CONDPOR examines all both outcomes, assuming both are satisfiable.

▶ **Example 1.** Let us demonstrate how CONDPOR works with an example.

$$\begin{array}{l} a := x \\ b := \texttt{nondet}() \\ \texttt{assert}(a = 0 \vee b \neq 42) \end{array} \;\Bigg\|\; x := 1 \qquad (\textsc{rs+w})$$

In the program above, an error manifests if thread I both reads 1 for x, and `nondet()` returns the unexpected value 42. CONDPOR's exploration for RS+W is depicted in Fig. 2.

At the first step, CONDPOR adds $\mathtt{R}(x)$ to the graph; this read can only read 0, as there is only one write to x in the current graph (the `init` event). When CONDPOR encounters `nondet()`, it adds an $\mathtt{A}(s)$ label to the graph. This label is used to keep track of the domain of the symbolic value (see §3), and also to inform the SMT solver of the new variable.

When CONDPOR reaches the assert statement, while the value of a is already known, the result of $b \neq 42$ is not. As such, CONDPOR examines both options: in graph B, we assume that $b = 42$ (since the SMT solver returns a satisfying assignment for $s = 42$), while in graph C we assume that $b \neq 42$ (again, the SMT solver is queried[3] for the satisfiability of $s \neq 42$).

[3] As in concolic execution, one of these two calls can be spared; see §4.

In graphs B and C, ConDpor subsequently encounters the statement $x := 1$. Merely adding the respective write event to the graph, however, is inadequate, as ConDpor also has to explore the case where thread I reads from this write. As such, ConDpor also examines whether $\mathtt{W}(x, 1)$ can revisit any of the existing same-location reads in the graph.

Revisiting complicates the exploration, as ConDpor has to *restrict* graphs resulting from revisits in order to ensure consistency. Indeed, in our example above ConDpor removes all $(\mathtt{po} \cup \mathtt{rf})^{+}$-successors of $\mathtt{R}(x)$ in the resulting graph D, as the existence of these events might be (control-flow) dependent on the read value.

Also observe that we need to maintain *optimality*: each execution graph should be explored exactly once. In the exploration above, if we are not careful and revisit $\mathtt{R}(x)$ from both B and C, we will get graph D twice, leading to duplication. ConDpor ensures $\mathtt{R}(x)$ is only revisited in exploration B.

Optimality

ConDpor ensures optimality by extending the notion of maximal extensions [45] from existing graph-based DPOR algorithms so that it accounts for constraint evaluation events. The key idea behind maximal extensions is roughly that if performing the same revisit in two graphs G_1 and G_2 leads to the same graph, then G_1 and G_2 only differ in the events deleted by the revisit. In the example above, graphs B and C differ in their constraint evaluation event. In turn, by imposing a criterion on the events deleted by the revisit – that they must be maximally added – we can perform this in only one of the possible sub-explorations.

We defer the presentation of our maximal-extension definition to §4, where we also show how they can be efficiently computed.

2.3 Engineering Challenges

ConDpor works for any programming language, assuming that concurrency primitives (e.g., shared-memory reads and writes) can be intercepted, and that symbolic expressions can be manipulated along program execution (e.g., with a symbolic interpreter).

2.3.1 Intercepting Concurrency Primitives in the JVM

DPOR implementations intercept concurrency primitives through system call interception [55], through a custom runtime [48, 35], or through mocking concurrency libraries [64, 27].

Unfortunately, neither technique is appropriate for the JVM: distinguishing system calls from the program under test from system calls made by the JVM itself is difficult, writing and maintaining a custom runtime for Java is a multi-person-year effort, and mocking is inapplicable because some concurrency primitives are handled natively within the JVM.

Our solution is to instrument the program through bytecode rewriting using the ASM library [1], and to provide our own asynchronous runtime for concurrency operations. Our instrumentation provides additional synchronization "under the hood" yielding the scheduler full control over Java's concurrency constructs.

2.3.2 Symbolic Execution without an Interpreter

As with concurrency primitives, manipulating symbolic expressions typically requires a custom interpreter for the instructions (e.g., [19], [15]). Sometimes, the interpreter instructions are instrumented into the code so that the symbolic interpretation occurs as the program runs.

While we do perform bytecode rewriting, the cost of developing a full symbolic interpreter for the JVM is quite high, and the interpreter has to understand and differentiate between the code under test and JVM's own operations.

To overcome this issue, we use Java's type system to push the symbolic interpretation into the language itself. More specifically, in addition to all constructs made available to the user by the language, ConDpor provides an API defining symbolic types (e.g., symbolic integers). These types provide the same operations as their non-symbolic counterparts, but also "carry" a symbolic expression that is being manipulated along with the respective values.

To manipulate symbolic values, the API provides the operations `nondet`() and `evaluate`(), which create a symbolic value and evaluate a symbolic constraint, respectively. Note that the purpose of evaluation is twofold: apart from allowing the result of a symbolic expression to be used in non-symbolic contexts (e.g., in the condition of an if-statement), it also serves as a backtracking point for ConDpor. Indeed, whenever `evaluate`() is encountered, the runtime calls ConDpor-specific code to add an $\mathtt{C}$ event to the graph (see § 2.2). While having such an API means that users must use it to capture all sources of data nondeterminism, such a constraint can be easily lifted (e.g., by means of a data-flow analysis over bytecode).

Crucially, our symbolic API remains a part of the program under test: it is implemented fully in Java and requires no special runtime. Internally, our API calls ConDpor-specific code only to register the generation/evaluation of a symbolic expression.

3 Partial Order Semantics

We now formally describe execution graphs, the data structure behind ConDpor's partial order semantics. We begin by defining events.

▶ **Definition 2.** *An event, $e \in \mathsf{Event}$, is either the initialization event* `init` *or a thread event $\langle t, i, lab\rangle$, where $t \in \mathsf{Tid}$ is a thread identifier, $i \in \mathsf{Idx} \triangleq \mathbb{N}$ an index within a thread, and $lab \in \mathsf{Lab}$ a label that takes one of the following forms:*

- *Write label, $\mathtt{W}(l, v)$, where $l \in \mathsf{Loc}$ is the location accessed and $v \in \mathsf{Val}$ is the value written.*
- *Read label, $\mathtt{R}(l)$, where $l \in \mathsf{Loc}$ is the location accessed.*
- *Symbolic generation label, $\mathtt{A}(s)$, where $s \in \mathsf{Sexpr}$ is the symbolic value generated.*
- *Constraint evaluation label, $\mathtt{C}(\phi)$, where $\phi \in \mathsf{Sexpr}$ is a boolean symbolic expression.*
- *Block label,* `B`*, denoting thread blockage (see below).*
- *Error label,* `error`*, denoting a program error (see below).*

We define the set of all read events as $\mathtt{R} \triangleq \{\langle t, i, lab\rangle \mid lab = \mathtt{R}(\_)\}$. Similarly, we denote the set of all write, symbolic-generation, and constraint-evaluation events as $\mathtt{W}$, $\mathtt{A}$, and $\mathtt{C}$, respectively, and assume that `init` $\in \mathtt{W}$.

Block and error labels are generated by `assume` and `assert` statements, respectively. An `assume`(b) statement is modeled as **if** $(!b)$ `block()`, while an `assert`(b) statement is modeled as **if** $(!b)$ `error()`. The `block()` and `error()` statements generate the respective labels, while $b \in \mathsf{Val}$ is a boolean value.

Having defined events, we define execution graphs as follows.

▶ **Definition 3.** *An execution graph G consists of:*

(1) *a set of events E that includes* `init` *and does not contain multiple events with the same thread identifier and serial number;*

(2) *the* reads-from *function $\mathtt{rf} : E \cap \mathtt{R} \to E \cap \mathtt{W}$ that maps each read to the same-location write event from which it gets its value;*

(3) *the* coherence order $\mathtt{co} \subseteq \bigcup_{l \in \mathsf{Loc}} \mathtt{W}_l \times \mathtt{W}_l$ *(with* $\mathtt{W}_l \triangleq \{\langle t, i, lab\rangle \in \mathtt{W} \mid lab = \mathtt{W}(l, \_)\}$*), a strict partial order that is total on* $\mathtt{W}_l$ *for every location* $l \in \mathsf{Loc}$*; and*

(4) *a total order* $\leq$ *on* E*, representing the order in which events were added to the graph.*

We write $G.\mathtt{E}$, $G.\mathtt{rf}$, $G.\mathtt{co}$, and $\leq_G$ to project the various components of an execution graph, and use $G|_E$ to denote the restriction of an execution graph G to a set of events E. We assume that $\mathtt{init} \in \mathtt{W}$, and use the functions $\mathtt{tid}$, $\mathtt{idx}$, $\mathtt{loc}$, $\mathtt{val}$ and $\mathtt{formula}$ to get (when applicable) the thread identifier, index, location, value and formula of an event, respectively. We write $G.\Phi \triangleq \bigwedge_{s \in G.\mathtt{C}} \mathtt{formula}(s)$ for the conjunction of all symbolic constraints of the graph, $G.\mathtt{W}$ for $G.\mathtt{E} \cap \mathtt{W}$ (and similarly for other sets), and use subscripts to further restrict these sets (e.g., $\mathtt{W}_l \triangleq \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$). Finally, given two events $e_1, e_2 \in G.\mathtt{E}$, we write $e_1 <_G e_2$ if $e_1 \leq_G e_2$ and $e_1 \neq e_2$.

As defined above, execution graphs do not have an explicit *program order* ($\mathtt{po}$) component. We thus induce $\mathtt{po}$ based on our event representation as follows:

$$\mathtt{po} \triangleq \{\langle \mathtt{init}, e\rangle \mid e \in \mathsf{Event} \setminus \{\mathtt{init}\}\} \cup \{\langle\langle t_1, i_1, lab_1\rangle, \langle t_2, i_2, lab_2\rangle\rangle \mid t_1 = t_2 \wedge i_1 < i_2\}$$

We write $G.\mathtt{porf} \triangleq (\mathtt{po} \cap (G.\mathtt{E} \times G.\mathtt{E}) \cup \{\langle G.\mathtt{rf}(r), r\rangle \mid r \in G.\mathtt{R}\})^+$ for the graph's causal dependency relation.

Finally, we define the semantics of a program P under a model M as the set of execution graphs corresponding to P that satisfy M's consistency predicate. In this paper, we assume a memory model M providing a consistency predicate $\mathsf{consistent}_{\mathsf{M}}(G)$, determining whether a graph G is consistent, as well as an error predicate $\textsc{IsErroneous}_{\mathsf{M}}(G)$, determining whether G contains an error (e.g., a data race) according to M. We further assume that $\mathsf{consistent}_{\mathsf{M}}(\cdot)$ is extensible, prefix-closed, and implies RMW atomicity and $\mathtt{porf}$ acyclicity (see [45]), and that $\textsc{IsErroneous}_{\mathsf{M}}(\cdot)$ is monotone (if a prefix-closed subset of a graph G contains an error, then so does G). Models satisfying these assumptions include SC [51], TSO [57] and RC11 [50] under shared-memory consistency, as well as asynchronous communication, peer-to-peer, and mailbox under message-passing consistency [25].

Given such an underlying memory consistency model M, we then define a new memory model M_S, with $\mathsf{consistent}_{\mathsf{M_S}}(G) \triangleq \mathsf{consistent}_{\mathsf{M}}(G) \wedge \mathtt{SAT}(G.\Phi)$ and $\textsc{IsErroneous}_{\mathsf{M}_S}(G) \triangleq \textsc{IsErroneous}_{\mathsf{M}}(G)$, where $\mathtt{SAT}(\cdot)$ is a predicate denoting whether a symbolic constraint is satisfiable. Intuitively, M_S extends the underlying communication semantics to also require that all symbolic constraints in a graph are satisfiable. It is easy to show that M_S satisfies the memory-model properties stated above.

4 Algorithm

Let us now present ConDpor in detail (cf. Algorithm 1). Although our algorithm works both under the Shasha-Snir and the reads-from equivalence partitioning, here we only provide the (more straightforward) Shasha-Snir version [62], the generalization of Mazurkiewicz equivalence partitioning for relaxed memory models such as TSO and PSO. Symbolic reasoning does not affect the choice of partitioning, and the required changes can be adapted from [45].

Given a program P, ConDpor explores all of its consistent execution graphs under a model M by calling $\textsc{Visit}_{P,\mathsf{M}_S}(G_\emptyset)$, where $G_\emptyset$ is the initial graph containing only the initialization event $\mathtt{init}$. In turn, Visit constructs the execution graphs of P one at a time and incrementally, while also recording the event addition order in the graph's $\leq$ component.

Let us now examine Visit more closely. As a first step, Visit checks whether G contains an error (line 2); errors include safety violations (which manifest with an $\mathtt{error}$ label), as well as memory-model-specific errors such as data races.

Algorithm 1 ConDpor: Optimal Concolic Dynamic Partial Order Reduction.

1: **procedure** $\textsc{Visit}_{P,\mathrm{M}_S}(G)$
2:     **if** $\textsc{IsErroneous}_{\mathrm{M}_S}(\mathrm{G})$ **then exit**(“Erroneous execution”)
3:     $a \leftarrow \textsc{Add}(G, \mathsf{next}_P(G))$
4:     **if** $a = \bot$ **then return** “Visited full execution graph G”
5:     **switch** a **do**
6:         **case** $a \in \mathtt{R}$
7:             **for** $w \in G.\mathtt{W}_{\mathtt{loc}(a)}$ **do** $\textsc{VisitIfConsistent}_{P,\mathrm{M}_S}(\mathsf{SetRF}(G, a, w))$
8:         **case** $a \in \mathtt{C}$
9:             $\textsc{VisitIfConsistent}_{P,\mathrm{M}_S}(\mathsf{SetSYM}(G, a, \mathtt{formula}(a)))$
10:             $\textsc{VisitIfConsistent}_{P,\mathrm{M}_S}(\mathsf{SetSYM}(G, a, \neg\mathtt{formula}(a)))$
11:         **case** $a \in \mathtt{W}$
12:             $\textsc{VisitCOs}_{P,\mathrm{M}_S}(G, a)$
13:             **for** $r \in G.\mathtt{R}_a$ **such that** $\langle r, a\rangle \notin G.\mathtt{porf}$ **do**
14:                 $\textsc{MaybeBackwardRevisit}_{P,\mathsf{M}_S}(G, r, a)$
15:         **case** _ $\textsc{Visit}_{P,\mathrm{M}_S}(G)$
16: **procedure** $\textsc{VisitCOs}_{P,\mathrm{M}_S}(G, a)$
17:     **for** $w_p \in G.\mathtt{W}_{\mathtt{loc}(a)}$ **do** $\textsc{VisitIfConsistent}_{P,\mathrm{M}_S}(\mathsf{SetCO}(G, w_p, a))$
18: **procedure** $\textsc{MaybeBackwardRevisit}_P(G, r, a)$
19:     $\mathit{Deleted} \leftarrow \{e \in G.\mathtt{E} \mid r <_G e \wedge \langle e, a\rangle \notin G.\mathtt{porf}\}$
20:     $[d_1, \dots, d_n] \leftarrow \mathsf{sort}_{<_G}(\mathit{Deleted})$
21:     **if** $\exists G', G''$ such that $G' \overset{r}{\rightsquigarrow} G'' \overset{d_1}{\rightsquigarrow} \cdots \overset{d_n}{\rightsquigarrow} G|_{G.\mathtt{E}\setminus\{a\}}$ **then**
22:         $\textsc{VisitIfConsistent}_{P,\mathsf{M}_S}(\mathsf{SetRF}(G|_{G.\mathtt{E}\setminus \mathit{Deleted}}, r, a))$
23: **procedure** $\textsc{VisitIfConsistent}_{P,\mathrm{M}_S}(G)$
24:     **if** $\mathsf{consistent}_{\mathsf{M}_\mathsf{S}}(G)$ **then** $\textsc{Visit}_{P,\mathrm{M}_S}(\mathrm{G})$

After ensuring that the graph is error-free, Visit extends the current graph with the next program event (line 3) with the help of Add and next. next returns an event from a thread that is not blocked or finished, and Add adds it to the graph in place, and then returns the new event. If there are no events to add (in which case Add/next returns $\bot$), a full execution of P has been explored, and Visit returns (line 4).

If a is a read, we recursively explore all consistent $\mathtt{rf}$ options for it (line 7). (We assume that $\mathsf{SetRF}(G, r, w)$ returns a new graph G' that only differs from G in $\mathtt{rf}$: $G'.\mathtt{rf}(r) = w$.)

If a is a constraint-evaluation event, ConDpor has to examine whether the newly added event (and its negation) renders $G.\Phi$ satisfiable. To do that, Visit calls VisitIfConsistent twice: once with a's constraint as is, and another time with a's expression negated. In order to set a's symbolic constraint, Visit employs the $\mathsf{SetSYM}(G, a, \phi)$ function, which returns a new graph G' that is identical to G, apart from the constraint of event a, which is set to ϕ. Observe that if $G'.\Phi$ is unsatisfiable, then the execution subsequently be dropped by VisitIfConsistent as inconsistent.

If a is a write, ConDpor examines both the non-revisit- and the revisit case. In the non-revisit case, ConDpor explores all consistent $\mathtt{co}$ options for a via VisitCOs (line 12). Analogously to SetRF, $\mathsf{SetCO}(G, w_p, w)$ returns a new graph G' that only differs from G in that the $\mathtt{co}$-predecessor of w is w_p.

In the revisit case, Visit examines previously-added reads as revisit candidates. Concretely, Visit only revisits same-location reads that are not $\mathtt{porf}$-before a (as revisiting those would create $\mathtt{porf}$ cycles), assuming that the events deleted from the revisit have been added maximally (lines 13-14). The maximal-addition definition closely follows the one of

Kokologiannakis et al. [46]: ConDpor backward-revisits r from a if all deleted events are added co-maximally (if non-symbolic), or in a unique fashion (if symbolic). Formally, we write $G_1 \overset{e}{\rightsquigarrow} G_2$ if $G_2 = \textsc{Add}(G_1, e)$ and:

$$
\begin{array}{llll}
G_2.\mathtt{rf} = G_1.\mathtt{rf} \cup \{\langle \mathsf{max}_{G_1.\mathtt{co}_e}, e\rangle\} & G_2.\mathtt{co} = G_1.\mathtt{co} & G_2.\Phi = G_1.\Phi & \text{if } e \in \mathtt{R} \\
G_2.\mathtt{rf} = G_1.\mathtt{rf} & G_2.\mathtt{co} = G_1.\mathtt{co} \cup (G_1.\mathtt{W}_e \times \{e\}) & G_2.\Phi = G_1.\Phi & \text{if } e \in \mathtt{W} \\
G_2.\mathtt{rf} = G_1.\mathtt{rf} & G_2.\mathtt{co} = G_1.\mathtt{co} & G_2.\Phi = \mathtt{ST}(G_1.\Phi, e) & \text{if } e \in \mathtt{C} \\
G_2.\mathtt{rf} = G_1.\mathtt{rf} & G_2.\mathtt{co} = G_1.\mathtt{co} & G_2.\Phi = G_1.\Phi & \text{otherwise}
\end{array}
$$

Above, the $\mathtt{ST}(\cdot)$ function acts as a tiebraker if both cases of a constraint-evaluation event are feasible. If both $\mathtt{formula}(e)$ and $\neg\mathtt{formula}(e)$ are feasible, it (arbitrarily) returns $G_1.\Phi \wedge \mathtt{formula}(e)$, while if only $\mathtt{formula}(e)$ (resp. $\neg\mathtt{formula}(e)$) is feasible, it returns $G_1.\Phi \wedge \mathtt{formula}(e)$ (resp. $G_1.\Phi \wedge \neg\mathtt{formula}(e)$).

4.1 Optimizing the Algorithm

Observe that Algorithm 1 performs a lot of redundant calls to the SMT solver. As one example, checking graph consistency requires checking the satisfiability of $G.\Phi$. However, as $G.\Phi$ only changes when adding evaluation labels, it suffices to only call the solver in the respective cases. As another example, observe that satisfiability is also necessary in the $\mathtt{ST}(\cdot)$ function, when checking for maximality of symbolic evaluation events. We can spare these solver calls by leveraging concolic execution, as explained below.

Indeed, observe that Algorithm 1 does not immediately leverage concolic execution: the SMT solver is called twice when adding a symbolic constraint (lines 9-10), as well as to evaluate the maximality of constraint-evaluation events during backward revisits ($\mathtt{ST}(\cdot)$ function). As done in concolic execution, however, if we make our symbolic types "carry" a model of each symbolic value along with the respective symbolic expression, one of these two calls can be spared. Concretely, when adding a constraint-evaluation event, we can evaluate the non-negated constraint (line 9) without resulting to an SMT solver (by using the model).

Backward revisits can also leverage concolic execution and spare SMT calls in the $\mathtt{ST}(\cdot)$ function, though with a bit more care. Assuming G_r is the graph resulting from the revisit, we first check G_r's consistency[4] and obtain a model of all symbolic values in G_r. Then, we iterate over the events in the deleted set (lines 19-20), and consider a constraint-evaluation event a as maximal, if $\mathtt{formula}(a)$ is true in the model obtained in G_r. If that's not the case, then this means that $\neg\mathtt{formula}(a)$ is feasible, and hence that case is deemed as maximal. (Any symbolic generation events encountered, simply extend the existing model.)

Note, however, that the above procedure requires the model that is returned by an SMT solver for a given formula to be deterministic (which is the case with e.g., Z3). To see why, consider the following program, where a symbolic value is shared between two threads:

$$
\begin{array}{c}
a := \mathtt{nondet}() \\
b := x \;\Big\|\; \mathtt{assume}(a > 42) \;\Big\|\; \mathbf{if}\ (a > 0)\ \ x := 1
\end{array}
$$

There are two executions in which $x := 1$ can revisit $b := x$: one where thread II has a $\mathtt{C}(a > 42)$ event (and the assume succeeds) and one where it has a $\mathtt{C}(a \leq 42)$ event (and the assume blocks). In both cases, the graph occurring if $\mathtt{W}(x, 1)$ revisits $\mathtt{R}(x)$ will contain $\mathtt{R}(x)$, $\mathtt{C}(a > 0)$ and $\mathtt{W}(x, 1)$, while the deleted set will be $\mathtt{C}(a > 42)$ and $\mathtt{C}(a \leq 42)$ respectively. If,

[4] VisitIfConsistent in line 15 can reuse this result.

however, the model we obtain from the solver for $\mathtt{C}(a > 0)$ is $a = 43$ in the first case and $a = 1$ in the second case, then the revisit will not occur, as each case would drop it, assuming it would take place in the other one.

4.2 Soundness, Completeness and Optimality

Given a program P and a consistency model M that satisfies the conditions of §3, Algorithm 1 is sound (i.e., only explores consistent full executions), complete (i.e., explores all of P's consistent executions), and optimal (i.e., explores each consistent execution exactly once, and does not engage in any wasteful explorations). The proofs follow by extending the framework of [46] to account for symbolic event types.

Before formally stating our results, let us provide some definitions.

- We write $[\![P]\!]_{\mathsf{M}_S}$ for the set of consistent full graphs corresponding to the program P under M_S, and $G_1 \approx G_2$ if G_1 and G_2 agree on all their components but $\leq_G$.
- Given graphs G_1, G_2, we say that G_1 is a *prefix* of G_2 (written $G_1 \sqsubseteq G_2$), if there exists a graph $G'_1 \approx G_1$ such that the algorithm can reach a graph $G'_2 \approx G_2$ from G'_1 in a series of non-revisit steps.

Let us now state our results. Observe that ConDpor is trivially sound, as Algorithm 1 never visits inconsistent executions.

▶ **Theorem 4** (Completeness). *Let G_f be an execution graph in $[\![P]\!]_{\mathsf{M}}$. Then $\textsc{Visit}_{P,\mathsf{M}_S}(G_\emptyset)$ calls $\textsc{Visit}_{P,\mathsf{M}_S}(G'_f)$ for some $G'_f \approx G_f$.*

▶ **Theorem 5** (Optimality). *$\textsc{Visit}_{P,\mathsf{M}_S}(G_\emptyset)$ never calls $\textsc{Visit}_{P,\mathsf{M}_S}$*

(a) *for graphs G_1, G_2 such that $G_1 \approx G_2$, or*

(b) *for a graph G that cannot lead to a full execution $G_f \in [\![P]\!]_{\mathsf{M}_S}$.*

We prove both theorems by showing the exact (unique) sequence of steps that the algorithm takes to reach a graph G_f, with $G_f \approx G'_f \in [\![P]\!]_{\mathsf{M}_S}$.

Given $G_s \sqsubseteq G_t$, we define a procedure $\textsc{GetNext}(G_s, G_t)$ that returns the (minimal, unique and nonempty) sequence of algorithmic steps S with which G_s reaches a graph G such that $G_s \sqsubseteq G \sqsubseteq G_t$, and with the property (proved by induction) that S does not revisit events in G_s (it may revisit events not in G_s). Crucially, the inductive proof for $\textsc{GetNext}(G_s, G_t)$ relies on the ability to extend its first argument G_s, which is not affected by our new symbolic events (M_S is maximally extensible).

Calling GetNext in a fixpoint yields the desired result.

5 Implementation

We implemented ConDpor as a tool for concurrent Java programs[5]. Our tool supports commonly used Java synchronization primitives such as locks, monitors, synchronized blocks, as well as thread creation and joining.

ConDpor can operate in two modes: a verification mode, and a "random mode" in which ConDpor samples executions from the program state space. The latter mode is useful for finding bugs faster, as random sampling does not involve backtracking.

Verifying a program with ConDpor involves three steps:

(1) compiling it into Java bytecode,

(2) instrumenting the bytecode so that we can intercept operations of interest, and

[5] `https://github.com/mpi-sws-rse/jmc`

(3) running the instrumented bytecode under a special runtime that implements ConDpor to systematically explore all program behaviors.

Even though our algorithm is parametric in the choice of the memory model, our implementation currently only works under SC [51].

To instrument the bytecode, we used the ASM library [1], and inserted yield calls to ConDpor at points of interest. For example, whenever a thread acquires or releases a lock, the instrumented bytecode yields control to ConDpor, which then updates the execution graph and schedules threads appropriately. ConDpor controls thread scheduling by employing a shared mutex between the native Java Thread and the scheduler thread. We implement the `yield` method as `Object.wait` over this shared mutex, effectively imposing a cooperative multithreading semantics on top of the Java runtime.

To return appropriate values during constraint evaluation, ConDpor uses the JavaSMT library [13] to query an SMT solver for satisfiability. Although JavaSMT can interface with multiple solvers, we have only experimented with `Z3` [3]. Similarly to TruSt [45], ConDpor copies the graph whenever a write revisits a read, meaning that a new solver instance has to be created as well. We found that creating new instances is expensive, and implemented a shared solver pool with a fixed number of instantiated solvers to allow solver reuse. Each solver instance also leverages incremental solving, which turned out to be very beneficial.

Finally, note that users must utilize our symbolic API to capture all sources of data nondeterminism manually: in practice, this involves subclassing some input types to their symbolic counterparts. Note that some existing symbolic (or concolic) execution engines, like EXE [20], employ source-to-source translation, where the instrumentation is written in the same programming language as the program under test. While a source-to-source translation is also possible for Java, we decided to implement ConDpor using bytecode instrumentation and by encoding symbolic objects directly in the language, so as to avoid maintaining a Java compiler infrastructure.

6 Evaluation

We now proceed with ConDpor's evaluation, which aims to answer the following questions:

§ 6.1 How does symbolic handling of data nondeterminism fare against an explicit one?

§ 6.2 Can ConDpor handle real-world programs that employ data nondeterminism?

To answer these questions, we conduct two case studies. To showcase the differences between explicit and symbolic handling of data nondeterminism, we evaluate ConDpor on a set of synthetic benchmarks designed to juxtapose the two approaches. To see how well ConDpor scales in realistic code, we use a diverse set of concurrent data structures that rely on randomness (modeled as data nondeterminism). In both studies, we compare ConDpor against a baseline DPOR implementation ExEn that handles data nondeterminism explicitly by exhaustively enumerating all possible values in the input domain, and repeatedly running vanilla DPOR for each value. To ensure a fair comparison, we restrict the data domain to $\{0, 1, 2\}$, making explicit enumeration feasible.

Our evaluation demonstrates that ConDpor's handling of nondeterminism is exponentially better than ExEn's, and that ConDpor is able to completely explore clients of data-nondeterministic concurrent data structures. Moreover, ConDpor's overhead over vanilla DPOR is proportional to the amount of data nondeterminism in the input program.

Table 1 ConDpor performs better than ExEn even for small data domains.

	ExEn			ConDpor		
	Execs	*Time*	*Mem*	*Execs*	*Time*	*Mem*
`SVQueue1(8)`	39 366	365	73	4	0	77
`SVQueue1(9)`	118 098	1743	73	4	0	71
`SVQueue1(10)`	⏱	⏱	⏱	4	0	70
`SVQueue2(2)`	1242	5	79	138	1	70
`SVQueue2(3)`	92 556	398	108	3428	13	100
`SVQueue2(4)`	⏱	⏱	⏱	86 708	530	118
`SVQueue3(7)`	⏱	⏱	⏱	2865	27	100
`SVQueue3(8)`	⏱	⏱	⏱	6506	70	130
`SVQueue3(9)`	⏱	⏱	⏱	14 505	175	158
`SVStack1(5)`	183 222	1481	126	754	5	108
`SVStack1(6)`	⏱	⏱	⏱	2770	26	131
`SVStack1(7)`	⏱	⏱	⏱	10 294	112	148
`SVStack2(6)`	⏱	⏱	⏱	1481	14	123
`SVStack2(7)`	⏱	⏱	⏱	4876	54	153
`SVStack2(8)`	⏱	⏱	⏱	16 422	213	172
`Counter1(4)`	11 988	32	70	2368	9	78
`Counter1(5)`	317 115	1044	100	41 760	217	90
`Counter1(6)`	⏱	⏱	⏱	883 584	5540	200
`Counter2(5)`	66 465	215	180	5770	25	140
`Counter2(6)`	1 411 770	6368	200	69 852	403	155
`Counter2(7)`	⏱	⏱	⏱	970 886	6929	220

Experimental Setup

We conducted all experiments on a Dell Latitude 5450 system running a custom Debian-based distribution with an Intel Core Ultra 5 135H CPU (18 cores @ 4.60 GHz) and 32GB of RAM.

In all tables, *Execs* denotes the number of executions explored, *Time* the required time (in seconds), and *Mem* the required memory (in MB). We set a timeout of 210 minutes and a memory limit of 512MB (denoted by ⏱ and OOM).

6.1 Explicit vs Symbolic Data Nondeterminism

Let us begin by comparing ConDpor to ExEn on synthetic benchmarks. For this part of our evaluation, we used all benchmarks employing data nondeterminism from SV-COMP's `pthread` category [67], as well as two handcrafted ones. Our results can be seen in Table 1.

The main takeaway from this table is that, despite the small data domain, for each benchmark there is an input parameter (the size of shared buffer in SV-COMP benchmarks and the number of threads in the handcrafted ones) for which ExEn times out, whereas ConDpor scales beyond it. Moreover the memory increase over ExEn (which consumes polynomial memory) is negligible due to the tests having only a few constraints.

Let us now examine the benchmarks of Table 1 more carefully to gain a better understanding of the differences between the two approaches. Starting with `SVQueue*(N)` and `SVStack*(N)`, these benchmarks implement a simple data structure (queue and stack, respectively) using an array of size N, while a producer and a consumer thread add/remove N symbolic integers using a shared lock. (Queue/stack tests only differ in the way the threads synchronize when modifying the data structure.) As can be seen, ExEn cannot

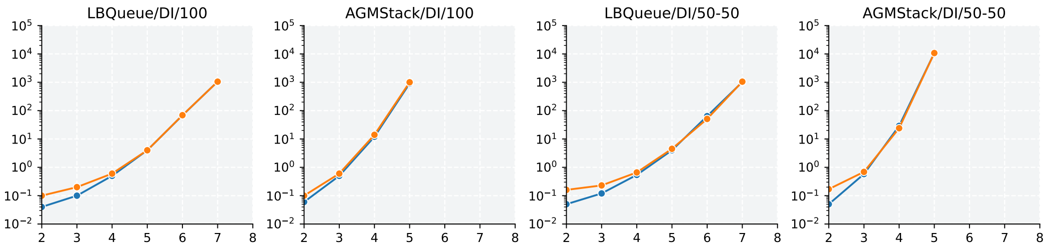

Figure 3 ConDpor adds no overhead over DPOR on data-independent tests (time/threads).

scale beyond very small arrays since, in addition to all possible schedules, it also explicitly enumerates all values for each array cell. ConDpor, on the other hand, scales nicely even for larger arrays, as it only examines whether a removed item matches a previously inserted one. We can draw similar conclusions for the `Counter*(N)` benchmarks, where each of N threads nondeterministically operates on one of two shared counters. Although there is a single source of data nondeterminism in each thread, ExEn quickly times out for larger Ns.

6.2 Verifying Concurrent Data-Structures

Let us now move on to applying ConDpor to realistic concurrent data structures. We classify these benchmarks into three categories, depending on their amount of data nondeterminism.

- *data-independent data structures* (lock-based queue [38] and Afek-Gafni-Morrison Stack [10]). These do not perform any computation on the arguments of their methods.
- *data-dependent data structures* (coarse-grained list [38], fine-grained list [14], optimistic list [38] and lazy list [36]). These check equality or ordering of their method arguments.
- *nondeterminism-based data structures* (timestamped stack [26] and elimination backoff stack [37]). These are data-independent on their arguments, but internally rely on data nondeterminism for improved performance.

We ran ConDpor on each category under two different workloads: a "100-0" workload, where N threads are inserting a symbolic integer to the data structure, and a "50-50" workload where $\lceil \frac{N}{2} \rceil$ threads insert an item and $\lfloor \frac{N}{2} \rfloor$ threads remove an item. (Tables including all experimental results against ExEn can be found in the appendix of our extended version [44].)

6.2.1 Data-Independent Data Structures

We begin by measuring ConDpor's overhead over vanilla DPOR on some data-independent data structures (cf. Fig. 3). As these data structures do not use the values of their arguments, vanilla DPOR and ConDpor explore the same number of executions. What these benchmarks demonstrate, however, is that manipulating symbolic integers in ConDpor does not induce any overhead, as no calls to the SMT solver are performed thanks to concolic execution. Indeed, ConDpor randomly concretizes all nondeterministic values, and never encounters constraint evaluation nodes, allowing it to scale in exactly the same manner as vanilla DPOR. (The small overhead of ConDpor for small input parameters is due to the solver initialization; see §5.)

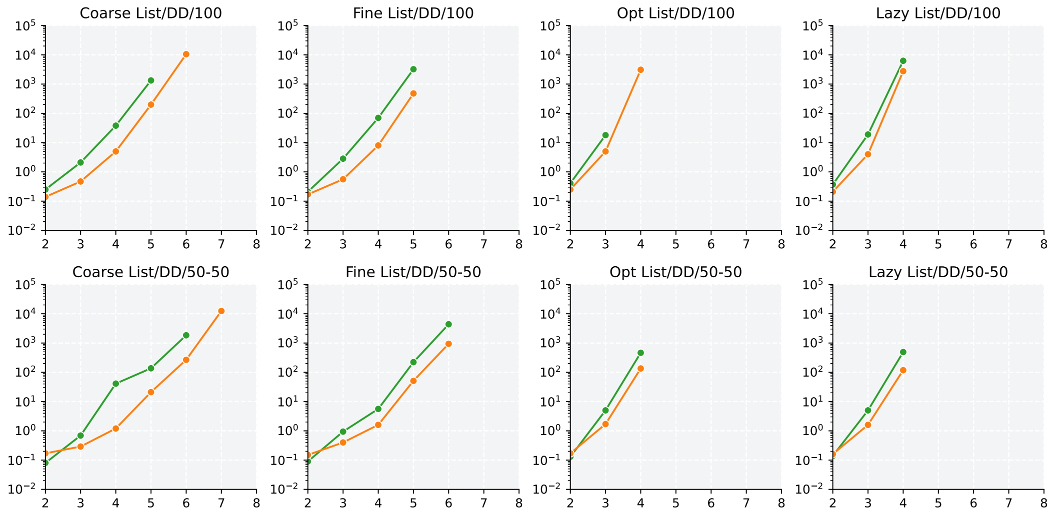

Figure 4 ConDpor scales better than ExEn on data-dependent tests (time/threads).

6.2.2 Data-dependent Data Structures

Data-dependent data structures, on the other hand, do incur some overhead over concurrency nondeterminism. A comparison between ConDpor and ExEn on such structures can be seen in Fig. 4. As expected, ConDpor scales better than ExEn in all benchmarks, and often scales to more threads than ExEn (e.g., in `Coarse List/DD/50-50`).

What we also observe in these benchmarks, however, is that most time is not spent on evaluating symbolic constraints, but rather on enumerating executions (see Appendix A.3 of [44]).

Indeed, taking `Coarse List/DD/100` as an example, ConDpor spends considerably less time evaluating constraints than enumerating executions: for 6 threads, only 12.5% of the total time was spent on solving constraints [44], despite the fact that there are $N!$ ways the symbolic constraints can be evaluated in each of the $N!$ concurrent behaviors. This is because the time per execution is greater than the time required to solve a single constraint. We also observe that ConDpor scales better for the "50-50" workload than "100-0" workload, as in the former ConDpor does not have to evaluate constraints if the data structure is empty.

6.2.3 Nondeterminism-based Data Structures

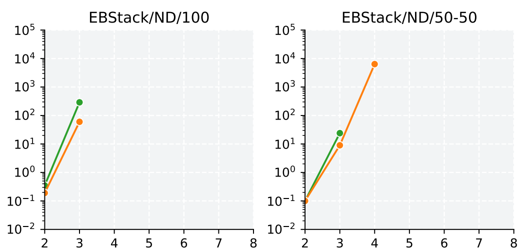

Figure 5 ConDpor outperforms ExEn for data structures employing randomness (time/threads).

The last class of benchmarks discusses data structures that exploit nondeterminism at the core of their operations. Such structures are interesting because although the data domain might be small (e.g., if we use a small elimination array in an elimination structure), ConDpor still provides significant benefits over ExEn or vanilla DPOR (assuming an equivalent encoding is possible).

Similar to the data-dependent data structures, `EBStack/ND/50-50`, as depicted in Figure 5, this confirms the scalability of ConDpor as the input parameter increases compared to ExEn. For instance, in `EBStack` with an elimination array of size $\lceil \frac{N}{2} \rceil$ for "50-50" workload, ExEn still enumerates all values in the data domain, even in the case where threads do not touch the elimination array; by contrast, ConDpor does not evaluate any constraints.

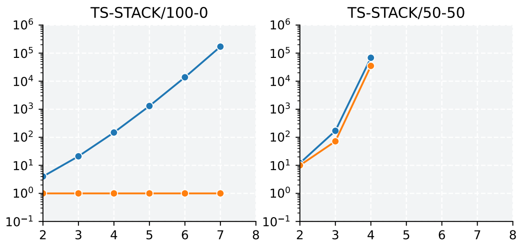

Figure 6 Vanilla DPOR quickly times out when the parameter of `TSStack/100-0` is increased. Also, ConDpor using symbolic encoding explores fewer executions than vanilla DPOR in both workloads.

Let us conclude by discussing `TSStack` in more detail, as this data structure was our motivating example in §2. As discussed, this benchmark can be encoded both concretely (using an atomic shared counter), as well as symbolically (using thread-local symbolic counters). As such, we perform the comparison between ConDpor and vanilla DPOR. Based on the `TSStack/100-0` workload in Figure 6, ConDpor explores a single execution. This is because ConDpor encodes timestamping symbolically, and hence push operations are never ordered if there are no pop operations (see § 2.1). Vanilla DPOR, on the other hand, eagerly explores all possible orderings for timestamps, thereby quickly timing out, as depicted in Figure 6. Moreover, as the `TSStack/50-50` workload demonstrates in Figure 6, the symbolic encoding is superior, as ConDpor explores fewer executions than DPOR. This would be even more pronounced using a workload similar to that of client NPUSH+POP in § 2.1.

7 Related Work

As far as handling data nondeterminism is concerned, a lot of research has focused around symbolic execution (e.g., [16, 68, 61]), which uses a symbolic interpreter to examine many program paths at once [43]. As symbolic execution is a static technique and the number of symbolic paths to explore explodes, recent work [31, 19, 32, 21, 53, 66] has focused on dynamic symbolic execution (aka concolic testing), where certain symbolic variables are concretized in order to reduce the state-space size. Despite their success, most symbolic tools do not reason about concurrency and weak memory consistency.

For scheduling nondeterminism, most automated tools are based on either testing or model checking. Testing tools (e.g., [41, 49, 52, 17, 2, 18, 71]) randomly sample the state space of a program in order to detect concurrency bugs, but they do not provide completeness guarantees, nor do they handle data-nondeterminism. Model checkers (e.g., [39, 35]), on the other hand, do guarantee completeness, as they exhaustively enumerate the state space of a program (up to a bound). In order not to store all the visited program states, model checking is often done "on the fly" (a.k.a. stateless model checking) [55, 30], while also employing partial order reduction [29, 7, 56, 45, 27, 47, 33, 4, 9]. While model checkers can handle data nondeterminism explicitly, explicit enumeration quickly blows up the state space (see §6).

There has been work aiming to handle both scheduling and data nondeterminism (e.g., [28, 65, 70]), either by adding concurrency reasoning to concolic testing, or the other way around. Con2Colic [28] extends CT by introducing interference scenarios to encode the scheduling constraints that lead to new execution paths. Similarly, Guo et al. [34] encode program assertions as symbolic constraints to develop sound methods that prune out redundant executions. Tools like CMBC [24] encode the program into an SAT formula, and offload all symbolic reasoning to an SMT solver. JPF has been extended to incorporate symbolic execution [12]. None of these tools perform optimal partial order reduction for general memory consistency models. Finally, there are several works that attempt to efficiently combine DPOR with symbolically encoded partial-order constraints to reduce redundant exploration (e.g., [11, 42]). Unlike our approach, however, these do not tackle the problem of data nondeterminism within concurrent programs.

Another approach to addressing both scheduling and data non-determinism is maximal path causality (MPC) [69]. MPC uses a coarser equivalence partitioning than ConDpor, as it packs all combinations of schedules and data that reach the same path into a single equivalence class. Upon obtaining a random program interleaving, MPC collects constraints that would lead to the exploration of an unseen path, and explores them by re-running the program.

$a :=$ `nondet`$()$
$b :=$ `nondet`$()$
if $(a > 0)$ $x := 1$
else
  if $(b > 0)$ $x := 3$
  else $x := 4$
	`assert`$(x \neq 2)$

Apart from not being able to handle weak memory consistency models, MPC is not optimal w.r.t. its partitioning. As an example, consider the program on the right. Assuming MPC first obtains the interleaving $\texttt{C}(a > 0) \cdot \texttt{W}(x, 1) \cdot \texttt{R}(x)$, it will then generate the constraint $a \leq 0 \wedge \texttt{val}(\texttt{R}(x)) = 2$, aiming to uncover the assertion violation. While this constraint (and all paths stemming from it) are infeasible, MPC cannot drop it, as some of the paths under the "else" branch could have been writing $x := 2$.

The closest works to ConDpor are those that aim to combine DPOR and CT. One of the such attempts was jCUTE [60]. Unlike ConDpor, jCUTE uses a non-optimal DPOR (and therefore explores redundant executions), and does not support weak memory consistency. Another work is that of Schemmel et al. [58], who combine quasi-optimal partial order reduction with CT. However, the resulting algorithm (PorSe) is not optimal, does not support weak memory consistency, has exponential memory requirements, and, since it does not use concrete execution, incurs a costly and unnecessary extra query to the SMT solver upon each symbol constraint evaluation. In our extended version [44] (App. A.5), we present a comparison between ConDpor and PorSe using the SV-COMP benchmarks. This comparison shows that, on average, ConDpor reduces the number of explored executions by 72% compared to PorSe across SV-COMP benchmarks.

8 Conclusion

We presented CONDPOR, a sound, complete and optimal algorithm that integrates concolic execution into dynamic partial order reduction. CONDPOR uses execution graphs for backtracking on both scheduling and symbolic constraints, and offloads all manipulation of symbolic expressions to the underlying language's runtime. CONDPOR significantly outperforms DPOR approaches that explicitly enumerate the values of a data domain (even with that domain is small), and can verify challenging concurrent data structures.

References

1 asm / asm · GitLab — gitlab.ow2.org. `https://gitlab.ow2.org/asm/asm`. [Accessed 29-01-2025].

2 GitHub - openjdk/jcstress. `https://github.com/openjdk/jcstress`. [Accessed 31-01-2025].

3 GitHub - Z3Prover/z3: The Z3 Theorem Prover — github.com. `https://github.com/Z3Prover/z3`. [Accessed 30-01-2025].

4 Parosh Abdulla, Mohamed Faouzi Atig, S Krishna, Ashutosh Gupta, and Omkar Tuppe. Optimal stateless model checking for causal consistency. In *TACAS*, pages 105–125. Springer, 2023.

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

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

7 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. Parsimonious optimal dynamic partial order reduction. In *CAV 2024*, pages 19–43, Cham, 2024. Springer. `doi:10.1007/978-3-031-65630-9_2`.

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

9 Parosh Aziz Abdulla, Ashutosh Gupta, Shankara Narayanan Krishna, and Omkar Tuppe. Dynamic partial order reduction for transactional programs on serializable platforms. In *International Symposium on Automated Technology for Verification and Analysis*, pages 181–202. Springer, 2024. `doi:10.1007/978-3-031-78709-6_9`.

10 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Comput.*, 20(4):239–252, 2007. `doi:10.1007/S00446-007-0023-3`.

11 Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. Optimal dynamic partial order reduction with context-sensitive independence and observers. *J. Syst. Softw.*, 202:111730, 2023. `doi:10.1016/J.JSS.2023.111730`.

12 Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *TACAS 2007*, pages 134–138, Berlin, Heidelberg, 2007. Springer. `doi:10.1007/978-3-540-71209-1_12`.

13 Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. Javasmt 3: Interacting with SMT solvers in java. In Alexandra Silva and K. Rustan M. Leino, editors, *CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2021. `doi:10.1007/978-3-030-81688-9_9`.

14 Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977. `doi:10.1007/BF00263762`.

15 Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS 2014*, pages 239–254, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2541940.2541977.

16 Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys 2011*, pages 183–198, New York, NY, USA, 2011. ACM. doi:10.1145/1966445.1966463.

17 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-Up: A complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI 2010*, pages 330–340. ACM, 2010. doi:10.1145/1806596.1806634.

18 Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS 2010*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM. doi:10.1145/1736020.1736040.

19 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI 2008*, pages 209–224, USA, 2008. USENIX Association. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

20 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), December 2008. doi:10.1145/1455518.1455522.

21 Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012*, pages 380–394. IEEE Computer Society, 2012. doi:10.1109/SP.2012.31.

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

23 Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012. doi:10.1145/2110356.2110358.

24 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS 2004*, volume 2988 of *LNCS*, pages 168–176, Berlin, Heidelberg, 2004. Springer. doi:10.1007/978-3-540-24730-2_15.

25 Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. A partial order view of message-passing communication models. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571248.

26 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *POPL 2015*, pages 233–246, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676963.

27 Constantin Enea, Dimitra Giannakopoulou, Michalis Kokologiannakis, and Rupak Majumdar. Model checking distributed protocols in must. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689778.

28 Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *ESEC/FSE*, pages 37–47. ACM, 2013. doi:10.1145/2491411.2491453.

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

30 Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL 1997*, pages 174–186, New York, NY, USA, 1997. ACM. doi:10.1145/263699.263717.

31 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI 2005*, pages 213–223, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065036.

32 Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008. URL: `https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/`.

33 Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *VMCAI 2017, Paris, France, Proceedings 18*, pages 246–265. Springer, 2017. `doi:10.1007/978-3-319-52234-0_14`.

34 Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE*, pages 854–865. ACM, 2015. `doi:10.1145/2786805.2786841`.

35 Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *Int. J. Soft. Tool. Tech. Transf.*, 2(4):366–381, 2000. `doi:10.1007/S100090050043`.

36 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS 2005*, pages 3–16, Berlin, Heidelberg, 2005. Springer-Verlag. `doi:10.1007/11795490_3`.

37 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA 2004*, pages 206–215, New York, NY, USA, 2004. ACM. `doi:10.1145/1007912.1007944`.

38 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann Publishers Inc., 2008.

39 G.J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. `doi:10.1109/32.588521`.

40 Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI 2015*, pages 165–174, New York, NY, USA, 2015. ACM. `doi:10.1145/2737924.2737975`.

41 Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *NFM 2009, Moffett Field, California, USA, April 6-8, 2009*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 121–125, 2009.

42 Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In Ahmed Bouajjani and Oded Maler, editors, *CAV 2009, Grenoble, France, Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009. `doi:10.1007/978-3-642-02658-4_31`.

43 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. `doi:10.1145/360248.360252`.

44 Michalis Kokologiannakis, Mohammad Hossein Khoshechin Jorshari, Srinidhi Nagendra, and Rupak Majumdar. Optimal concolic dynamic partial order reduction, June 2025. `doi:10.5281/zenodo.15649583`.

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

46 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. Unblocking dynamic partial order reduction. In Constantin Enea and Akash Lal, editors, *CAV 2023*, volume 13964 of *LNCS*, pages 230–250. Springer, 2023. `doi:10.1007/978-3-031-37706-8_12`.

47 Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. SPORE: Combining symmetry and partial order reduction. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. `doi:10.1145/3656449`.

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

49 Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In Constantin Enea and Akash Lal, editors, *CAV 2023*, pages 156–169, Cham, 2023. Springer. `doi:10.1007/978-3-031-37706-8_8`.

50 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062352.

51 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, September 1979. doi:10.1109/TC.1979.1675439.

52 Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. Fray: An efficient general-purpose concurrency testing platform for the jvm, 2025. doi:10.48550/arXiv.2501.12618.

53 Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. Legion: Best-first concolic testing (competition contribution). In Heike Wehrheim and Jordi Cabot, editors, *FASE 2020, Dublin, Ireland, Proceedings*, volume 12076 of *Lecture Notes in Computer Science*, pages 545–549. Springer, 2020. doi:10.1007/978-3-030-45234-6_31.

54 Antoni Mazurkiewicz. Trace theory. In *PNAROMC 1987*, volume 255 of *LNCS*, pages 279–324, Berlin, Heidelberg, 1987. Springer. doi:10.1007/3-540-17906-2_30.

55 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI 2008*, pages 267–280. USENIX Association, 2008. URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.

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

57 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, pages 391–407. Springer, 2009. doi:10.1007/978-3-642-03359-9_27.

58 Daniel Schemmel, Julian Büning, Cesar Rodriguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In Shuvendu K. Lahiri and Chao Wang, editors, *CAV*, volume 12224, pages 376–400. Springer, 2020. doi:10.1007/978-3-030-53288-8_18.

59 Koushik Sen and Gul Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *CAV 2006*, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11817963_38.

60 Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *HVC 2006*, pages 166–182, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/978-3-540-70889-6_13.

61 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *ESEC/FSE 2013*, pages 263–272, New York, NY, USA, 2005. ACM. doi:10.1145/1081706.1081750.

62 Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988. doi:10.1145/42190.42277.

63 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

64 Shuttle. URL: https://github.com/awslabs/shuttle.

65 Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. Civl: the concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2807591.2807635.

66 Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. Symbolic execution for randomized programs. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1583–1612, 2022. doi:10.1145/3563344.

67 SV-COMP. Competition on software verification (SV-COMP), 2019. URL: `https://sv-comp.sosy-lab.org/2019/`.

68 Nikolai Tillmann and Jonathan de Halleux. Pex–White box test generation for .NET. In *TAP 2008*, pages 134–153, Berlin, Heidelberg, 2008. Springer. `doi:10.1007/978-3-540-79124-9_10`.

69 Qiuping Yi and Jeff Huang. Concurrency verification with maximal path causality. In *ESEC/FSE 2018*, pages 366–376, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3236024.3236048`.

70 Hengbiao Yu, Zhenbang Chen, Xianjin Fu, Ji Wang, Zhendong Su, Jun Sun, Chun Huang, and Wei Dong. Symbolic verification of message passing interface programs. In *ICSE 2020*, pages 1248–1260, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3377811.3380419`.

71 Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *CAV 2018*, pages 317–335, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-319-96142-2_20`.