PSYM: Efficient Symbolic Exploration of Distributed Systems

LAUREN PICK, University of California, Berkeley and University of Wisconsin-Madison, USA
ANKUSH DESAI, Amazon Web Services, USA
AARTI GUPTA, Princeton University, USA

Verification of distributed systems using systematic exploration is daunting because of the many possible interleavings of messages and failures. When faced with this scalability challenge, existing approaches have traditionally mitigated state space explosion by avoiding exploration of redundant states (e.g., via state hashing) and redundant interleavings of transitions (e.g., via partial-order reductions). In this paper, we present an efficient symbolic exploration method that not only avoids redundancies in states and interleavings, but additionally avoids redundant computations that are performed during updates to states on transitions. Our symbolic explorer leverages a novel, fine-grained, canonical representation of distributed system configurations (states) to identify opportunities for avoiding such redundancies on-the-fly. The explorer also includes an interface that is compatible with abstractions for state-space reduction and with partial-order and other reductions for avoiding redundant interleavings. We implement our approach in the tool PSYM and empirically demonstrate that it outperforms a state-of-the-art exploration tool, can successfully verify many common distributed protocols, and can scale to multiple real-world industrial case studies across Amazon.

CCS Concepts: • **Computing methodologies → Distributed programming languages**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: distributed systems, systematic exploration, binary decision diagrams

1 INTRODUCTION

Verification of distributed systems is challenging because of the need to reason about all possible behaviors resulting from a myriad interleavings of messages and failures. Existing approaches fall into two broad categories: interactive theorem-proving with specialized logics (e.g., [Jung et al. 2015; Sergey et al. 2018]), and automatic systematic exploration (e.g., [Desai et al. 2013a; Holzmann 1997; Lamport 2002]). Researchers have used theorem provers to construct correctness proofs for complex distributed systems [Hawblitzel et al. 2015; Padon et al. 2016; Wilcox et al. 2015]. Although these proofs of correctness are invaluable, theyrequire significant manual effort. In this paper, we are interested in mostly-automated systematic-exploration-based approaches that need less expert guidance. However, in practice, such approaches can achieve correctness guarantees only for small bounded instances of systems because of the state-space explosion problem, which results in poor scalability with increasing system complexity. Hence, there is a need for more efficient techniques that can push the limits on the complexity of systems that can be verified using systematic exploration.

Explicit-state model-checkers (stateful explorers) (e.g., TLC [Yu et al. 1999], Zing [Andrews et al. 2004], and SPIN [Holzmann 1997]) have been widely successful in industry and academia for verification of distributed systems [Newcombe 2014; TLA$^+$ 2023]. Most stateful explorers address the state explosion problem by using state caching (hashing) to avoid re-exploring already visited states, thereby avoiding redundant computation. Explorers also use partial-order reduction (POR) [Clarke et al. 2001; Peled 2018] and its dynamic variants [Abdulla et al. 2014; Flanagan and Godefroid 2005; Nguyen et al. 2018; Tasharofi et al. 2012] to avoid exploring interleavings that are redundant due to independent transitions.

In distributed systems, transitions operate only on the local state of a process (no shared memory), and it is common to have multiple instances of the same process in the system that execute the

Authors' addresses: Lauren Pick, pick@berkeley.edu, University of California, Berkeley and University of Wisconsin-Madison, USA; Ankush Desai, ankushpd@amazon.com, Amazon Web Services, USA; Aarti Gupta, aartig@cs.princeton.edu, Princeton University, USA.

same code but have different local state (e.g., replicas in a storage system [Chang and Roberts 1979], proposer and acceptors in Paxos [Lamport 2001]). As a result, treating distributed system configurations and transitions monolithically, as is done in existing techniques, may still lead to redundant computations in updates due to transitions that "overlap," i.e., share computations that update the same (or partially same) local state (defined in §4.1). We are thus motivated to improve upon existing techniques and present an efficient symbolic explorer that *recognizes and exploits redundancies not only in configurations (states) and interleavings, but also in the computations in overlapping transitions.*

Our Approach. To ground our contributions in a real-world setting, we consider distributed systems modeled with P [Desai et al. 2013a, 2018], a state-machine-based programming language for modeling and specifying distributed systems. P is being used across industry and academia for analysis of complex distributed systems [Desai 2022; GitHub 2021]; e.g., to reason about Amazon S3's core distributed protocols [GitHub 2021] and the USB device driver stack that shipped with Microsoft Windows 8 [Desai et al. 2013a]. Teams across Amazon are using P to reason about the core distributed protocols driving their services. We present our approach as a symbolic explorer for P programs, though our ideas apply to systematic exploration of distributed systems in general.

As a first step for efficient systematic exploration of P programs, we adapt macro-step semantics for actor systems [Agha et al. 1997], on top of which we design a novel symbolic stateful explorer. For our explorer to identify redundancies in transitions on-the-fly, we propose a novel, fine-grained symbolic representation of sets of configurations. Our symbolic representation is inspired by *value summaries* (sets of guard-value pairs) used in MultiSE [Sen et al. 2015], a systematic exploration method for sequential programs. We adapt value summaries to distributed systems by introducing *Schedule-Control-Input guards (SCI Guards)* to capture symbolic scheduling choices in addition to control and input nondeterminism. We also propose *composite* value summaries (for tuples, lists, maps, etc.) and symbolic operations on them to get fine-grained representations, which we show are *canonical*. Canonicity helps our explorer identify redundant configurations as well as overlapping transitions.

We lift the macro-step P semantics to operate over these symbolic representations, where a single *symbolic transition* in the lifted semantics can capture *multiple overlapping transitions* in the original semantics, avoiding redundant computations in these transitions. We design our explorer so that it includes an interface compatible with abstractions for state-space reduction and with partial-order and other reductions for avoiding redundant interleavings. To avoid redundant interleavings, our explorer supports a persistent-set based partial-order reduction [Clarke et al. 2001; Peled 2018] that we adapt to P. We provide theoretical guarantees for the soundness and efficiency of our explorer.

Psym. We implemented our approach in Psym, a symbolic explorer for P programs. We compare Psym with the state-of-the art stateful explorer TLC (model checker for TLA$^+$ [Lamport 2002; Yu et al. 1999]) and show that Psym outperforms TLC on many open source distributed system benchmarks, finishing verification when TLC times out on two benchmarks, and achieving a runtime improvement of 2.5X over TLC on average (geometric mean) on the remaining benchmarks. Psym can also successfully verify common distributed protocols and challenging industrial case-studies of four complex real-world distributed protocols used at Amazon Web Services (AWS) where other verification tools failed.

Contributions. In summary, we make the following main contributions:

- We present a novel, fine-grained symbolic representation of configurations in a distributed system (§4) that helps avoid redundancies during exploration. Inspired by MultiSE, we extend canonical value summaries to include *schedule* nondeterminism and represent common *composite* data structures (e.g., tuples, lists, maps) in a fine-grained, decomposed manner. To the best of

our knowledge, our work is the first to use value summaries for distributed systems. These representations are not specific to P and could be used by other distributed systems frameworks.
- We propose a new symbolic stateful explorer for P programs (§5), where the macro-step semantics (§3) is lifted to leverage our novel value summaries for efficient exploration of reachable configurations while avoiding redundancies in configurations, interleavings, and overlapping transitions. Additionally, it can use abstractions (§6.1) to handle infinite-state systems.
- We implemented our ideas in a prototype tool Psym that includes an extensible filter interface (§6.2) for integrating POR and other reductions with our symbolic explorer.
- We demonstrate Psym's efficacy on real-world benchmarks, including industrial scale distributed protocols (§8).

2 MOTIVATING EXAMPLE AND KEY IDEAS

We first introduce P and then highlight the key ideas of our approach on a motivating example.

2.1 P Language

P is a state-machine-based programming language for modeling and specifying complex distributed systems. P was first used to implement and validate the USB device driver stack that ships with Microsoft Windows 8 and Windows Phone [Desai et al. 2013a] and is used extensively in Amazon (AWS) for formal modeling and analysis of complex distributed systems such as the core distributed protocols involved in Amazon S3's strong consistency launch [Desai 2022; Desai et al. 2021; GitHub 2021]. P currently leverages randomized stateless exploration [Desai et al. 2015; Microsoft Coyote 2022] to find critical bugs in industrial-scale distributed protocols. Randomized search (run on a distributed cluster) is highly effective in finding low-probability bugs but fails to provide correctness guarantees. Hence, there is a need for a verification backend for P that can handle industrial-scale systems.

```
1  machine Server {
2  var reg : machine;
3   start state Init {
4    entry {
5     var i : int;
6     reg = new Registry();
7     while(i< 3)//create workers
8     {
9      var w = new Worker(i);
10     //send workitem to worker
11     send w, eWorkItem, reg;
12     i++;
13    }
14   }
15  }
16 }
```

(a) Server State Machine

```
17
18 // event to send work-item to worker
19 event eWorkItem: machine;
20
21 machine Worker {
22  var id : int;
23  start state Init {
24   entry (arg : int){ id = arg; }
25    // receive work-item (w0, w1, w2)
26   on eWorkItem do (reg : machine) {
27    // register the worker at registry
28    send reg, eRegisterWorker, id;
29   }
30  }
31 }
```

(b) Worker State Machine

```
32
33 // event to register the worker id
34 event eRegisterWorker: int;
35
36 machine Registry {
37  var workerIds : set[int];
38  start state Init {
39   // receive register request
40   // (r0, r1, r2)
41   on eRegisterWorker do (id : int) {
42    // add the id's to workerIds set.
43    workerIds += (id);
44   }
45  }
46 }
```

(c) Registry State Machine

Fig. 1. An example P program (adapted from the TransDPOR paper [Tasharofi et al. 2012]).

2.2 Motivating Example in P

A P program is a collection of concurrently executing state machines that communicate with each other by sending messages (i.e., events and payloads) asynchronously. (The underlying model of computation is similar to actors [Agha 1986].) Fig. 1 presents a simple P program adapted from an example in the TransDPOR work [Tasharofi et al. 2012]. It consists of three types of state machines: `Server` (line 1), `Worker` (line 21), and `Registry` (line 36). The `Server` creates a set of `Workers` and sends them work items to be processed, and the `Registry` maintains the set of all workers in the system. State machines in P communicate by sending events with payloads. Line 19 declares the `eWorkItem` event that has an associated payload of machine reference type. Each

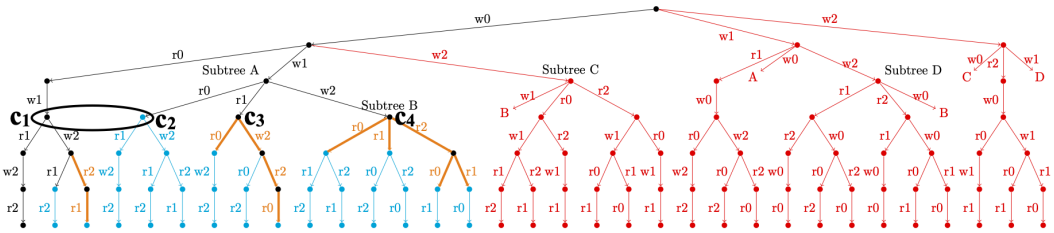

Fig. 2. Full execution tree for the example in Fig. 1, with the reductions due to POR shown in red. For the remaining executions, our symbolic explorer avoids the redundant work shown in blue and orange.

machine has a `start` state (e.g., line 3), where it starts execution after being created. Each state has an `entry` handler that is executed upon entering that state. The entry handler for the `Init` state of `Server` creates one `Registry` machine (line 6) and three `Worker` machines (line 9), and then sends each `Worker` a work item to be processed (line 11) along with a reference to the `Registry` machine. Each machine can also have an associated set of local variables (e.g., `Registry` has a local variable `workerIds`). After executing an `entry` procedure, a machine blocks to receive an event. On receiving an event, the event's *handler* is executed, transitioning the system from one configuration (global state) to another. If there are no messages in its buffer, a machine blocks until a message is received. For example, the `Registry` machine, after entering the `Init` state, blocks to receive an event. On receiving the `eRegisterWorker` event, it executes the corresponding event handler (line 41) that adds the `id` in the payload to the local set. The `Server`, `Registry`, and three `Worker` machines execute concurrently, asynchronously sending messages to each other. In our example, the program's initial configuration is one in which one `Server` instance has been created.

2.3 Systematic Exploration: Baseline Macro-step Semantics and POR for P

As a first step for efficient systematic exploration of P programs, we adapt *macro-step semantics* for P (§3). Macro-step semantics has been demonstrated to drastically reduce the number of interleavings for sound verification in actor systems [Sen and Agha 2006a]. The basic idea is to treat transitions starting from (and including) a receive of a message at an actor, up until (but not including) the next receive of a message at that actor, as a single atomic step. Fig. 2 shows the execution tree for the motivating example under macro-step semantics. Each node denotes a *configuration* (i.e., a global state) of the program, and each branch corresponds to an event received by a machine in the system. (For simplicity of exposition, we omit branches corresponding to dynamic machine creation though we handle this in Psym.) We have labeled the receives of each event (see lines 28 and 40 in Fig. 1): `w0`, `w1`, `w2` represent the receive of `eWorkItem` events at the `Worker` machine instances with `id`s 0, 1, 2, respectively. Similarly, `r0`, `r1`, `r2` represent the receive of `eRegisterWorker` at the `Registry` machine instance with payloads 0, 1, 2, respectively.

A naive systematic explorer would explore the *entire* execution tree shown in Fig. 2, which is already based on macro-step semantics. We improve upon this by applying a persistent-set-based POR technique for P programs called Pred (§6.2). When applying Pred to our example, the parts in red are found to be redundant and not explored. For real-world distributed systems, even with POR, there remain redundancies during exploration that present a challenge for scalability.

2.4 Symbolic Stateful Exploration using Value Summaries

Popular explicit-state model checkers (e.g., SPIN [Holzmann 1997], TLC [Yu et al. 1999]) leverage state caching to avoid exploring revisited states, along with applying POR to avoid exploring redundant interleavings. In Fig. 2, all blue subtrees need not be explored if state caching is used. For example, c_2 is the same configuration as c_1 and will not be explored if c_1 is visited first. However, such approaches will separately compute the orange transitions in Fig. 2. Because these transitions arise

from two different configurations, traditional explicit-state explorers will perform the computations separately for each configuration. In this work, we aim to *eliminate such redundant computations.*
Symbolic State Exploration. Our approach is based on *symbolic* representations of configurations. Similar to well-known symbolic exploration algorithms [Chaki and Gurfinkel 2018], we compute a set of *frontier* configurations starting from an initial configuration, where each symbolic step considers possibly *multiple* transitions from the frontier set to compute the set of next configurations. *The novelty in our work is in our new symbolic representations of configurations of distributed systems (§4) and in our explorer's ability to avoid redundant computations in overlapping transitions (§4.1, §5).*
Putting It All Together. The blue and orange parts of Fig. 2 show the *additional* work saved by using our symbolic stateful exploration on top of using macro-step semantics and POR. To the best of our knowledge, our work is the first to target redundancies in overlapping transitions, and the first to use value summaries for distributed systems exploration. Because distributed systems often involve having several instances of processes that exhibit the same or similar behaviors, we expect a high number of transitions to have shared behaviors that we can handle efficiently.
Beyond systematic exploration. The ability to perform scalable stateful exploration is a useful utility. Beyond systematic exploration, it also allows discovery of invariants through computation of fixed points (§5.4) and the use of abstractions (§6.1). Discovering invariants on small instances of distributed protocols is a crucial component in some recent approaches such as I4 [Ma et al. 2019] and DistAI [Yao et al. 2021] that generalize results from small instances to large or arbitrary-sized instances. Our symbolic explorer could be integrated with such techniques in future work.

3 ADAPTING MACRO-STEP SEMANTICS FOR P

We now describe an adaptation of macro-step semantics for P, which provides a baseline for our symbolic explorer. This semantics treats transitions starting from and including a receive of a message at a state machine up until but not including the next receive of a message at that machine as a single *atomic step*, i.e., only receives of messages at different machines are interleaved. As in actor systems [Agha et al. 1997], the interleaving semantics is *equivalent* to its macro-step semantics. All executions of a P program under interleaving semantics (as seen in prior work [Desai et al. 2014, 2013a, 2018, 2015]) also have a Mazurkiewicz-equivalent execution in the macro-step semantics.

3.1 Notation for P Semantics

Machine. Let $\mathcal{A}$ represent the set of names of all machine types. Let $\mathcal{I}$ represent the set of all the machine identifiers referencing dynamic instances of machine types in $\mathcal{A}$. Let $\mathcal{S}_{id}$ represent the set of local states for a machine with reference identifier $id \in \mathcal{I}$.
Message. Let $\mathcal{E}$ be the set of names of events and $\mathcal{V}$ be the set of all possible payload values that may accompany any event. Let $\mathcal{M}$ represent the set of all possible messages. Each message is a tuple $(src, ev, v, tgt, mid) \in (\mathcal{I} \times \mathcal{E} \times \mathcal{V} \times \mathcal{I} \times \mathbb{N})$ where src is the source or sender of the message, ev is the event being sent, v is the associated payload value with the event, tgt is the target or intended recipient of the message, and mid is an unique identifier associated with each message. For any $m \in \mathcal{M}$, we refer to its components as $m.src$, $m.ev$, $m.v$, $m.tgt$, and $m.mid$ respectively.
Event Handlers. Recall that each state machine declaration in P has a set of event handlers per state. Each event handler is responsible for processing messages with the associated event type (e.g., in Fig. 1, line 26 defines an event handler for messages with event type `eWorkItem`). All messages with the same event type that are received in the same target machine state thus *have the same event handler.* Let $\mathcal{H}$ be the set of all event handlers in the given P program. Let $H : \mathcal{S}_{id} \times \mathcal{E} \rightarrow \mathcal{H}$ be a partial function that given the current state of a machine and an event type maps it to the event-handler declared in the P program that gets executed to handle the received event. An event handler $h \in \mathcal{H}$ is a function that takes the message received as an argument and returns a sequence of P statements. We refer the readers to [Desai et al. 2013a] for more details.

$$\textsc{Schedule Step}\quad \frac{\texttt{enabled}(m,c)^{(c1)} \qquad c'' = (L_c, B_c \setminus \{m\}, SO_c)^{(c2)} \qquad H[L_c[m.tgt], m.ev] = h^{(c3)} \qquad m.tgt \vdash (h(m), c'') \rightarrow^* (\textbf{skip}, c')^{(c4)}}{H \vdash c \overset{m}{\Longrightarrow} c''}$$

$$\textsc{Sequence}\quad \frac{id \vdash (S_0, c) \rightarrow^* (\textbf{skip}, c') \quad id \vdash (S_1, c') \rightarrow^* (\textbf{skip}, c'')}{id \vdash (S_0; S_1, c) \rightarrow (\textbf{skip}, c'')}$$

$$\textsc{Assign-Var}\quad \frac{L_c[id] \vdash e \rightarrow v \quad c' = (L_c[id][x \mapsto v], B_c, SO_c)}{id \vdash (x := e, c) \rightarrow (\textbf{skip}, c')}$$

$$\textsc{Send}\quad \frac{m = (id, ev, v, tid, \texttt{fresh}(mid, c))^{(c5)} \quad c' = (L_c, B_c \cup \{m\}, SO_{c'})^{(c6)} \quad SO_{c'} = SO_c \cup \{(m', m) \mid m'.src = id \wedge m' \in B_c\}^{(c7)}}{id \vdash (\textbf{send}(tid, ev, v), c) \rightarrow (\textbf{skip}, c')}$$

$$\textsc{If-Then}\quad \frac{L_c[id] \vdash e \downarrow \textbf{true}}{id \vdash (\textbf{if } e \textbf{ then } S_0 \textbf{ else } S_1, c) \rightarrow (S_0, c)}$$

$$\textsc{If-Else}\quad \frac{L_c[id] \vdash e \downarrow \textbf{false}}{id \vdash (\textbf{if } e \textbf{ then } S_0 \textbf{ else } S_1, c) \rightarrow (S_1, c)}$$

$$\textsc{Choose}\quad \frac{v \in V}{L_c[id] \vdash \textbf{choose } V \downarrow v}$$

Fig. 3. Macro-Step Semantics for P programs

Configurations. A configuration of a P program is represented as a tuple (L, B, SO): **(1)** first component L is a partial map from $id : \mathcal{I}$ to $\mathcal{S}_{id}$. If $id \in dom(L)$, then $L[id]$ is the state of the machine referenced by id; **(2)** second component B is the global buffer represented set of messages that have been sent during the execution but have not been received by the target machine; **(3)** third component SO ($\subseteq \mathcal{M}_u \times \mathcal{M}_u$) is a relation used to capture the *send-order* of instances of messages sent during the execution and ensure the FIFO buffer semantics in P (unlike the *bag* semantics allowed in typical actor systems). If $(m_1, m_2) \in SO$, then m_1 was sent before m_2. Let C represent the set of all configurations. The initial configuration c_0 of a P program is defined as $(L_{c_0}, B_{c_0}, SO_{c_0})$ where L_{c_0} maps the set of machine instances at the start of the program to their local state and B_{c_0} contains only messages from the same sender, i.e., $\forall m, m' \in B_{c_0}.\ m.src = m'.src$.

3.2 Macro-step Semantics for P

A P program steps from one configuration to another via a labeled-transition, each of which receives a message. A macro step (or schedule step) is represented as $c \overset{m}{\Longrightarrow} c'$, where label m indicates the message received during the transition that takes the system from configuration c to c' by executing the appropriate event handler. The rules for macro-step semantics are shown in Fig. 3.

Schedule Step. The first rule presents the macro-step transition, given a mapping H from machine states and messages to event handlers. A step $c \overset{m}{\Longrightarrow} c'$ is only possible whenever the transition that receives message m is *enabled* in c. Formally, this condition (labeled (c1)) is denoted by predicate $\texttt{enabled}(c, m)$ defined as : $m \in B_c \wedge \forall m' \in B_c.m' \neq m \wedge m'.tgt = m.tgt \Rightarrow (m', m) \notin SO_c$.

In P programs, because of FIFO buffer semantics, messages must be received in an order consistent with the causal relation of their sendings. In particular, **(1)** any message m' sent in response to a receive of another message m must be received after m is received, and **(2)** any two messages sent to the same target must be received in the order they were sent, respecting the send-order relation. The first condition is handled by adding messages to B_c immediately upon sending them and removing them immediately upon receiving them, as noted in Fig. 3. The second condition is addressed by the $\texttt{enabled}(c, m)$ predicate for the P programs, which ensures that for any message m that can be received by $m.tgt$, all other messages m' with the same target are sent after m.

Executing the macro step involves removing the message m from buffer B_c (c2), and executing the corresponding event handler h in the target machine (c3) on message m. The condition (c4) represents execution of the sequence of statements $h(m)$ at the target machine $m.tgt$, which may only change the local state of the target machine and the global buffer (if a **send** statement is executed). $\rightarrow^*$ denotes the reflexive transitive closure of $\rightarrow$ transitions (used for statements).

Statements. In Fig. 3, the next five rules present semantics of some of the statements in P handlers that have non-trivial semantics when lifted for symbolic exploration (revisited in §5.1). The

Sequence, Assign-Var, If-Else, and If-Then rules are straightforward and present the common semantics for sequence, assignment, and if-then-else statements, respectively. Here, $\downarrow$ denotes the big-step semantics for evaluation of an expression e to a value v, and $\top$, $\bot$ denote the Boolean values true, false, respectively. The Send rule present the semantics of asynchronous send statement `send t,ev,v`, which sends a message m (c5) with event ev, payload v to target machine t, and a fresh message id. It adds message m to the global buffer (c6) and updates the send-order relation (c7) so that all previously-sent messages in the buffer that came from the same source machine are related to the new message m. SO_c is updated so if B_c is nonempty, there is at least one message m for which $\texttt{enabled}(c, m)$ holds. Finally, the Choose rule presents the semantics of the **choose** operation in P used for introducing data non-determinism in the programs.

Execution. An *execution* of a P program is a sequence of macro steps for it $c_0 \stackrel{m_0}{\Longrightarrow} \cdots \stackrel{m_{n-1}}{\Longrightarrow} c_n \stackrel{m_n}{\Longrightarrow} \cdots$. They are *partial* (vs. *full*) when they are finite and the last configuration can take a macro step.

4 SYMBOLIC REPRESENTATION OF CONFIGURATIONS: VALUE SUMMARIES

In this section, we describe details of our proposed symbolic representation of configurations for distributed systems. These are critical for efficient symbolic exploration (§5) and could be potentially useful in other verification techniques for distributed systems. The technical definitions are summarized in a cheat-sheet shown in Fig. 4, with explanations in the related subsections.

4.1 Identifying Overlapping Transitions

Recall from our motivating example (§2.4) that a novel element of our approach is to identify redundancies due to *overlapping transitions*, e.g., the transitions from c_4 in Fig. 2. A *transition* corresponds to a schedule step in the concrete semantics (Fig 3), that receives a message at a target machine, executes the corresponding event handler, and moves the system to the next configuration. Fig. 4(a) shows the technical definition for overlapping transitions. Informally, two transitions overlap if they operate on the same machine (tgt), and at least one of their update computations can be performed by the same function (h), on the same local variables ($\mathcal{S}_o$) and payloads (v_o). Other update computations may also be performed, constituting the transitions' non-overlapping parts.

Identifying functionally equivalent computations in arbitrary transitions may require additional semantic analyses in general. Instead, in this work, we follow a simple syntactic approach that uses having the same event handler as a proxy. In other words, when two transitions correspond to executing the same event handler at the same target machine, we consider them as *overlapping transitions* and aim to ensure that their overlapping computations are executed only once.

4.2 Requirements on Symbolic Representations of Configurations

We now consider requirements on our symbolic representations that will help identify overlapping transitions on-the-fly during symbolic exploration and achieve savings by avoiding redundancies. We illustrate them by revisiting the motivating example in Fig. 2.

R1. Fine-grained Component-level Representation of Configurations and Messages. We require a *decomposed* representation of configurations, which maintains a separate representation for each configuration component, so that we can identify when transitions correspond to the same event handler. Recall that in P, event handlers are identified by the local state and event type; furthermore, the possible event types and payloads for the next transition are determined by the global message buffer. Thus, the decomposed representation should be fine-grained enough to represent the local state of a machine instance, and the event types and payloads of messages separately. For example, c_4 in Fig. 2 has messages for $r0, r1, r2$ in its global buffer. These messages all have event type `eRegisterWorker` and are all sent to the `Registry` instance – this can be detected only with a fine-grained representation of configurations and message components.

(a) Overlapping transitions

Let $c_1 \xrightarrow{m_1} c_1'$, $c_2 \xrightarrow{m_2} c_2'$ be transitions and tgt, $\mathcal{S}_o$, v_o be such that the following conditions hold:

Same target, tgt:	$tgt = m_1.tgt = m_2.tgt$
Local state overlap, $\mathcal{S}_o$:	$\forall x.\mathcal{S}_o[x] = c_1.L[tgt].x \Leftrightarrow c_1.L[tgt].x = c_2.L[tgt].x$
Payload overlap, v_o:	$v_o[p] = m_1.v.p \Leftrightarrow m_1.v.p = m_2.v.p$

For $i \in \{1, 2\}$, let $\{x_1, \ldots, x_n\}$ be the domain of local variables $c_i.L[tgt]$,
and let $\{f_i(x_1), \ldots f_i(x_n)\}$ be the update functions that compute updates to these local variables,
i.e., for $1 \leq j \leq n$, $c_i.L[tgt][x_j] = f_i(x_j)(c_i.L[tgt], m_i.v)$.
The transitions c_1 and c_2 are *overlapping* if there exists an x in the domain of $\mathcal{S}_o$, such that there exists a function h with $h(\mathcal{S}_o, v_o) = f_1(x)(c_1.L[tgt], m_i.v) = f_2(x)(c_2.L[tgt], m_i.v)$, and there exists $i \in \{1, 2\}$ such that $h(\mathcal{S}_o, v_o) \neq c_i.L[tgt][x]$ (i.e., x is changed by h in c_1 or c_2).

(b) Primitive value summaries: Invariant properties for guarded values (g, v)

Non-overlapping guard	$\forall (g', v') \in pvs.v \neq v' \Rightarrow g \wedge g' \Rightarrow \bot$
Unique value	$\forall (g', v') \in pvs.g \not\equiv g' \Rightarrow v \not\equiv \neg v'$
Non-vacuous	$g \not\Rightarrow \bot$

(c) Composite value summary representations

Tuple value summary	$(x_0, \ldots, x_n)$	where all x_i are value summaries with same domain
List value summary	(s, ls)	with integer value summary s, value summary list ls
Map value summary	$mp : \tau \rightarrow \tau'$	where every element in τ' is a value summary

(d) Core operations on primitive/composite value summaries

Removing spurious guards: $rmf(pvs) = \{(g, v) \mid (g, v) \in pvs \wedge g \neq \bot\}$

Domain operation:
$$D(vs) = \begin{cases} \bigvee\{g \mid (g, v) \in vs\} & vs \text{ is a primitive value summary} \\ D(s) & vs = (s, ls), vs \text{ is a list value summary} \\ \bigvee\{D(vs_i) \mid i \in \{0, \ldots, n\}\} & vs = (vs_0, \ldots, vs_n) \\ \bigvee\{D(vs') \mid \exists k.vs[k] = vs'\} & vs \text{ is a map} \end{cases}$$

Merging operation: $M(\{vs_0, \ldots, vs_k\}) =$ if $k = 1$ then $M(vs_0, vs_1)$ else $M(vs_0, M(vs_1, \ldots, vs_k))$ and

$$M(vs_0, vs_1) = \begin{cases} vs_0 \cup vs_1 & vs_0, vs_1 \text{ are primitive value summaries} \\ (M(vs_0^0, vs_1^0), \ldots, M(vs_0^n, vs_1^n)) & vs_i = (vs_i^0, \ldots, vs_i^n) \text{ (tuple or list value summary)} \\ vs_i & \text{where } vs_j = [] \text{ for } j \neq i, j, i \in \{0, 1\} \\ M(x_0, x_1) :: M(xs_0, xs_1) & vs_0 = x_0 :: xs_0 \text{ and } vs_1 = x_1 :: xs_1, \text{ where } :: \text{ is cons} \\ mergeMap(mp_0, mp_1) & vs \text{ is a map} \end{cases}$$

where $D(vs_0) \wedge D(vs_1) = \bot$ (i.e., non-overlapping domains) and $mergeMap(mp_0, mp_1)$ is

$$\{k \mapsto vs \mid vs = M(mp_0[k], mp_1[k]) \vee \exists i, j \in \{0, 1\}.vs = mp_i[k] \wedge k \notin keys(mp_j)\}$$

Restriction operation:
$$(vs|\phi) = \begin{cases} \bigcup rmf(\{(g \wedge \phi, v) \mid \exists (g, v) \in vs\}) & vs \text{ is a primitive value summary} \\ ((vs_0|\phi), \ldots, (vs_n|\phi)) & vs = (vs_0, \ldots, vs_n) \\ ((s|\phi), (ls|\phi)) & vs = (s, ls) \\ \{k \mapsto (vs'|\phi) \mid vs[k] = vs'\} & vs \text{ is a map} \end{cases}$$

Getting guards for values: $\exists g.(g, v) \in pvs \Rightarrow getGuardFor(pvs, v) = g$

Updating under a guard: $updateUnderG(vs, vs', g) = M((vs|\neg g), (vs'|g))$

Fig. 4. Technical cheat sheet for overlapping transitions and value summaries.

R2. Canonicity of Symbolic Representations. We need to identify when configuration and message components are equivalent. Canonical representations – i.e., where equivalent sets of configurations/messages have identical representations – make such identification quick and cheap,

and allows us to efficiently detect when transitions have the same event handlers. We also require that each configuration update maintain canonicity, preventing the exploration from revisiting redundant configurations at the same execution depth. For example, in Fig. 2, canonicity ensures that c_1 and c_2 have the same representation during exploration, making it easy to detect and avoid redundant exploration of the blue subtree under c_2. (Note that POR techniques cannot detect the equivalence of c_1 and c_2 because they occur under different interleavings of *dependent* transitions.)
R3. Association of Nondeterministic Choices with Values. Finally, we require that symbolic representations implicitly implement a function from *sequences of nondeterministic choices* taken during exploration to specific values, in order to understand the explored behaviors and to produce counterexample traces. For systematic exploration, these sequences should include all scheduling choices taken during execution, as well as any choices due to control-flow or data nondeterminism.

4.3 Value Summaries for Configurations of Distributed Systems

To satisfy requirements **R1-R3**, we use a symbolic representation inspired by MultiSE [Sen et al. 2015], a systematic exploration framework for sequential programs. In MultiSE, value summaries represent sets of concrete values for individual variables under different guards, where guards capture control-flow and input nondeterminism. Importantly, value summaries provide a canonical representation of the program state, which other symbolic representations, such as those based on formulas in solvers for Satisfiability Modulo Theories [Barrett et al. 2009] do not. In this work, we extend guards in value summaries to capture *scheduling choices* also. We also propose *composite* value summaries to represent components of a distributed system in a fine-grained manner. To the best of our knowledge, we are the first to use MultiSE-style value summaries for distributed systems.

4.3.1 Guards and Scheduling Choices. Value summaries are based on guard-value pairs, called *guarded values*, where a *guard* is a propositional logic formula over Boolean-valued guard variables, and all guards in a summary are non-overlapping [Sen et al. 2015]. We propose the use of *SCI Guards*, which represent choices in the presence of **S**cheduling, **C**ontrol, and **I**nput nondeterminism. As we will see in §5.1, we introduce fresh variables in guard formulas to symbolically encode the *set* of possible choices for any nondeterministic choice (both input and scheduling) made during exploration. This associates a *unique* guard with each possible nondeterministic choice, leading to unique guards for each sequence of choices arising during exploration. Thus, this representation satisfies **R3**.

4.3.2 Primitive Value Summaries. A *primitive value summary pvs* is a set of guarded values. Each pair (g, v) represents a fact that the value v is taken under the guard g We define three invariant properties for each guarded value (g, v) in a value summary *pvs*, shown in Fig. 4(b). Two value summaries are *equivalent* (denoted with $\equiv$) iff they have equivalent values under all the guards.
Tunable Value Summaries. Guards that capture nondeterministic choices are represented symbolically; however, the representation for the value components of each guarded value, as in MultiSE [Sen et al. 2015], is not restricted, as long as it supports a check for equivalence. (Though for best performance, canonical representations should be used.) In our work, we use *concrete* as well as *abstract* values. We assume concrete values when discussing our symbolic explorer (§5), to focus on the symbolic guard-based operations needed for efficient exploration, but discuss how to use a simple predicate abstraction [Graf and Saïdi 1997] to represent values in §6. Thus, our value summaries are tunable, where the value representation can range from concrete to symbolic to abstractdomains.

4.3.3 Composite Value Summaries. We use primitive value summaries as building blocks to propose composite value summaries for representing composite data structures (e.g., tuples, lists). Composite

value summaries are also a novel contribution of our work. They enable fine-grained symbolic representations of configurations and messages in distributed systems, satisfying requirement **R1**. A composite value summary is constructed out of other value summaries (primitive or composite). We show them for tuples, lists, and maps in Fig. 4(c) – these are needed to represent P programs.

Example 4.1. Consider the messages $(\texttt{worker}_i, \texttt{eRegisterWorker}, i, \texttt{reg})$, $i \in \{0,1,2\}$ sent by `Worker` machines (Fig. 1). The following value summary captures these under distinct guards where g_0, g_1, g_2 are formulas with pairwise empty conjunctions, and id_0, id_1, id_2 are unique message ids:

$$(\{(g_i, \texttt{worker}_i) \mid i \in \{0,1,2\}\}, \{(\top, \texttt{eRegisterWorker})\}, \{(g_i, i) \mid i \in \{0,1,2\}\}, \{(\top, \texttt{reg})\}, \{(g_i, id_i) \mid i \in \{0,1,2\}\})$$

Note the lack of redundancy in representing the three messages' types – `eRegisterWorker` occurs only once in the value summary.

4.3.4 Core Operations on Value Summaries. We now discuss core operations on value summaries, shown in Fig. 4(d), that will be used for symbolic exploration. In particular, the lifted semantics (§5.1) used by the exploration algorithm in §5 uses the *domain* operation to get the guard under which a configuration should be modified, and the *merge* and *restrict* operation to update only the part of the configuration that should be modified by a transition.

Domain. The domain $D(vs)$ of a value summary vs (symbolically) is a Boolean formula that represents the set of guards under which it has a value. In particular, note that the domain of a list value summary (s, ls) is recovered by taking the domain of its size component s. Updates to a list value summary must maintain the invariant that each element of ls has a domain that implies $D(s)$.

Merging. The result of merging of two value summaries vs_0, vs_1 is another value summary, defined under the condition $D(vs_0) \wedge D(vs_1) = \bot$ (i.e., non-overlapping domains). The result of merging a set of value summaries is defined similarly.

Restriction. Restricting a value summary vs's domain by a guard ϕ, denoted $(vs|\phi)$, yields a value summary with domain $D(vs) \wedge \phi$, which represents only the values in vs under guard ϕ.

Getting Guards for Values. The function *getGuardFor* takes a primitive value summary *pvs* and a value v and returns an SCI Guard g such that $(g, v) \in pvs$.

Updating under a Guard. Updating a value summary vs under an SCI Guard g allows us to model assignments under the nondeterministic choices captured by g. Such an update can be easily defined by using merging (M): $updateUnderG(vs, vs', g) = M((vs|\neg g), (vs'|g))$. Note that the restriction operations make the domains of the arguments to the merge non-overlapping.

4.4 Canonicity of Value Summaries

We now state an important theorem for value summaries that is maintained by the operations on value summaries defined above(a proof is provided in Appendix A).

Theorem 4.2 (Canonicity of Value Summaries). *If a canonical representation is used for propositional formulas and for value components of guarded values, then all value summaries are such that, for any two value summaries vs_0, vs_1, $vs_0 \equiv vs_1$ iff $vs_0 = vs_1$.*

Here $\equiv$ denotes semantic equivalence and $=$ denotes syntactic equality. By this theorem, value summary representations satisfy requirement **R2**. Like MultiSE [Sen et al. 2015], we represent guards with Binary Decision Diagrams (BDDs) [Bryant 1986]. BDDs are used in a very limited way here – only to represent the guards in value summaries. In general, the size of state representations is a scalability concern, but the regular structure in practical programs of interest can help avoid blowup. As demonstrated in §8, our tool successfully handles many standard and industry-strength benchmark programs.

5 SYMBOLIC STATEFUL EXPLORATION USING VALUE SUMMARIES

In this section, we first describe how to lift P values and semantics to value summaries, and then present our novel symbolic explorer that operates over the lifted P program.

Tuples	$\ell(x_0, \ldots, x_n) = (\ell(x_0), \ldots, \ell(x_n))$	**Maps**	$\ell(m) = \{x \mapsto \ell(m[x]) \mid x \in Dom(m)\}$
Lists	$\ell([]) = (\ell(0), [])$		$\ell(x :: xs) = (\ell(\text{len}(x :: xs)), \ell(x) :: \ell(xs))$

Fig. 5. Lifting of data structures

5.1 Lifting Values and Functions to Value Summaries

Let $VS(\tau)$ represent the value summary representation for any value of type τ. Let $\ell : \tau \to VS(\tau)$ denote the lifting function[1] and *Vals* denote the function that determines the set of values represented by a lifted value summary under any guard. *Vals* and ℓ meet the following requirements:

- For any v, $Vals(\ell(v)) = \{v\}$
- For any function f, $\ell(f)$ is deterministic and $Vals(\ell(f)(\ell(args))) = \{r \mid f(args) \text{ can return } r\}$

Functions ℓ and *Vals* can be seen as restricted forms of an abstraction and concretization function, respectively, where abstraction does not introduce any imprecision.

Lifting Data Structures. The P language contains several types that may be used in event and state handlers: primitive types, tuples, lists, sets, and maps [P-GitHub 2023]. For these types other than sets, we directly use their corresponding value summaries to represent them; for sets we use list value summaries, e.g., by representing sets as sorted lists with no duplicate elements. Formally, we have $\ell(p) = \{(\top, p)\}$ for primitive value p and $Vals(pvs) = \{v \mid \exists g.(g, v) \in pvs\}$ for primitive value summary *pvs*. Fig. 5 shows how to lift concrete composite data structures by recursively applying ℓ.

Lifting Functions. A function $f : \tau \to \tau'$ over primitive types can be lifted to $\ell(f) : VS(\tau) \to VS(\tau')$, which iterates over values in the provided primitive value summary vs and combines guarded values with the same values into a single guarded value as follows: $\lambda vs.\{(\bigvee G, v') \mid G = \{g \mid (g, v) \in vs \wedge f(v) = v'\}\}$. In effect, evaluating $f(x)$ requires evaluating f on each concrete value represented by x. For composite value summaries, lifting is more involved, since for efficiency, we want to avoid having to enumerate all the concrete values represented by a composite value summary. As an example, the equality check $v_0 = v_1$, which returns a value $b \in \{\top, \bot\}$, can be lifted to $=_\ell$, which takes two value summaries and returns a Boolean value summary $VS(\{\top, \bot\})$. For composite arguments, such a lifting of $=_\ell$ should recursively evaluate $=_\ell$ on corresponding components of the value summaries and return the conjunction of the results.

Example 5.1. Consider checking the equality of the following tuple value summaries, where X, Y are finite subsets of $\mathbb{N}$: $(\{(g_i, x_i) \mid i \in X\}, \{(\top, y_0)\}), \quad (\{(\top, x_0)\}, \{(g_i, y_i) \mid i \in Y\})$. We first recursively check equality of the first elements, yielding result $\{(g_0, \top), (\neg g_0, \bot)\}$. This check involves $|X|$ comparisons, comparing each x_i with x_0. We then do the same for the second elements, yielding the same in $|Y|$ comparisons. The equality check gives result $\{(g_0, \top), (\neg g_0, \bot)\}$. Note that if we used a primitive value summary representation of these tuples rather than composite value summaries, we would need to perform at least $|X| \times |Y|$ comparisons, since we would have to compare each pair in the first summary to each in the second.

When implementing lifted data structures and functions, it is important for the sake of efficiency to expose and take advantage of the decomposed representation of composite value summaries as much as possible. For example, the list value summary representation maintains and exposes the size of the list, which allows efficient implementations of the operations that get the domain of the value summary or the size of the list. More details about the lifting of operations on composite value summaries are provided in Appendix B.

5.2 Lifted Semantics for P Programs

Fig. 6 shows the result of lifting the rules for P semantics from Fig. 3. See Appendix C for the full set of rules. While we use the same notation for configurations, messages, handlers, etc., note that

[1]We overload the notation $\ell(\cdot)$ to denote lifting of both values and functions.

$$\textsc{Schedule Step}\quad \frac{\begin{array}{c} choices = \ell(\lambda x.\{y \mid \texttt{enabled}(y,x)\})(c) \qquad g = \phi \wedge \bigvee\{D(choice) \mid choice \in choices\}^{(c8)} \qquad g, \{\} \vdash \textbf{choose}\ choices \downarrow m^{(c9)} \\ c'' = (L_c, B_c \setminus \{m\}, SO_c)^{(c10)} \qquad \{g_0, \ldots, g_n\} = \{\phi \mid \exists t, ev.\phi = getGuardFor(m.tgt, t) \wedge getGuardFor(m.ev, ev)\}^{(c11)} \\ \forall 0 \le i \le n.m_i = (m|g_i) \wedge H[L_c[m.tgt_i], m_i.ev] = h_i^{(c12)} \\ g_0, m_0.tgt \vdash (h_0(m_0), c'') \rightarrow^* (\textbf{skip}, c'_0)\ \ldots\ g_n, m_n.tgt \vdash (h_n(m_n), c'_{n-1}) \rightarrow^* (\textbf{skip}, c'_n)^{(c13)} \qquad c' = (c'_n|D(m))^{(c14)} \end{array}}{H, \phi \vdash c \overset{m}{\Longrightarrow} c'}$$

$$\textsc{Assign-Var}\quad \frac{\begin{array}{c} L_c[id] \vdash e \downarrow v \\ c' = (L_c[id][x \mapsto updateUnderG(x, v, g)], B_c, SO_c) \end{array}}{g, id \vdash (x := e, c) \rightarrow (\textbf{skip}, c')}$$

$$\textsc{Send}\quad \frac{\begin{array}{c} m = ((id, ev, v, tid, \texttt{fresh}(mid, c))|g)^{(15)} \qquad c' = (L_c, B_c \cup \{m\}, SO_{c'})^{(16)} \\ SO_{c'} = SO_c \cup \ell(\lambda x, y.\{(m', x) \mid m'.src = x.id \wedge m' \in y\})(m, (B_c|g))^{(c17)} \end{array}}{g, id \vdash (\textbf{send}(tid, ev, v), c) \rightarrow (\textbf{skip}, c')}$$

$$\textsc{If}\quad \frac{\begin{array}{c} L_c[id], g \vdash e \downarrow v \qquad g_0 = g \wedge getGuardFor(v, \top) \\ g_1 = g \wedge getGuardFor(v, \bot) \qquad g_0, id \vdash (S_0, c) \rightarrow^* (\textbf{skip}, c') \\ g_1, id \vdash (S_1, c') \rightarrow^* (\textbf{skip}, c'') \end{array}}{g, id \vdash (\textbf{if}\ e\ \textbf{then}\ S_0\ \textbf{else}\ S_1, c) \rightarrow (\textbf{skip}, c'')}$$

$$\textsc{Choose}\quad \frac{\begin{array}{c} B = getFreshGuardVariables(\lceil \log(|V|) \rceil)^{(c18)} \\ V = \{v_1, \ldots, v_{|V|}\} \\ v = M(\{(v_i|encodeUsing(i, B, |V|))\}_{i=1..|V|})^{(c19)} \end{array}}{g, L_c[id] \vdash \textbf{choose}\ V \downarrow (v|g)}$$

Fig. 6. Lifted semantics for P programs where $\setminus$ and $\cup$ are lifted versions of their corresponding operations.

these have all been lifted to value summaries in Fig. 6. In particular, the set of pending messages B_c and the send order SO_c are *also* value summaries – this enables processing overlapping transitions together when they have the same event handler. The map H has also been lifted to map value summary arguments to a lifted handler. Because these have been lifted to value summaries, we meet requirements **R1-R3** (§4.2) for our configuration and message representations.

Guard in a Context. For all rules, there is a guard g (or ϕ in Schedule Step) in the context – this keeps track of the *current* SCI Guard in the execution so far, and *all* lifted operations occur under g (or ϕ). Recall that the SCI Guard characterizes all the nondeterministic choices made in the execution, thus g determines a subset of the paths in the execution tree and restricts which configurations in the frontier get updated. More formally, if $g, id \vdash (S, c) \rightarrow (S', c')$, we have that $(c|\neg g) = (c'|\neg g)$ (and analogously for the $\overset{m}{\Longrightarrow}$ step), i.e., the configurations under $\neg g$ are not updated. This can be seen in the Send rule's use of restriction and in the Assign-Var rule's use of *updateUnderG* (both defined in Fig. 4(d)). Similarly, the operations also occur under the domain of the current configuration $D(c)$, so that if $g, id \vdash (S, c) \rightarrow (S', c')$, we have that $D(c')$ implies $D(c)$ (and analogously for the $\overset{m}{\Longrightarrow}$ step).

Lifted Event Handler. For any configuration, all enabled messages that share the same event handler are handled by a *single* invocation of a lifted version of the event handler, which is executed on a value summary that captures *all* their payloads. This results in fewer handler invocations and less redundant computation than handling them separately. As we will show in §8.2, there are many opportunities in distributed systems to handle messages together this way.

Choose. The Choose procedure implements the semantics for the **choose** operation, which takes a set V of value summaries. It generates a *single* value summary that captures *all* the values $v_i \in V$. Specifically, it introduces *fresh* Boolean guard variables B (c18) to construct a primitive value summary v (c19), such that $Vals(v) = V$ and each guarded value in v uses only variables in B. This effectively encodes all the possible nondeterministic choices in V. In particular, the function $encodeUsing(i, B, size)$ is defined for $1 \le i \le size$ and behaves as follows: for $i < size$, it gives the binary encoding $bin(i-1, B)$ for $i-1$ using the guard variables in B, and for $i = size$, it returns $\bigwedge\{bin(k-1, B) \mid size \le k \le |B|^2\}$. The effect is that $encodeUsing$ uses B to create $size$ symbolic

partitions, each of which becomes a guard in v. However, this **choose** operation as described leaves some "left over" values under the guard $(\bigvee\{D(v_i) \mid v_i \in V\} \wedge \neg D(vs))$ (when it is not false). Such values should be eliminated by restricting subsequent operations by $D(vs)$ as leaving "left over" values can result in re-exploration of choices.

Schedule Step. Compared with Fig. 3, the Schedule Step rule looks the most different, since it now must advance *all* configurations in the frontier set rather than a single frontier configuration. It finds the guard under which there are enabled messages to schedule (c8), and then uses Choose to make a nondeterministic choice over enabled messages (c9), since scheduling nondeterminism is handled in the same way as input and control-flow nondeterminism. The enabled message value summary m here represents the *set of messages* that can be received next for all configurations in the frontier set. Put another way, consider the case in the original semantics where we have a set of messages *Mess* and configuration value *config*, where each $mess \in Mess$ is enabled in *config* (see (c1) in Fig. 3). For any configuration value summary c in the lifted semantics with $config \in Vals(c)$, if we can take a Schedule Step from c, then $Mess \subseteq Vals(m)$, where m is as in (c9).

Removing $\{m\}$ from the pending messages set B_c (c10) involves removing all these messages *only under their corresponding domains*, i.e., the domains introduced by the **choose** operation. After this removal, which yields the (intermediate) configuration c'', the guards change accordingly: for any message value $mess \in Vals(m)$, $mess \notin Values((B_{c''}|D(m)))$. Any "left over" enabled messages in $(B_{c''}|\neg D(m))$ are eliminated in (c14).

For each distinct target and event in m under guard g_i (c11), there is a unique (lifted) event handler h_i (c12). The number of distinct targets and events in m may be less than the number of distinct messages in m – in such cases, we process these overlapping messages with the same h_i. In order to process all messages, Schedule Step runs all such event handlers under each guard g_i (c13), and then finally restricts the resulting configuration to the domain $D(m)$ under which a message was processed (c14). Note that $m_i.v$ is the payload for *all* messages in m_i, so each lifted handler runs *only once* per distinct target and event pair in m, and on a value summary that represents potentially *multiple* message payloads. This achieves the goal in requirement **R1**: to identify when transitions correspond to the same event handler and handle them together.

Assign-Var. The Assign-Var rule is as in Fig. 3, but with the the map $L_c[id]$ lifted to be over value summaries (as in Fig. 5). The update of the map assignment to x correspondingly becomes an update of the value *only under the guard* g, via *updateUnderG*.

Send. The Send rule is also a fairly straightforward lifting of the corresponding rule from Fig. 3. Recall that the set of pending messages B_c and the send orders $SO_c, SO_{c'}$ have been lifted to set value summaries. The update to the set of pending messages B_c (c16) happens only under the domain for message m, which has been restricted by g (c15). The update to the send-order relation (c17) is also done similarly, adding messages only under g.

If. The If rule replaces both of the branching rules from Fig. 3. It executes the **then** branch under the guard g_0 for which condition e evaluates to $\top$, and the **else** branch under the guard g_1 for which it evaluates to $\bot$, suitably updating the value summaries in a single step of the semantics.

Example 5.2. Consider running Schedule Step for the example in Fig. 1 at the root of the execution tree in Fig. 2. Suppose no POR is applied (for simple illustration), so that we must consider all interleavings of `eWorkItem` messages. The enabled message value summaries at this point are the following three summaries, for $i \in \{0, 1, 2\}$ for some message ids $\texttt{id}_i$:

$$(\{(\top, \texttt{Server})\}, \{(\top, \texttt{eWorkItem})\}, \{(\top, \texttt{reg})\}, \{(\top, \texttt{worker}_i)\}, \{(\top, \texttt{id}_i\})$$

Each summary is a 5-tuple of value summaries for the message sender, event being sent, payload for the event, message target, and unique identifier of the message. For this application of Schedule

Algorithm 1 Explicit exploration of one execution.

```
1: procedure Explore(P)
2:     (c, H) ← (InitialConfig(P), GetHandlers(P))
3:     while B_c ≠ ∅ do
4:         c ← ScheduleStep(H, c)
```

Algorithm 2 Symbolic exploration of all executions.

```
1: procedure SymExplore(P_ℓ)
2:     (vs_c, H, prev) ← (InitialConfig(P_ℓ), GetHandlers(P_ℓ), {})
3:     while bufferNotEmpty(c) do
4:         Optional: prev ← vs_c :: prev
5:         vs_c ← ScheduleStep(H, notFixedPoint, vs_c)
6:         vs_c ← (vs_c|¬checkFixed(prev, vs_c))        ▷ checkFixed returns ⊥ unless line 4 enabled
```

Step, $g = \top$ (c8). The resulting value summary m with all nondeterministic choices (c9) follows:

$$(\{(\top, \texttt{Server})\}, \{(\top, \texttt{eWorkItem})\}, \{(\top, \texttt{reg})\}, \{(f_i, \texttt{worker}_i)\}_{i=0..2}, \{(f_i, \texttt{id}_i)\}_{i=0..2})$$

Here, Choice has introduced fresh guard variables b_0, b_1, b_2, and guard formulas $f_0 = b_0 \wedge b_1$, $f_1 = b_0 \wedge \neg b_1$, and $f_2 = \neg b_0$ to encode the three possible choices. Note that there are three distinct target and event pairs in m, $(\texttt{worker}_i, \texttt{eWorkItem})$ for $i \in \{0, 1, 2\}$, which each occur under their respective guard f_i (c11). Thus, a lifted event handler for eWorkItem (c12) for each $\texttt{worker}_i$ is run under the guard f_i on the message payload reg (c13), yielding a value summary that symbolically represents *all* configurations at depth 1 in Fig. 2.

Theorem 5.3 (Efficiency of Schedule Step). *For any program P with lifted program P_ℓ, consider a frontier set represented by a configuration value summary c that can step to configuration c' via the Schedule Step. Each lifted event handler in P_ℓ is executed* at most once *in this step.*

Proof. This follows from (c11) and (c12) in P_ℓ (Fig. 6). Each h_i is a unique event handler, and is executed at most once in (c12). □

5.3 Exploring a single execution in P

Alg. 1 shows a straightforward procedure for exploring a single execution of a P program based on the semantics described in §3. We can lift this to value summaries to get an breadth-first explorer of all executions in a P program. To explore an execution from an initial configuration c, in each iteration of the while loop, the procedure calls ScheduleStep to nondeterministically chooses an enabled message and takes the step that receives it according to the Scheduling Step rule from Fig. 3. ScheduleStep, when called on arguments H, c returns a configuration c' such that the judgment $H \vdash c \xRightarrow{m} c'$ is derivable according to P's operational semantics for some message m. The explorer continues until it has fully explored a single execution of program P.

5.4 Symbolic Stateful Explorer

Alg. 2 shows our symbolic stateful explorer, which operates over a lifted program P_ℓ and makes use of a predicate *bufferNotEmpty* that is true whenever the given configuration has a nonempty global buffer under some guard. Ignoring the optional blue parts involving *prev*, Alg. 2 is a lifting of Alg. 1 to operate over value summaries rather than single values. It maintains a *frontier set* of configurations represented by a value summary vs_c rather than a single configuration as in Alg. 1 ,

where each symbolic step performs exploration by advancing configurations along the frontier by all possible schedule steps, resulting in a breadth-first search style of exploration.

Recall from the lifted semantics of the Schedule Step, Alg. 2 (line 5) explores *all* scheduling choices; given H, g, c, it returns configuration c' such that $H, g \vdash c \overset{m}{\Longrightarrow} c'$ is derivable in the lifted semantics. Because Alg. 2 operates over value summaries, which are fine-grained (requirement **R1**) and exhibit canonicity (requirement **R2**), it *avoids revisiting redundant configurations and retaking redundant transitions* in the same step. Alg. 2 continues (line 3) until there are no enabled messages in the global buffer under any guard, or, if the check on line 4 is included, until a fixed point is found (in which case *notFixedPoint* $=\bot$). The line 4 can be optionally included to perform a fixed point check (explained in §5.5). Without it, we assume *checkFixed* always returns $\bot$.

Soundness. We now state a theorem on the correctness of our explorer in Alg. 2 (without line 4).

Theorem 5.4 (Soundness of Symbolic Explorer). *For any program P with lifted program P_ℓ, if there exists an execution $c_0 \overset{m_0}{\Longrightarrow} c_1 \overset{m_1}{\Longrightarrow} \ldots$ of P, then Alg. 2 (without line 4) on P_ℓ explores an execution $vs_{c_0} \overset{vs_{m_0}}{\Longrightarrow} vs_{c_1} \overset{vs_{m_1}}{\Longrightarrow} \cdots$, such that for all i, $c_i \in Vals(vs_{c_i})$ and $m_i \in Vals(vs_{m_i})$.*

Proof. If for given program P with lifted program P_ℓ, there exists an execution $c_0 \overset{m_0}{\Longrightarrow} c_1 \overset{m_1}{\Longrightarrow} \ldots$, then Alg. 1 can explore this execution. Alg. 2 without line 4 explores the same executions as Alg. 2 without any of the blue parts. Alg. 2 without any of the blue parts is a lifting of Alg. 1, so from the properties of lifting, this holds. □

Intuitively, the theorem holds because each configuration in the execution of P is a part of some frontier configuration summary explored in P_ℓ and because Schedule Step runs handlers for all the enabled messages in the frontier configurations.

Efficient Symbolic Exploration. Because our frontier representations meet requirements **R1**-**R3**, and our lifted semantics based on these representations are efficient (Theorem 5.3), our explorer, which executes these semantics, avoids redundancies due to identical configurations and overlapping transitions in the frontier.

We now present a precision theorem that relates executions explored by Alg. 2 on a lifted program with executions of the P program, showing that Alg. 2 does not explore any spurious behaviors. Informally, this theorem follows from the fact that Alg. 2 starts from a configuration value summary that are initial configurations of the original P program and that the Schedule Step only executes event handlers for messages that are enabled in the P program.

Consider an execution $e = c_0 \overset{m_0}{\Longrightarrow} c_1 \overset{m_1}{\Longrightarrow} \ldots$ of a P program. We refer to $(c_0, m_0, c_1), (c_1, m_1, c_2) \ldots$ as its *execution triple sequence (ets)*, and do so analogously for executions of lifted P programs, where the ets is a sequence of triples of value summaries. For a sequence of triples of value summaries $\sigma_\ell = (vs_{c_0}, vs_{m_0}, vs_{c_0}), (vs_{c_1}, vs_{m_1}, vs_{c_2}) \ldots$, we let $Vals(\sigma_\ell)$ be the set of sequences $(c_0, m_0, c_1), (c_1, m_1, c_2) \ldots$, where $(c_i, m_i, c_{i+1}) \in Vals(vs_{c_i}) \times Vals(vs_{m_i}) \times Vals(vs_{c_{i+1}})$ and $\bigwedge_i getGuardFor(vs_{c_i}, c_i) \wedge getGuardFor(vs_{m_i}, m_i) \wedge getGuardFor(vs_{c_{i+1}}, c_{i+1})$ is not false. Intuitively, $Vals(\sigma_\ell)$ is the set of sequences with triples of values that are represented by the summaries in σ_ℓ.

Theorem 5.5 (Precision of Symbolic Explorer). *For any program P with lifted program P_ℓ, if Alg. 2 (without line 4) on P_ℓ explores an execution e_ℓ, the ets σ_ℓ of triples in e_ℓ is such that for all $\sigma \in Vals(\sigma_\ell)$, σ is the ets of an execution in P.*

Proof. This follows as a consequence of the following lemma. □

Lemma 5.6. *For any program P with lifted program P_ℓ, if Alg. 2 (without line 4) on P_ℓ explores an execution e_ℓ with nonempty prefix e_ℓ^p, the ets σ_ℓ^p of triples in e_ℓ^p is such that for all $\sigma_p \in Vals(\sigma_\ell^p)$, σ_p is the ets of an execution in P.*

Proof. We proceed by induction on the length of σ_ℓ^P. The base case is where the length is 1. In this case, $\sigma_\ell^P = (vs_{c_0}, vs_{m_0}, vs_{c_i})$. We have that $vs_{c_0} = \ell(c_0)$ from lifting P to P_ℓ and that $\{vs_{c_0}\} = Vals(\ell(c_0))$ from requirements on *Vals*. From inspecting Schedule Step in Fig. 5, we can see that the only handlers that can be run are those for the messages enabled in c_0, so any configuration $c \in Vals(vs_{c_1})$ must be such that there is a message m where $c_0 \overset{m}{\Longrightarrow} c$ is a step for the original P program. It follows that for any $\sigma_p \in Vals(\sigma_\ell^P)$, σ_p is an ets of an execution in P.

In the inductive case, we consider σ_ℓ^P of length $n+1$. Let $(vs_{c_n}, vs_{m_n}, vs_{c_{n+1}})$ be the final triple in σ_ℓ^P, and let σ_ℓ^n be the prefix of σ_ℓ^P of length n. From the inductive hypothesis, we know that any $\sigma^n \in Vals(\sigma_\ell^n)$ is an ets of an execution of P. It follows that for all $c_n \in Vals(vs_{c_n})$, there exists an execution of P that reaches this configuration. From again inspecting Schedule Step and reasoning about it similarly to the base case, we can see that for any configuration $c \in Vals(vs_{c_{n+1}})$ must be such that, for any $c_n \in Vals(vs_{c_n})$, there exists m where $c_n \overset{m}{\Longrightarrow} c$. It follows that σ_p is an ets of an execution in P. □

Symbolic Exploration Terminates. Alg. 2 (without line 4) terminates for all (liftings of) P systems with finite executions. Note that if P systems have only finite executions then for every execution, there is a configuration where none of the event handlers for the enabled messages send new messages, resulting in a strictly decreasing number of messages in the buffer until the buffer becomes empty. From Theorem 5.5, every configuration explored and every message explored by Alg. 2 on the lifted P program have corresponding configurations and messages in an execution of the P program. In particular, Alg. 2 does not add any spurious messages in the global buffer. Therefore, the global buffer during exploration by Alg. 2 will also have a decreasing number of messages until it becomes empty. At this point, the condition in Alg. 2, line 3 will be false, leading to termination. We illustrate this termination with the following example:

Example 5.7. Consider running Alg. 2 on a configuration value summary c with global buffer B_c containing two messages with summaries m_1, m_2, both enabled under the guard $g = D(m_1) = D(m_2)$. Assume that neither m_1 nor m_2 has a handler that sends a new message. Alg. 2 will terminate for this configuration value summary after running Schedule Step twice.

- **First Schedule Step.** Message value summaries m_1, m_2 will be in *choices*.
  - The Choose rule introduces a fresh Boolean variable b_0 to distinguish between the choices, associating b_0 with m_1 and $\neg b_0$ with m_2.
  - $(m_1|b_0), (m_2|\neg b_0)$ are removed from the buffer (c10), leaving m_1 in the buffer under guard $g \wedge \neg b_0$ and m_2 in the buffer under guard $g \wedge b_0$.
  - After restricting the resulting buffer with $(g \wedge b_0) \vee (g \wedge \neg b_0)$ (c14), it does not change.
- **Second Schedule Step.** At this point, we still have two enabled messages in the buffer (at the end of the above step). Let $m_1' = (m_1|g \wedge \neg b_0)$, and $m_2' = (m_2|g \wedge b_0)$.
  - The Choose rule introduces another fresh Boolean variable b_1 to distinguish between these choices, associating b_1 with m_1' and $\neg b_1$ with m_2'.
  - $(m_1'|b_1), (m_2'|\neg b_1)$ are removed from the buffer (c10), leaving m_1' in the buffer under guard $g \wedge \neg b_0 \wedge \neg b_1$ and m_2' in the buffer under guard $g \wedge b_0 \wedge b_1$.
  - After restricting the resulting buffer with $(g \wedge \neg b_0 \wedge b_1) \vee (g \wedge b_0 \wedge \neg b_1)$ (c14), the guards in the messages in the buffer both become false. Hence, the buffer is empty, and Alg. 2 terminates.

5.5 Symbolic exploration with fixed-point detection

So far, we have considered systems with only finite executions. If we want to handle systems with infinite executions, we can still terminate and remain sound if they have *fixed points* in their

Algorithm 3 Fixed point detection using value summaries.

```
1: procedure CheckFixed(prevFrontiers, vs_c)
2:     coveredGuard = ⊥
3:     for vs ∈ prevFrontiers do
4:         if coveredGuard ≠ D(vs_c) then coveredGuard ← coveredGuard ∨ GuardEqualUnder(vs, vs_c)
5:         else return coveredGuard
6:     return coveredGuard
```

executions, after which point no new behaviors of the system are explored. We can detect fixed points by finding a case where a configuration in the execution repeats; exploring past this repeated configuration will only yield more already-seen configurations.

When considering a configuration as repeating, we do not consider exact values of message ids, as the values of messages' unique ids do not affect execution. As long as there is a bijection between the message ids of one configuration and another such that applying this mapping to one configuration results in the other, we consider them the same. Let *sameC* be a function over pairs of configurations (c_1, c_2) that returns $\top$ iff there exists a bijection *map* for c_1 and c_2, where replacing each message id in c_1 with the result of calling *map* on it results in a configuration equal to c_2.

If we consider an explorer that explores a single execution at a time, to find fixed points, we need to detect when there are c_i, c_k in the execution so far $c_0 \xRightarrow{m_0} c_1 \xRightarrow{m_1} \ldots \xRightarrow{m_{n-1}} c_n$, where $sameC(c_i, c_k)$. This is similar to identifying *covered* nodes in program unwindings [Henzinger et al. 2002; McMillan 2006]: during exploration, for any node representing a state seen before on the same path, the node is regarded as covered, and the subtree at the node is no longer explored.

For detecting fixed points of lifted P programs, we can lift this approach to operate over configuration value summaries instead and thus apply it to frontiers of configurations. Recall that SCI Guards symbolically represent sets of sequences of nondeterministic choices that correspond to paths in the execution tree of the program. Using the the lifted version $sameC_\ell$ of *sameC*, which operates over configuration value summaries and returns a Boolean value summary, we can thus detect the guards under which a fixed point has been found. Specifically, for a value summary *vs* representing an earlier frontier and vs_c representing a later one, we use GuardEqualUnder(vs, vs_c) to compute the guard under which configurations in vs_c are repeats of configurations in *vs*. We define GuardEqualUnder as $\lambda x, y.getGuardFor(sameC_\ell(x, y), \top)$, where *getGuardFor* computes the guard under which the first argument has the value of the second argument (as described in §4.3). If GuardEqualUnder(vs, vs_c) does not return $D(vs)$, then there are paths under which some configuration in vs_c is not covered by (i.e., not included in) the set of configurations in *vs*.

Fixed-point detection in the lifted setting thus should check whether the current frontier value summary vs_c contains *only* configurations that have occurred in previously-seen frontiers (stored in the set of value summaries *prev*, Alg. 2) – if so, then the whole exploration has reached a fixed point. This can be done as shown in Alg. 3, which computes the SCI Guard *checkFixed* by performing a disjunction over the set of previous configurations:

$$checkFixed(prevFrontiers, vs_c) = \bigvee_{vs \in prevFrontiers} \{\text{GuardEqualUnder}(vs, vs_c)\}$$

Once all configurations in vs_c have been covered (line 5 in Alg. 3), a fixed point has been reached. If they have not all been covered ($checkFixed(prev, vs_c) \neq D(vs_c)$), then exploration should and does continue (see Alg. 2). This fixed point detection is enabled by the canonicity of value summaries (requirement **R1**) and by including all nondeterministic (including scheduling) choices in the guards (requirement **R3**). These features allow Alg. 3 to find fixed points that correspond to those in the original program executions, allowing Alg. 2 to terminate for P programs that contain only finite

executions or those that have fixed points. This is stated in the following theorem, which is a consequence of Theorem 5.5 and Alg. 3.

Theorem 5.8. *For any program P with lifted program P_ℓ, if every full execution $c_0 \xrightarrow{m_0} c_1 \xrightarrow{m_1} \dots$ of P is finite or such that there exists i, k with $i \neq k$, where $sameC(c_i, c_k)$, then when Alg. 2 with line 4 is run on P_ℓ, it terminates.*

Proof. Let the *effective length* of a full execution of P be n when the execution is n steps long, i.e., c_n is the last configuration of a full finite execution, or when n is the number of steps needed to reach a fixed point, i.e., where $sameC(c_i, c_n)$ for some $0 \leq i < n$. We assume that P is of finite length or has a fixed point, so every execution of P has an effective length. Furthermore, we have that there is a largest effective length N of any full execution of P.

We will now show that the loop in Alg. 2 cannot iterate more than N times. Let vs_{c_N} be the configuration that results from the N^{th} iteration of the loop on line 3. Let g_f be the guard under which vs_{c_N} represents configurations that have finite executions, i.e., for all $c \in (vs_{c_N}|g_f)$, c's buffer is empty. The next ScheduleStep taken by Alg. 2 thus cannot be made under any guard in g_f. Let g_∞ be the guard under which vs_{c_N} represents configurations that have infinite executions. Since N is the largest effective length of any execution of P, by Lemma 5.9 (below), we have that the result of *checkFixedPoint* is g_∞. After restricting by the negation of guard in Alg. 2, vs_c at the end of the iteration is such that $D(vs_c) = g_f$[2]. Since we have already established that the buffer is empty under guard g_f, Alg. 2 cannot iterate another time since the condition at line 3 is false. □

Lemma 5.9. *For any program P with lifted program P_ℓ, if there exists an execution $c_0 \xrightarrow{m_0} c_1 \xrightarrow{m_1} \dots \xrightarrow{m_{n-1}} c_n \xrightarrow{m_n} \dots$ of P where n is the smallest index such that $sameC(c_n, c_k)$ for $0 \leq k < n$, then for the execution $vs_{c_0} \xrightarrow{vs_{m_0}} vs_{c_1} \xrightarrow{vs_{m_1}} \cdots \xrightarrow{vs_{m_{n-1}}} vs_{c_n} \xrightarrow{vs_{m_n}} \cdots$ explored by Alg. 2 with line 4 on P_ℓ, the loop iterates at least n times and on the n^{th} iteration of the loop on line 3 of Alg. 2, checkFixed computes a guard formula f such that $\{c_k\} \in Vals((vs_n|d))$ for a disjunct d in f.*

Proof. (Sketch) We can show by induction that for any length i, for an execution of i transitions of P, if no pair of configurations in the first i are $sameC$[3], then for an execution of Alg. 2 on P_ℓ, the i^{th} call to ScheduleStep on line 5 yields a value summary vs_c such that the i^{th} configuration of P is in $Vals(vs_c)$. It follows that in the n^{th} iteration of the loop on line 3 (Alg. 2), after running ScheduleStep, $c_n \in Vals(vs_c)$. Alg. 2 then computes *checkFixed* on $([vs_{n-1}, \dots, vs_1, vs_0], vs_c)$. In Alg. 3, GuardEqualUnder is invoked on each pair of $vs_j \in [vs_{n-1}, \dots, vs_1, vs_0]$ and vs_c. From the definition of GuardEqualUnder, we have that GuardEqualUnder(vs_k, vs_n) results in formula d for which $\{c_k\} = Vals((vs_n|d))$. This formula d is disjoined to the current *coveredGuard* formula (Alg. 3, line 4) and is a disjunct in the final *coverdGuard* formula f returned by Alg. 3. □

In practice, the cost of fixed-point detection increases as greater depths are explored, since more BDD variables are introduced and *prev* grows larger. As a result, we use the fixed-point check in practice mostly when the set of possible configurations is small so that convergence is likely. Abstractions (described in §6.1) can help ensure faster convergence.

[2]Note that $g_f \vee g_\infty = D(vs_c)$.
[3]Note that an execution with i transitions has $i + 1$ configurations.

6 SYMBOLIC EXPLORATION WITH ABSTRACTIONS AND REDUCTIONS

6.1 Abstract Value Summaries

Our approach can employ abstractions [Cousot and Cousot 1977; Cousot et al. 2013; Fähndrich and Logozzo 2010; Flanagan and Qadeer 2002; Graf and Saïdi 1997] to convert large or infinite number of concrete configurations into a smaller and finite number of *abstract* configurations. As noted previously [Sen et al. 2015], one merit of a value summary representation is the ability to replace the value component of a primitive value summary with any representation – concrete values that occur when executing the program or abstractions of sets of such values.

Let $\mathbb{C}$ be a set of concrete values and $\mathbb{A}$ be an abstract set for these values, such that there exists a map $\alpha : \mathcal{P}(\mathbb{C}) \rightarrow \mathbb{A}$ that maps each element of $\mathcal{P}(\mathbb{C})$ to the element of $\mathbb{A}$ that abstracts it. We refer to value summaries whose primitive value summaries' value components are all in $\mathbb{C}$ as *concrete value summaries* and those whose value components are all in $\mathbb{A}$ as *abstract value summaries*.

Abstract Values. It is straightforward to abstract a concrete value summary to an abstract value summary by traversing the recursive structure of a given value summary and, for each value component c, applying the map α to the singleton set $\{c\}$.

Abstract Semantics. The semantics of any program defined over concrete value summaries can be lifted to work over the abstract domains in a manner similar to other settings. The semantics are as in §5.1, except that each concrete value (resp. value summary) is replaced by an abstract value (resp. abstract value summary). While we do not consider this in-depth here, for increased precision, we can perform special handling for branches on nondeterministic values whose nondeterminism is captured in the abstraction rather than in SCI Guards. The next section provides a case study on using abstractions to verify infinite-state distributed systems.

6.2 Partial-Order and Other Reductions as Filters

To extend our symbolic explorer with capabilities to perform reduction, we leverage *Filters* functions ($C \rightarrow \mathcal{P}(\mathcal{M}) \rightarrow \mathcal{P}(\mathcal{M})$). Each filter, when applied to a configuration and a set of enabled messages, returns a *subset* of the enabled messages that can be scheduled at that step.

Using Filters. Let $\mathit{ApplyFilters}(\{f_0, \ldots, f_n\}, c) = (f_0 c) \circ \cdots \circ (f_n c)(\{y \mid \texttt{enabled}(m, c)\})$, where $\circ$ denotes function composition. This function can be used to apply a set of *Filters* to the set of messages in a given configuration. *Filters* can be incorporated into the lifted P semantics by amending the Schedule Step rule. *Filters* should be added to the context of the judgment and the premise $m \in \ell(\lambda x.\mathit{ApplyFilters}(\mathit{Filters}, x))(c)$ should be added to the rule.

Reductions and pruning. We can implement several reductions using filters. Left movers from Lipton's theory of reductions [Lipton 1975] can be implemented with a function *LM* that guarantees that left movers are scheduled before any other enabled messages: $LM(c)(ms)$ returns $\{\mathit{left}\}$ for some $\mathit{left} \in ms$ if *left* is a left mover and otherwise returns *ms*. Persistent-set-based POR can be implemented with a filter function that removes messages outside of the persistent set. In §8, we make use of a persistent-based POR called Pred that we developed for P; the filter function is given by $\lambda c.\lambda ms.\{m \in ms \mid \forall m' \in B_c.m' \not<_c m\}$. This function removes from the enabled set all messages for which another message in the global message buffer is lower according to the $<$ relation, an extension of send-order (SO_c) with the happens-before relation. More details can be found in Appendix D. Monotonic POR [Kahlon et al. 2009] can be implemented similarly to persistent-set-based POR, and a variant of sleep-set-based POR [Godefroid 1990] can be implemented by using information across different executions being explored. We can also use filtering functions to implement sound prunings based on external analyses, e.g., of symmetries in the system.

7 ILLUSTRATIVE CASE STUDY: ABSTRACTIONS AND FIXED POINTS

We now show an example application where our symbolic stateful explorer uses abstractions and the fixed-point check to handle a program with an infinite state space and nonterminating runs.

Consider the example program shown in Fig. 7. The `TestKVStore` machine is initialized with nondeterministic values for an instance of a distributed key-value store implementation `kvStore`, a key `k` and a value `v` (lines 2-3). The initialization procedure for `kvStore` (not shown) performs a nondeterministic number of writes of random values to random keys. `TestKVStore` first writes the value `v` to `k` in the `kvStore` by initiating a write (line 6). Then, it issues a nondeterministic (possibly infinite) number of write transactions to `kvStore` (line 13). Each such write is of a nondeterministic value to a nondeterministic key not equal to `k`, and is performed only after the previous write has finished. `TestKVStore` then finally (line 25) issues a read for `k`. It asserts (line 28) that the result of this read transaction is equal to the initially-written value `v`.

```
1  machine TestKVStore {
2   var kvStore: KVStore;
3   var k : tKey; v : tVal;
4   start state Init {
5    entry {
6     send kvStore, eWriteTransReq, (key = k, val = v);
7    }
8    on eWriteTransRsp goto PerformNotKWrites;
9
10  state PerformNotKWrites {
11   entry {
12    if (*)
13     send kvStore, eWriteTransReq,
14                          (key = *Except(k), val = *);
15    else
16     goto CheckKey;
17   }
18   on eWriteTransRsp {
19     goto PerformNotKWrites;
20   }
21  }
22
23  state CheckKey {
24   entry {
25    send kvStore, eReadTransReq, k;
26   }
27   on eReadTransRsp (val: int) {
28    assert (val == v);
29   }
30  }
31 }
```

Fig. 7. Key-value store test client

The **choose** construct (denoted by *) is used (lines 12, 14) to choose a type-appropriate nondeterministic value. `*Except` (lines 14) chooses a nondeterministic value guaranteed to be not equal to its argument. If the set of `tKeys` or `tVals` is infinite, our system is infinite-state.

Suppose `KVStore` is implemented using a distributed commit protocol in which replica nodes each have a map `store` from keys to values to maintain the state of the key-value store. We will show how to use our symbolic exploration algorithm and appropriate abstractions to prove the following invariants:

Safety Invariant: The assertion on line 28 always holds.

Data Structure Invariant: When `TestKVStore` is not in state `Init`, `store` in each replica always map the key `k` to value `v`. Note that the data structure invariant is a global property about the data structures in any node of the system and the state of the `TestKVStore`.

Abstractions. We handle data nondeterminism by abstracting via the following predicates:

- *EqKey*, indicates whether or not a key is equal to the distinguished key k
- *EqVal*, indicates whether or not a value is equal to the distinguished value v

After this abstraction, the only remaining nondeterministic call in Fig. 7 is the call to `*`.

Symbolic Exploration with Abstract Value Summaries. We now focus on some components of the abstract value summary as they change during exploration. For the i^{th} message sent by `TestKVStore`, let c_i be the configuration after response for the corresponding transaction has been processed. That is, if the message was a `eWriteTransReq` (resp. `eReadTransReq`), then the corresponding `eWriteTransRsp` (resp. `eReadTransRsp`) has been sent by `kvStore` and received by `TestKVStore`. For each configuration c_i, Table 1 shows **(1)** the map `store` in the replica machines of the implementation, **(2)** the machine state of `TestKVStore` after each c_i, and **(3)** the result of calling *checkFixed* for c_i.

Note that the `store`, `TestKVStore` machine state, and global buffer are the only components of the configuration of significance involved in computing *checkFixed*, since the remaining configuration components will all be the same (i.e., GuardEqualUnder would return $\top$) for these configurations. We do not show the global buffer because the results for GuardEqualUnder for it and the `TestKVStore` state are always the same. We also show only a single `store` map because all

Config.	Abstract Value Summary (partial)	
c_0	`store`	$\{\neg EqKey \mapsto \{(g, \top)\}, EqKey \mapsto \{(g, \top)\}\}$
	`TestKVStore`	$(\top, \texttt{Init})$
	checkFixed	$\bot$
c_1	`store`	$\{\neg EqKey \mapsto \{(g, \top)\}, EqKey \mapsto \{(\top, EqVal)\}\}$
	`TestKVStore`	$(\top, \texttt{PerformNotKWrites})$
	checkFixed	$\bot$
c_2	`store`	$\{\neg EqKey \mapsto \{(g \vee b_0, \top)\}, EqKey \mapsto \{(\top, EqVal)\}\}$
	`TestKVStore`	$\{(b_0, \texttt{PerformNotKWrites}), (\neg b_0, \texttt{CheckKey})\}$
	checkFixed	$g \wedge b_0$
c_3	`store`	$\{\neg EqKey \mapsto \{(g \vee b_0 \vee b_1, \top)\}, EqKey \mapsto \{(\top, EqVal)\}\}$
	`TestKVStore`	$\{b_0 \wedge (g \vee b_1), \texttt{PerformNotKWrites}), (\neg b_0 \vee \neg g \wedge \neg b_1, \texttt{CheckKey})\}$
	checkFixed	$(b_0 \wedge (g \vee b_1)) \vee \neg b_0$
c_4	`store`	$\{\neg EqKey \mapsto \{(g \vee b_0 \vee b_1, \top)\}, EqKey \mapsto \{(\top, EqVal)\}\}$
	`TestKVStore`	$\{b_0 \wedge (g \vee b_1), \texttt{PerformNotKWrites}), (\neg b_0 \vee \neg g \wedge \neg b_1, \texttt{CheckKey})\}$
	checkFixed	$\top$

Table 1. Symbolic exploration: (Partial) Abstract Value Summaries for Configurations

replicas should have the same abstract value summary for their local `store` in these configurations. Exploration begins with c_0, where `store` is empty under SCI Guard formula $\neg g$ and otherwise nondeterministically maps any key to any value. After processing the write transaction on line 6, it arrives at c_1. We detail how the exploration arrives at c_2 from c_1:

When encountering the nondeterministic choice on line 12 after reaching c_1, the Choose rule (Fig. 5.1) introduces guard b_0, which is used to indicate that the then-branch is taken in the nondeterministic branch in state `PerformNotKWrites`. When b_0 holds, the second message sent to `kvStore` is a `eWriteTransReq` sent in state `PerformNotKWrites`, and when it does not, it is `eReadTransReq` sent in state `CheckKey`, as indicated in the `TestKVStore` state component for c_2 shown in Table 1. The processing of the write on line 13 under guard b_0 results in the `store` maps mapping keys other than `k` (abstracted as $\neg EqKey$) to any key (abstracted as $\top$). This is reflected in the `store` component for c_2 in Table 1.

At this point, the explorer will use *checkFixed* to detect guards under which configurations have been covered. Configuration value summaries c_2 and c_0 are equal only under guard $\bot$, but c_2 and c_1 are equal under guard $g \wedge b_0$: i.e., $(c_2|g \wedge b_0) = (c_1|g \wedge b_0)$. The covered configurations are thus those in $(c_2|g \wedge b_0)$, which specify those configurations under which the abstract `store` maps $\neq EqKey$ to $\top$ and the abstract `TestKVStore` is in state `PerformNotKWrites`. Because these configurations are covered, we do not continue exploration from these configurations. The explorer thus advances configurations under guard $\neg g \vee \neg b_0$ to get to c_3.

The third and fourth messages are the same as the second and give us configurations such that $c_4 = c_3$. *checkFixed* returns $\top$ since all configurations represented by c_4 are now covered, and exploration concludes since a fixed point has been reached.

Deriving Invariants. At this point, since the assertion in the `TestClient` state has not been violated, we also have shown that the safety property holds. We can also use the abstract value summaries to derive invariants such as the data structure invariant described above; if the invariant holds for each visited configuration up until we reach a fixed point, then it is inductive. For all configurations throughout the execution, calling *getGuardFor* on the `TestKVStore` state and the value `Init` yields either $\top$ (for c_0) or $\bot$ (for c_i, $i > 0$). We restrict `store` by the negation of this guard for each visited configuration c_i to check the data structure invariant. In particular, we have $(\texttt{store}|\ \bot) = \{\}$ for c_0, showing that there is no value for `store` to consider in c_0. For each c_i for $i > 0$, we have that $(\texttt{store}|\top) = \{\neg EqKey \mapsto \{(\phi_i, \top)\}, EqKey \mapsto \{(\top, EqVal)\}\}$ for some guard

Table 2. Comparison between TLC and Psym, with 12GB memory (max) and 15-minute timeout.

Benchmark	TLC	Psym	Benchmark	TLC	Psym
Chang-Roberts (N=5)	7s	**1s**	2PCwithBTM	17s	**10s**
HLC (MaxT=3)	18s	**3s**	BenOr (MaxRound = 2)	24s	**7s**
HLC (MaxT=4)	24s	**8s**	BenOr (MaxRound = 3)	1m20s	1m11s
HLC (MaxT=5)	42s	**24s**	Paxos (simple, STOP=3, M=2, MAXB=3)	3m20s	3m21s
Streamlet Blockchain	Timeout	**5s**	Paxos (full, STOP=1, M=9, MAXB=10)	Timeout	**6m54s**

ϕ_i. None of the restricted `store` value summaries ever map *EqKey* to an abstract value summary containing a guarded value (g, ψ) for $\psi \implies \neg EqVal$, so it is indeed an invariant that whenever `TestKVStore` is not in state `Init`, then `store` maps distinguished key k to distinguished value v.

8 EVALUATION

Implementation. We implemented our approach in the Psym tool as an extension of the open-source P framework. Our implementation consists of three parts: **(1)** a Java library that implements a symbolic runtime with support for creating and manipulating value summaries (as described in §4.3); **(2)** a P compiler extension to generate Java code that symbolically represents lifted P program using the value summary implementations in the symbolic runtime (thus lifting the P program over value summaries as described in §5.1); and finally, **(3)** a systematic explorer that implements Alg. 2. Guards in value summaries are represented using the PJBDD [Beyer et al. 2021] library. Unless otherwise stated, experiments were run on a Linux machine with Intel(R) Xeon(R) CPU @ 2.30GHz, 16 cores, and 200GB main memory.

Research Questions. Our evaluation seeks to answer the following research questions:

Q1. *How does Psym compare against a state-of-the-art model checker for verifying distributed protocols?*

Q2. *Does Psym succeed in verifying P models from previous papers?*

Q3. *Does Psym scale to verify distributed protocols from real-world industrial systems created by developers (not by the authors)?*

Q4. *Does Psym succeed in verifying infinite state systems and how does it compare against other approaches for systematic exploration?*

We use the Pred partial-order reduction in Psym in all experiments except in Q1.

Characteristics of models. Other than TLA$^+$ models (in §8.1 for Q1), each P model we consider has all types of nondeterminism: asynchronous message interleaving, dynamic machine creations, and data or input nondeterminism (e.g., node failures). All the models are closed and have a finite number of configurations, i.e., for each system, there are a finite number of processes, and the system takes finite inputs. Also, for each model (other than those in §8.4 for Q4, which have infinite executions), every execution is terminating and the total number of executions is finite.

8.1 Q1: Comparison with TLC (model checker for TLA+)

P no longer supports the Zing [Andrews et al. 2004] model checking backend used in previous approaches [Desai et al. 2013b]; instead it relies on Coyote [Microsoft Coyote 2022] for randomized testing of P models and has no support for verification. To demonstrate the efficacy of our approach, we would like to compare against a state-of-the-art model checker for distributed systems (not built by the authors). TLA$^+$ [Lamport 2002] is a popular modeling language and is a standard in industry [Newcombe 2014] and academia [TLA$^+$ 2023] for model-checking distributed protocols. TLC [Yu et al. 1999] is an explicit state model checker for TLA$^+$ models and has evolved over the years with optimizations to scale model checking to complex models.

To evaluate the efficacy of our approach, we compare Psym with TLC on several open-source TLA$^+$ models, which we manually translated into P programs. Note that P and TLA$^+$ support different models of computation; our objective with this experiment is to compare the search exploration

Table 3. Benchmarking Psym on standard distributed protocols

Benchmark	LOC (P)	#M	Time	#Steps w/ (and w/o) overlapping transitions	Mem.
Token Ring	164	5	2s	143 (243)	27 MB
BoundedAsync	96	4	11s	534 (534)	45 MB
German	283	5	44s	244 (300)	860 MB
Failure Detector	189	7	38s	277 (435)	1.4 GB
Two-Phase Commit	284	7	57m22s	27 (2643)	13 GB
Paxos	241	8	2h7s	931 (2143)	18 GB

techniques implemented by Psym and TLC. We confirmed through manual review that the P models have same nondeterminism as allowed in the TLA$^+$ models. These models include the following: Hybrid Logical Clocks [Demirbas 2017a; Kulkarni et al. 2014] with a parameterizable maximum time (`MaxT`); the decentralized consensus algorithm proposed by Ben-Or [Ben-Or 1983; Demirbas 2019]; a version of two-phase commit with a transaction manager [Demirbas 2017b,c]; a simplified version and the full version of the Flexible Paxos distributed consensus protocol [Howard et al. 2016], for which the open-source TLA$^+$ model has four proposers [fpa 2022] and a parameterizable number of values to reach consensus on (`STOP`), number of proposers (`M`), and maximum number of ballots (`MAXB`); the Streamlet blockchain protocol [Chan and Shi 2020]; the Chang-Roberts ring leader election algorithm [cha 2021; Chang and Roberts 1979] with a parameterizable number of nodes (`N`). We compare Psym against TLC when run with 1 worker thread (Psym is single-threaded), and limit both tools to 12GB memory. The results are shown in Table 2; experiments were run on a Macbook Pro with an M1 processor, 16GB RAM with a 15-minute timeout.

For a fair comparison to TLC (which does not implement POR), we do not use the Pred POR in Psym for these experiments – thus, our results are due to use of symbolic representations and targeting additional redundancies due to overlapping transitions. On the other hand, TLC uses state hashing to avoid re-exploring from previously-seen configurations. In contrast, we do not currently use any state hashing in Psym (and plan to implement it in future work). In Psym, merging allows us to avoid re-exploration from the same configurations in the same frontier.

For most benchmarks, there are many distinct transitions that are overlapping, leading Psym to outperform TLC. In particular, in both Paxos and Streamlet, there are many instances of the same process that perform the same or similar computations (e.g., finding the maximal notarized chain in Streamlet). Psym is able to detect and take advantage of these overlapping transitions, to successfully complete verification, whereas TLC timed out. For the remaining benchmarks, Psym showed a runtime improvement of **2.5X** on average (geometric mean) in comparison to TLC. However, some benchmarks have configurations that are the same at different depths in the execution tree, where Psym may perform redundant work.

8.2 Q2: Evaluation on P programs from previous papers

We evaluated Psym on common distributed protocols available in the P GitHub repository, used as benchmarks in previous publications [Deligiannis et al. 2015; Desai et al. 2015; Liu et al. 2019]. These are: (consensus) Two-Phase Commit [Gray and Lamport 2006] and Paxos [Lamport 2001]; (leader election) Token Ring [Lynch 1996], Failure Detector, and Bounded broadcast [Liu et al. 2019]; and German cache coherence [Pnueli et al. 2001]. Our Two-Phase Commit benchmark has two clients and two participants, where each client tries to perform two read and two write transactions; Paxos runs 4 rounds and has two proposers, which each propose a single value, and three acceptors. The Token Ring is of size 4. We model failures for these benchmark. While some of these benchmarks (e.g., German, Token Ring) exhibit symmetry, we did not use symmetry-based pruning filters.

Table 3 shows the results of running Psym on these benchmarks, where we report the number of lines of P code (LoC), the number of concurrently executing state machines (#M), the time taken

Table 4. Verification of industrial case studies using Psym (L: Lipton reduction, S: Symmetry based pruning)

Case Study	LOC (P)	#M	Psym		Psym with custom filters		
			Time	# Step (Mem.)	Time	# Step (Mem.)	Filter
OTA	1103	14	33m23s	678 (33 GB)	15m6s	422 (5.3 GB)	L
Data migration	959	22	6h23m11s	5332 (48 GB)	2h23s	2318 (15 GB)	L, S
Data replication	1459	16	8h4m1s	8132 (59 GB)	3h1m14s	3118 (16 GB)	L, S
Conflict resolution	1202	13	7h22m1s	2182 (122 GB)	5h1m	1164 (44 GB)	L, S

(Time), and the max memory consumed (Mem). The #Steps column reports the number of (lifted) event handler invocations made during exploration, where the number in parentheses reports the number of event handler invocations that would have occurred without handling messages with the same event handlers in one step. This demonstrates the benefits from identifying overlapping transitions. For example, in Two-Phase Commit, many different participant machines send messages to a single coordinator machine. Because all participants send only two kinds of messages to the coordinator, only (at most) two lifted event handler executions are needed to handle the messages sent to the coordinator. We found that in most of these distributed protocols, the configurations reached after exploring different interleavings of messages, still have many equivalent components, leading to efficient handling of redundancies in overlapping transitions.

8.3 Q3: Verifying industrial case studies: distributed storage and database protocols

We used Psym to verify some real-world industrial distributed protocols from the storage, IoT, and database systems at Amazon Web Services (AWS); these protocol models were implemented by expert engineers in AWS, and have the characteristics described earlier with failures and asynchrony. We only applied Psym and helped developers identify opportunities to apply sound reductions. We used Psym to verify the correctness of four industrial case studies: (1) a distributed protocol used at Amazon S3 for reliable data migration; (2) a distributed protocol used at Amazon S3 for durable data backup; (3) The AWS Over the Air (OTA) protocol used to update an IoT device firmware reliably and securely; and (4) a multi-region distributed transaction conflict resolution protocol used for a database service at AWS. We asserted safety properties such as consistency for the distributed transaction conflict resolution protocol and reliable file transfer for OTA. Psym could verify finite instances of these models, whereas the other techniques that we tried failed (see a discussion in §8.4). Finite instances resulted from bounding the number of processes and inputs, but not depth. For example, Psym verified the conflict resolution protocol for 2 geographical regions, each with 3 replicas, and 2 clients concurrently sending 3 non-deterministic commands (insert, delete, update) to any region.

Table 4 reports the results of running Psym on these benchmarks both with and without additional reductions. Psym was able to successfully verify the correctness of these benchmarks, demonstrating that it can be used as a verifier for realistic systems. These results also demonstrates the effectiveness of incorporating custom filters into Psym. Specifically, the OTA protocol makes several synchronous interactions with other components, and the response messages for these interactions are *left movers* based on Lipton's theory of reductions [Lipton 1975]. The data migration and replication protocols use a centralized locking service to gain consensus, and the release message from this service is also a left mover. We also found that certain operations (messages) can be safely pruned because of symmetry. Thus, adding Lipton's left mover (L) and symmetry-based pruning (S) filters helped reduce the number of interleavings, which led to smaller sizes of the representations and hence faster verification times. The ability of Psym to integrate these filters proved effective.

Table 5. Benchmarking Psym with abstractions

Benchmark	LOC (P)	#M	Time	# Step	Mem.
Two-Phase Commit(2)	292	5	<1s	48 (48)	8.1
Two-Phase Commit(4)	292	7	<1s	80 (80)	8.7 MB
Two-Phase Commit(6)	292	9	1s	112 (112)	9 MB
Paxos(3)	421	6	12s	142 (142)	123.3 MB
Paxos(5)	421	8	3h18s	145 (145)	18 GB

8.4 Q4: Verifying infinite-state systems and comparison with other approaches

Infinite-state systems. Here we consider evaluating Psym on variations of the case study in §7 using the abstractions described. We consider a two implementations of the key-value store: one based on Two-Phase-Commit (modeled without failtures) and another based on the Paxos consensus protocol. We evaluate Psym with both kinds of implementation for several instance sizes of the protocols. Two-Phase Commit(n) denotes that there are n participant machines in the protocol. Paxos(n) denotes that there are n acceptor nodes. Table 5 reports the result of this evaluation. We can see that our abstractions effectively finitize the infinite state space, thereby allowing Psym to converge on these benchmarks by using our fixed-point detection method.

The problem of systematic exploration of distributed systems behaviors is a well-studied problem with several different directions. In the interest of positioning Psym among closely related techniques, we briefly summarize our observations and experience with other tools.

Comparison with bounded model checking. Before developing our proposed approach, we experimented with building Psym using bounded model checking [Biere et al. 1999]. We hand-coded the Two-Phase Commit protocol in the open-source tools UCLID5 [Seshia and Subramanyan 2018] and JKind [Gacek et al. 2018]. Both tools failed to verify Two-Phase Commit within 2 hours. While Psym uses a large memory footprint typical for BDD-based tools, we are encouraged that Psym is able to successfully complete verification on many challenging benchmark examples, with sizes that exceed those used to demonstrate successful prior tools like I4 or DistAI. For verification/inference techniques that rely on systematic exploration or model checking of small or finitized instances of a particular distributed system, using Psym can potentially help to improve their effectiveness.

BDD vs. SAT (Memory vs. Time tradeoff). BDD-based and SAT/SMT-based verification engines often have complementary strengths – e.g., many commercial hardware model checkers have both engines. BDDs are especially well-suited for iterative computations over sets of states, while SAT/SMT solvers work better in non-iterative settings where state sets are not needed, e.g., when checking verification conditions (for inductiveness, safety, etc.). We tried using a SAT solver in place of BDDs for manipulating guards in value summaries. As expected, the memory footprint was much smaller, but for all the benchmarks reported in this paper, its runtime performance was orders of magnitude worse than with BDDs. Too many SAT queries were needed (even with techniques like FRAIGs [Mishchenko et al. 2005]); the canonicity of BDDs allows for efficient querying.

Comparison with Dynamic POR. We compare Psym with Concuerror, a state-of-the-art model checker for Erlang that uses optimal dynamic partial order reduction (ODPOR) [Abdulla et al. 2014; Aronis et al. 2018]. Note that ODPOR does not compute sets of reachable configurations, but a comparison is of interest because our Psym does not use dynamic POR. We consider two synthetic benchmarks: the `lastzero` example from [Abdulla et al. 2014] and a modified version (`counter`) of the example in Fig. 1. Table 6 reports timing results for Concuerror and Psym. We also report the steps taken and max memory used by Psym (not reported by Concuerror). For `lastzero`, we report the results from the ODPOR paper which were obtained on an i7-3770 CPU (3.40 GHz), 16GB of RAM [Abdulla et al. 2014]. All other experiments were run on a Macbook Pro with an M1 processor, 16GB RAM. For Psym, we also implemented a monotonic POR [Kahlon et al. 2009] (MPOR) filter

Table 6. Comparison with ODPOR (✖: failed to finish in 1 hour, *: results reported from [Abdulla et al. 2014], Erlang source not available, N/A: MPOR filter not used)

Benchmark	#M	ODPOR	PSym		PSym with MPOR Filter	
		Time	Time	#Step (Mem.)	Time	#Step (Mem.)
`lastzero(5)`	13	32s*	1m33s	371 (3.6GB)	**3s**	134 (20MB)
`lastzero(10)`	23	27.61s*	✖	283 (9.5GB)	**12s**	370 (180MB)
`lastzero(15)`	33	30m13s*	✖	317 (8.4GB)	**73s**	705 (1GB)
`counter(5)`	6	47s	**1s**	11 (8.1MB)	N/A	N/A
`counter(10)`	11	34m15s	**7.6s**	21 (200MB)	N/A	N/A
`counter(15)`	16	✖	**5m17s**	122 (13GB)	N/A	N/A

for `lastzero`, enabling PSym to verify the benchmark efficiently. The results suggest that even without ODPOR, MPOR can help PSym scale.

9 RELATED WORK

Verification and Invariant Discovery for Distributed Systems. Many approaches use deductive proof techniques to formally verify distributed systems [Hawblitzel et al. 2015; Padon et al. 2016; Wilcox et al. 2015]. They can prove correctness of distributed systems for unbounded inputs but require users to provide complex inductive invariants.

Model checking provides an algorithmic approach for verifying either a model (e.g., SPIN [Holzmann 1997], Zing [Andrews et al. 2004], TLC [Yu et al. 1999]) or an implementation (e.g., Verisoft [Godefroid 1997], JPF [Visser and Mehlitz 2005], CHESS [Musuvathi and Qadeer 2007]), but suffers from state space explosion. Several sequentialization techniques [La Torre et al. 2009; Lal and Reps 2008] reduce verification of concurrent or distributed programs to verification of sequential programs, but the resulting sequentialized programs are highly nondeterministic. State caching (used e.g., in TLC [Yu et al. 1999]) addresses some scalability issues by avoiding re-exploration from redundant configurations. We provide an approach that *additionally* avoids repeating redundant computations in updates due to overlapping transitions.

Recent efforts in automated invariant discovery for distributed protocols generalize from small instances of protocols to learn invariants for all instance sizes [Ma et al. 2019; Yao et al. 2021]. Our explorer could be used as a subprocedure in such techniques e.g., as a model-checker for a technique such as in I4 [Ma et al. 2019], or as a systematic explorer for a data-driven technique such as in DistAI [Yao et al. 2021]. It may also be used to generate additional invariants as candidates for generalization in either kind of technique.

Symbolic and Abstraction Techniques. Symbolic model checking [Burch et al. 1992] has been successfully applied to many problems but does not scale well when applied to concurrent or distributed systems due to state space explosion. Recent efforts include a symbolic model checker for TLA$^+$ [Konnov et al. 2019], which improves scalability of invariant checking but does not show clear advantages over TLC for model checking safety properties; it is much slower than TLC on distributed benchmarks such as two-phase commit and Paxos. (We thus compared only against TLC in our experiments.)

Symbolic and concolic execution have been somewhat effective in addressing state space explosion due to input and control nondeterminism by focusing on one control path at a time [Cadar et al. 2008; Farzan et al. 2013; Godefroid et al. 2005; Sen and Agha 2006b]. However, this leads to path explosion for reasonable coverage. Several techniques have been proposed to handle path explosion by state merging [Anand et al. 2008; Kuznetsov et al. 2012; Sen et al. 2015; Torlak and Bodík 2014]. We follow in the steps of MultiSE [Sen et al. 2015], extending their guards to handle scheduling choices as well as adapting their symbolic representation. *To the best of our knowledge, no existing approaches have proposed handling scheduling choices symbolically or merge state in distributed*

systems as we do. One work [Sen and Agha 2006a] combines concolic execution with POR and uses backtracking to explore both different scheduling choices and control-flow paths but does not involve a symbolic encoding of scheduling choices. A more recent effort [Schemmel et al. 2020] also combines POR and symbolic execution but also does not represent scheduling nondeterminism symbolically. Our fixed point check is an adaptation of an existing covering approach for model checking [McMillan 2006] to work on configuration value summaries.

Abstraction has been widely applied in many domains to help combat state-space explosion and finitize infinite-state systems [Cousot and Cousot 1977; Cousot et al. 2013; Fähndrich and Logozzo 2010; Flanagan and Qadeer 2002; Graf and Saïdi 1997]. We leverage prior work on abstraction to achieve similar benefits, allowing our technique to scale even further.

POR and Symmetry Reductions. (Dynamic) POR [Abdulla et al. 2014; Aronis et al. 2018; Flanagan and Godefroid 2005; Kahlon et al. 2009; Nguyen et al. 2018; Peled 2018] and symmetry reduction [Iosif 2002] address state space explosion problem by taking a control-centric view during the exploration of a distributed system's state space. They do not consider the state of the system during exploration.

Our approach provides a complementary data-centric view that allows our explorer to recognize not only redundant interleavings (as in POR) and configurations (as in state caching [Holzmann 1997]) but also redundancies *in overlapping transitions*. As seen previously, we can integrate POR and symmetry reductions with our proposed techniques, but there are limitations in combining our approach with Dynamic POR techniques [Abdulla et al. 2014; Aronis et al. 2018; Flanagan and Godefroid 2005] that rely on a depth-first order of exploration.

Systematic Testing of Distributed Systems. Systematic testing has enjoyed success in uncovering successfully uncovered deep, hard-to-find bugs in distributed systems. Systematic testing techniques often use prioritized search [Deligiannis et al. 2015; Desai et al. 2015; Leesatapornwongsa et al. 2014; Mukherjee et al. 2020] and stratified random testing [Deligiannis et al. 2015; Jepsen 2021; Killian et al. 2007; Majumdar and Niksic 2017; Ozkan et al. 2018] to guide exploration . They typically do not aim to exhaustively explore behaviors as we do here. For P programs in particular, researchers have proposed bounded exploration based on delay-bounding [Desai et al. 2015] and reductions based on almost-synchronous invariants [Desai et al. 2014]; our work is the first to use POR specific to P semantics (in the form of Pred). Recent work [Liu et al. 2019] presents sound partial abstract transformers for verification of P programs. This approach, like ours, computes fixed points over abstract states, but it does not use a fine-grained configuration representation nor identify overlapping transitions. Furthermore, it is unclear how to combine POR with this approach.

10 CONCLUSIONS

We presented a novel approach for scalable stateful exploration of distributed systems implemented in P. Our approach leverages a novel canonical fine-grained symbolic representation of distributed system configurations, yielding an explorer that implicitly recognizes equivalent configurations and reduces redundancies due to overlapping transitions. Our explorer is designed so that scalability can be further increased via abstractions, POR techniques, and other reductions. Our evaluation shows that our tool Psym outperforms a state-of-the-art stateful explorer and that it can successfully verify P models of common distributed protocols and challenging industrial case studies. Future work includes the use of our approach in invariant discovery techniques and the implementation of state caching in Psym.

11 ACKNOWLEDGMENTS

We would like to thank Aman Goel, Cambridge Yang, William Brandon, and Eric Ge who contributed to Psym. We would also like to thank our anonymous reviewers for their valuable suggestions.

This work was supported in part by the National Science Foundation under Grant # 2127309 to the Computing Research Association for the CIFellows project, NSF-1837030, and an Amazon research award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the Computing Research Association.

REFERENCES

2021. Chang-Roberts algorithm for leader election in a ring. https://github.com/tlaplus/Examples/tree/616c4c2e00dd7084c623d1dcc83b140279652fb4/specifications/chang_roberts

2022. TLA+ Specification of Flexible Paxos. https://github.com/fpaxos/fpaxos-tlaplus

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *POPL*. ACM, 373–384.

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A Foundation for Actor Computation. *J. Funct. Program.* 7, 1 (1997), 1–72.

Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 367–381.

Tony Andrews, Shaz Qadeer, SriramK. Rajamani, Jakob Rehof, and Yichen Xie. 2004. Zing: A Model Checker for Concurrent Software. In *Proceedings of CAV*.

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *TACAS (2) (Lecture Notes in Computer Science, Vol. 10806)*. Springer, 229–248.

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885.

Michael Ben-Or. 1983. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In *PODC*. ACM, 27–30.

Dirk Beyer, Karlheinz Friedberger, and Stephan Holzner. 2021. PJBDD: A BDD Library for Java and Multi-Threading. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 144–149.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, Proceedings (Lecture Notes in Computer Science, Vol. 1579)*. Springer, 193–207.

Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic Model Checking: 10ˆ20 States and Beyond. *Inf. Comput.* 98, 2 (1992), 142–170.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX Association, 209–224.

Sagar Chaki and Arie Gurfinkel. 2018. BDD-Based Symbolic Model Checking. In *Handbook of Model Checking*. 219–245.

Benjamin Y. Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. In *AFT*. ACM, 1–11.

Ernest J. H. Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (1979), 281–283.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model checking*. MIT Press.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *VMCAI (Lecture Notes in Computer Science, Vol. 7737)*. Springer, 128–148.

Pantazis Deligiannis, Alastair F Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. Asynchronous programming, analysis and testing with state machines. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 154–164.

Murat Demirbas. 2017a. https://github.com/muratdem/HLC

Murat Demirbas. 2017b. https://github.com/muratdem/PlusCal-examples/blob/master/2PCTM/2PCwithBTM.tla

Murat Demirbas. 2017c. TLA+/Pluscal modeling of 2-phase commit transactions. http://muratbuffalo.blogspot.com/2017/12/tlapluscal-modeling-of-2-phase-commit.html

Murat Demirbas. 2019. The Ben-Or decentralized consensus algorithm. https://muratbuffalo.blogspot.com/2019/12/the-ben-or-decentralized-consensus.html

Ankush Desai. 2022. Formal Modeling and Analysis of Distributed Systems. https://www.youtube.com/watch?v=5YjsSDDWFDY

Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 709–725. https://doi.org/10.1145/2660193.2660211

Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013a. 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*. 321–332. https://doi.org/10.1145/2491956.2462184

Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013b. P: safe asynchronous event-driven programming. In *PLDI*. ACM, 321–332.

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

Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 73–83. https://doi.org/10.1145/2786805.2786861

Ankush Desai, Serdar Tasiran, and Vishwas Narenda. 2021. Amazon S3 Strong Consistency. https://www.twitch.tv/videos/962963706?t=0h15m15s. [Online; accessed 2022].

Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *FoVeOOS (Lecture Notes in Computer Science, Vol. 6528)*. Springer, 10–30.

Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 37–47.

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *POPL*. ACM, 110–121.

Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *POPL*. ACM, 191–202.

Andrew Gacek, John Backes, Mike Whalen, Lucas G. Wagner, and Elaheh Ghassabani. 2018. The JKind Model Checker. In *CAV (2) (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 20–27.

P GitHub. 2021. P Case Studies. https://p-org.github.io/P/casestudies/. [Online; accessed 2021].

Patrice Godefroid. 1990. Using Partial Orders to Improve Automatic Verification Methods. In *CAV (Lecture Notes in Computer Science, Vol. 531)*. Springer, 176–185.

Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *Proceedings of POPL*. 174–186.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. ACM, 213–223.

Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *CAV (Lecture Notes in Computer Science, Vol. 1254)*. Springer, 72–83.

Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Trans. Database Syst.* 31, 1 (March 2006), 133–160.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*.

Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 58–70.

Gerard Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. on Software Engineering* (1997).

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum Intersection Revisited. In *OPODIS (LIPIcs, Vol. 70)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:14.

Radu Iosif. 2002. Symmetry Reduction Criteria for Software Model Checking. In *SPIN (Lecture Notes in Computer Science, Vol. 2318)*. Springer, 22–41.

Jepsen. 2021. Jepsen Tool. https://jepsen.io/.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 637–650.

Vineet Kahlon, Chao Wang, and Aarti Gupta. 2009. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV (Lecture Notes in Computer Science, Vol. 5643)*. Springer, 398–413.

Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Symposium on Networked Systems Design and Implementation*.

Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 123:1–123:30.

Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In *International Conference on Principles of Distributed Systems*. Springer, 17–32.

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *Acm Sigplan Notices* 47, 6 (2012), 193–204.

Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2009. Reducing context-bounded concurrent reachability to sequential reachability. In *International Conference on Computer Aided Verification*. Springer, 477–492.

Akash Lal and Thomas Reps. 2008. Reducing concurrent analysis under a context bound to sequential analysis. In *International Conference on Computer Aided Verification*. Springer, 37–51.

Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (Dec. 2001).

Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

Richard J Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12 (1975), 717–721.

Peizun Liu, Thomas Wahl, and Akash Lal. 2019. Verifying asynchronous event-driven programs using partial abstract transformers. In *International Conference on Computer Aided Verification*. Springer, 386–404.

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *SOSP*. ACM, 370–384.

Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–24.

Antoni W. Mazurkiewicz. 1986. Trace Theory. In *Advances in Petri Nets (Lecture Notes in Computer Science, Vol. 255)*. Springer, 279–324.

Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *CAV (Lecture Notes in Computer Science, Vol. 4144)*. Springer, 123–136.

Microsoft Coyote. 2022. Fearless coding for reliable asynchronous software. https://github.com/microsoft/coyote

Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert K Brayton. 2005. *FRAIGs: A unifying representation for logic synthesis and verification*. Technical Report. ERL Technical Report.

Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of PLDI*.

Chris Newcombe. 2014. Why amazon chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 25–39.

Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. 2018. Quasi-Optimal Partial Order Reduction. In *CAV (2) (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 354–371.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

P-GitHub. 2023. The P Programming Langugage. https://github.com/p-org/P.

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 '16)*.

Doron Peled. 2018. Partial-Order Reduction. In *Handbook of Model Checking*. Springer, 173–190.

Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. 2001. Automatic Deductive Verification with Invisible Invariants. In *Proceedings of TACAS*. Springer.

Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. 2020. Symbolic Partial-Order Execution for Testing Multi-Threaded Programs. In *CAV (1) (Lecture Notes in Computer Science, Vol. 12224)*. Springer, 376–400.

Koushik Sen and Gul Agha. 2006a. Automated Systematic Testing of Open Distributed Programs. In *FASE (Lecture Notes in Computer Science, Vol. 3922)*. Springer, 339–356.

Koushik Sen and Gul Agha. 2006b. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV (Lecture Notes in Computer Science, Vol. 4144)*. Springer, 419–423.

Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *ESEC/SIGSOFT FSE*. ACM, 842–853.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proceedings of ACM Programming Languages* 2, POPL (2018), 28:1–28:30.

Sanjit A. Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating Modeling, Verification, Synthesis and Learning. In *MEMOCODE*. IEEE, 1–10.

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 *FMOODS/FORTE (Lecture Notes in Computer Science, Vol. 7273)*. Springer, 219–234.

TLA$^+$. 2023. TLA$^+$ Examples. https://github.com/tlaplus/Examples

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*. ACM, 530–541.

Willem Visser and Peter C. Mehlitz. 2005. Model Checking Programs with Java PathFinder. In *Proceedings of SPIN*.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Pierre Wolper and Patrice Godefroid. 1993. Partial-Order Methods for Temporal Verification. In *CONCUR (Lecture Notes in Computer Science, Vol. 715)*. Springer, 233–246.

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*. USENIX Association, 405–421.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 54–66.

A PROOF OF CANONICITY FOR VALUE SUMMARIES

Theorem A.1 (Canonicity of Primitive Value Summaries). *If a canonical representation is used for propositional formulas and for value components of guarded values, then for any two primitive value summaries pvs_0 and pvs_1, $pvs_0 \equiv pvs_1$ iff $pvs_0 = pvs_1$.*

Proof. That equivalence implies equality follows from the unique value and non-vacuous properties. The intuition is that, for each $(g, v) \in pvs_0$, because of the non-vacuous property and $pvs_0 \equiv pvs_1$, there must be $(g_1, v) \in pvs_1$. We know from the unique value property that this is the only guarded value in pvs_1 with value v. It follows that $g \Leftrightarrow g_1$. Because we assume a canonical representation for propositional formulas, we know that $g = g_1$. Thus, for each $(g, v) \in pvs_0$, $(g, v) \in pvs_1$, and, by symmetry, for each $(g, v) \in pvs_1$, $(g, v) \in pvs_0$. We conclude that the sets pvs_0 and pvs_1 contain the same elements and thus are equal. That equality implies equivalence follows from the definition of equivalence. □

Theorem A.2 (Canonicity of Value Summaries). *If a canonical representation is used for propositional formulas and for value components of guarded values, then all value summaries are such that, for any two value summaries vs_0, vs_1, $vs_0 \equiv vs_1$ iff $vs_0 = vs_1$.*

Proof. That equality implies equivalence follows from the definition of equivalence. We proceed by structural induction to show that equivalence implies equality. Note that from the definition of equivalence, we have that the equivalence of composite value summaries implies equivalence of their components.

Primitive Value Summaries. Handled above.

Tuples. Assume that we have two equivalent tuple value summaries $(vs_0, \ldots, vs_n)$ and $(ws_0, \ldots, ws_k)$. Our induction hypothesis as equivalence between any two nested value summaries vs_i and ws_j ($i \in \{0, \ldots, n\}$, $j \in \{0, \ldots k\}$) implies their equality. For these to represent the same sets of concrete tuples under the same guards, we must have that:

- $n = k$
- For each $i \in \{0, \ldots n\}$, we must have that $vs_i \equiv ws_i$. From induction hypothesis, it follows that $vs_i = ws_i$.

Since $(vs_0, \ldots, vs_n)$ and $(ws_0, \ldots, ws_k)$ have the same number of components and are component-wise equal, they are equal.

Lists. Assume that we have two equivalent list value summaries ls_0 and ls_1. Either they are both $(\{(g, 0)\}, [])$, in which case they are trivially equal, or they are of the form $(s_x, 0 :: xs)$ and $(s_y, y :: ys)$, respectively. Our induction hypothesis is that if s_x and s_y are equivalent, then they are equal and that for any $x \in xs$ and any $y \in ys$, if x an y are equivalent, then they are equal. We now need to show that ls_0 and ls_1 are equal. We have that s_x and s_y are equivalent, and that the lists xs and ys are element-wise equivalent. From the induction hypothesis, s_x and s_y are equal and that the lists are are element-wise equal as well.

Maps. Assume that we have two equivalent map value summaries mp_0 and mp_1. Our induction hypothesis is that the value summaries in the range of m are such that equivalence of any two of them implies equality. From the equivalence of m_0 and m_1, we know that m_0 and m_1 are such that $t \mapsto vs_0 \in m_0$ iff $t \mapsto vs_1 \in m_1$ for some semantically equal vs_0, vs_1. We know from the induction hypothesis that all such vs_0, vs_1 are syntactically equal. It follows that the maps mp_0 and mp_1 are also syntactically equal. □

B LIFTING OF COMMON COMPOSITE DATA STRUCTURES AND OPERATIONS

The P language contains several types that may be used in handlers: primitive types, tuples, lists, sets, and maps. We will outline how we choose to represent these here, and give some liftings of common operations over them as operational semantics for P expressions, and, where relevant (i.e., for assignments), P statements. The P semantics also refers to configurations and messages, which can both be represented using these types.

B.1 Primitives

The P primitives include numerical values, Boolean values, and enums, all of which we model with primitive value summaries. Events $\mathcal{E}$ and machine identifiers $\mathcal{I}$ are also lifted to primitive value summaries. Formally, we have that $\ell(p) = \{(\top, p)\}$ and $Vals(pvs) = \{v \mid \exists g.(g, v) \in pvs\}$. Updates are performed using the *updateUnderG* operation defined previously.

Unary operations. The following rule shows the lifted semantics for a unary operation *op* that can be applied directly on concrete values $op(v)$:

$$\textsc{Unary}\quad \frac{Int = \{(g', op(v)) \mid (g', v) \in pvs\}}{g, loc_c[id] \vdash op(pvs) \downarrow \{(g'', v) \mid \forall (g', v') \in Int.v' = v \Rightarrow (g'' \Rightarrow g')\}}$$

Binary operations. The following rule shows the lifted semantics for a binary operation *binop* that can be applied directly on concrete values $binop(v_0, v_1)$ (note that this includes the semantics for checking equality of two primitive value summaries):

$$\textsc{Binary-Empty}\quad g, L_c[id] \vdash binop(\{\}, pvs_1) \downarrow \{\}$$

$$\textsc{Binary-via-Unary}\quad \frac{g, L_c[id] \vdash (\lambda v_1.binop(v_0, v_1)(pvs_1) \downarrow res}{g, L_c[id] \vdash binop(\{(g_0, v_0)\}, pvs_1) \downarrow res}$$

$$\textsc{Binary}\quad \frac{g, L_c[id] \vdash binop(\{(g_0, v_0)\}, pvs_1) \downarrow (g'_0, v'_0) \ \ldots \ g, L_c[id] \vdash binop(\{(g_n, v_n)\}, pvs_1) \downarrow (g'_n, v'_n) \quad Int = \{(g'_0, v'_0), \ldots, (g'_n, v'_n)\}}{g, L_c[id] \vdash binop(\{(g_0, v_0), \ldots, (g_n, v_n)\}, pvs_1) \downarrow \{(g'', v) \mid \forall (g', v') \in Int.v' = v \Rightarrow (g'' \Rightarrow g')\}}$$

B.2 Tuples

Tuples are encoded as tuples of value summaries. A tuple $(v_0, \ldots, v_n)$ can be lifted to a tuple $(vs_0, \ldots, vs_n)$, where each $vs_i = \ell(v_i)$. The *Vals* function is then given by the following:

$$Vals(vs) = \{(v_0, \ldots, v_n) \mid \exists g. \forall i \in \{0, \ldots, n\}. \{v_i\} = Values((vs_i|g))\}$$

An invariant for the tuple representation $(vs_0, \ldots, vs_n)$ is that for all $i, j \in \{0, \ldots, n\}$, $D(vs_i) = D(vs_j)$, i.e., all entries of the tuple must be defined for the same guards.

Indexing. Getting the i^{th} element of a tuple for a concrete i can be implemented by getting the i^{th} element of its value summary. Note that if we used a primitive value summary representation, we would need to extract the i^{th} element of each tuple represented by the value summary, then merge the results.

$$\textsc{Tuple-Index}\quad \frac{0 \leq i \leq n}{g, L_c[id] \vdash (vs_0, \ldots, vs_n).i \downarrow vs_i}$$

Updating. Setting the value of an element of a tuple to a new value vs_{new} can similarly be performed by setting the i^{th} element of the tuple value summary t to $updateUnderG(t.i, vs_{new}, g \wedge D(vs_{new}))$, where g is the guard under which the update is performed.

$$\textsc{Tuple-Update}\quad \frac{0 \le i \le n \qquad L_c[id][t] = (vs_0, \ldots, vs_n) \qquad vs_i' = updateUnderG(vs.i, vs_{new}, g \wedge D(vs_{new})) \qquad c' = (L_c[id][t \mapsto (vs_0, \ldots, vs_{i-1}, vs_i', vs_{i+1}, \ldots, vs_n)], B_c, SO_c)}{g, id \vdash (t.i = vs_{new}, c) \rightarrow (\textbf{skip}, c')}$$

Pairwise. Pairwise applications of binary operations *binop* (including equality) to tuples have the following semantics:

$$\textsc{Pairwise-Tuple}\quad \frac{g, L_c[id] \vdash binop(vs_0, vs_0') \downarrow vs_0'' \qquad \ldots \qquad g, L_c[id] \vdash binop(vs_n, vs_n') \downarrow vs_n''}{g, L_c[id] \vdash binop((vs_0, \ldots, vs_n), (vs_0', \ldots, vs_n')) \downarrow (vs_0'', \ldots, vs_n'')}$$

B.3 Lists

We encode a list xs as a pair (s, xs) of an integer value summary $s = \ell(size(xs))$ representing the size of the list and a list ls ($ls[i] = \ell(xs[i])$) of value summaries representing the list element values. The *Vals* function is then given by the following:

$$Vals((s, xs)) = \{\{x_0, \ldots, x_n\} \mid \exists g, n. \forall i \in \{0, \ldots, n\}. \{x_i\} = Vals((xs[i] | g))\}$$

An invariant for the list representation (s, ls) is that there is an element $ls[i]$ iff there is a $(g, k) \in s$ with $k > i$ and $D(ls[i]) \Leftrightarrow \bigvee\{g \mid (g, k) \in s, k > i\}$. Maintaining the size of the list separately allows the domain for the the size value summary to determine the domain for the list. It additionally allows for efficient list size queries.

Size. Since we store the current size of the list in the representation, the *size* function that returns the value of the list can be implemented by just returning the first element of the pair.

$$\textsc{List-Size}\quad g, loc_c[id] \vdash size(p, ls) \downarrow p$$

Cons. Cons (::) can be lifted to value summaries as follows:

$$\textsc{List-Cons-Empty}\quad \frac{g, L_c[id] \vdash p + \ell(1) \downarrow p_{new} \qquad p' = updateUnderG(p, p_{new}, g \wedge D(vs))}{g, L_c[id] \vdash vs ::_\ell (p, []) \downarrow (p', [(vs|g)])}$$

$$\textsc{List-Cons-Nonempty}\quad \frac{g, L_c[id] \vdash p + \ell(1) \downarrow p_{new} \qquad p' = updateUnderG(p, p_{new}, g \wedge D(vs)) \qquad x' = updateUnderG(x, vs, g \wedge D(vs)) \qquad xs' = M((x :: xs | g \wedge D(vs)), (xs | \neg g \vee \neg D(vs)))}{g, L_c[id] \vdash vs ::_\ell (p, x :: xs) \downarrow (p', x' :: xs')}$$

Indexing. Getting the i^{th} element from a list value summary (p, ls) for a concrete i can be implemented by getting the i^{th} element of ls. If the index to get is in the form of an integer value summary vs_i, then for each (g, i), $ls[i]$ should be gotten, restricted with guard g, and added to a set. The merge of all the value summaries in the resulting set gives the result of the *get* operation:

$$\textsc{Get-Helper-Concrete-Singleton}\quad g, L_c[id] \vdash x :: xs[0] \downarrow x$$

$$\textsc{Get-Helper-Concrete}\quad \frac{i > 0 \qquad g, L_c[id] \vdash xs[i-1] \downarrow res}{g, L_c[id] \vdash x :: xs[i] \downarrow res}$$

$$\textsc{Get-Helper-Base}\quad \frac{getGuard = g \wedge getGuardFor(i, 0) \qquad g' = g \wedge \neg getGuard \qquad g' \Rightarrow\!\!\!\!\!/\ \bot}{g, L_c[id] \vdash x :: xs[vs_i] \downarrow (x|getGuard)}$$

$$\textsc{Get-Helper}\quad \frac{\begin{array}{c} getGuard = g \wedge getGuardFor(i, 0) \\ g' = g \wedge \neg getGuard \qquad g' \Rightarrow\!\!\!\!\!/\ \bot \qquad g', L_c[id] \vdash vs_i - 1 \downarrow i' \qquad g', L_c[id] \vdash xs[i'] \downarrow res \end{array}}{g, L_c[id] \vdash x :: xs[vs_i] \downarrow M((x|getGuard), (res|g'))}$$

$$\textsc{Get}\quad \frac{g, L_c[id] \vdash xs[vs_i] \downarrow res}{g, L_c[id] \vdash (p, xs)[vs_i] \downarrow res}$$

Update. Setting an element of a list to a value summary value works similarly as for tuples. Here we introduce a list setting *expression* shown in rules Set-Base and Set, that is used to handle the assignment to a particular index in List-Update.

$$\textsc{Set-Helper-Base}\quad \frac{\begin{array}{c} getGuard = g \wedge getGuardFor(0, vs) \\ g' = g \wedge \neg getGuard \qquad g' \Rightarrow \bot \qquad x' = updateUnderG(x, vs, getGuard) \end{array}}{g, L_c[id] \vdash x :: xs[vs_i \leftarrow vs] \downarrow x' :: xs}$$

$$\textsc{Set-Helper}\quad \frac{\begin{array}{c} getGuard = g \wedge getGuardFor(0, vs) \qquad g' = g \wedge \neg getGuard \qquad g' \Rightarrow\!\!\!\!\!/\ \bot \\ x' = updateUnderG(x, vs, getGuard) \qquad g', L_c[id] \vdash vs_i - 1 \downarrow vs_i' \qquad g', id \vdash xs[vs_i'] = vs \downarrow xs' \end{array}}{g, L_c[id] \vdash x :: xs[vs_i \leftarrow vs] \downarrow x' :: xs'}$$

$$\textsc{Set}\quad \frac{g, L_c[id] \vdash xs[vs_i \leftarrow vs] \downarrow xs'}{g, L_c[id] \vdash (p, xs)[vs_i \leftarrow vs] \downarrow (p, xs')}$$

$$\textsc{List-Update}\quad \frac{g, L_c[id] \vdash L_c[id][ls][vs_i \leftarrow vs] \downarrow res \qquad c' = (L_c[id][ls \mapsto res], B_c, SO_c)}{g, [id] \vdash (ls[vs_i] = vs, c) \rightarrow (\textbf{skip}, c'))}$$

Equality. Equality of lists is checked as follows:

$$\textsc{Equality-List-Empty}\ \frac{g, L_c[id] \vdash p_x = p_y \downarrow equalSize}{g, L_c[id] \vdash ((p_x, []) = (p_y, [])) \downarrow equalSize} \qquad \textsc{Equality-List-Helper-Empty}\ g, L_c[id] \vdash [] = [] \downarrow \{(g, \top)\}$$

$$\textsc{Equality-List-Helper-Empty-1}\ \frac{ys \neq []}{g, L_c[id] \vdash [] = ys \downarrow \{(g, \bot)\}} \qquad \textsc{Equality-List-Helper-Empty-2}\ \frac{xs \neq []}{g, L_c[id] \vdash xs = [] \downarrow \{(g, \bot)\}}$$

$$\textsc{Equality-List-Helper}\ \frac{g, L_c[id] \vdash x = y \downarrow equalElt \quad g, L_c[id] \vdash xs = ys \downarrow equalElts \qquad g, L_c[id] \vdash equalElt \wedge equalElts \downarrow res}{g, L_c[id] \vdash x :: xs = y :: ys \downarrow res}$$

$$\textsc{Equality-List}\ \frac{g, L_c[id] \vdash p_x = p_y \downarrow equalSize \quad g, L_c[id] \vdash xs = ys \downarrow equalElts \qquad g, L_c[id] \vdash equalSize \wedge equalElts \downarrow res}{g, L_c[id] \vdash (p_x, x :: xs) = (p_y, y :: ys) \downarrow res}$$

GetIndex and Containment. Getting the index of an element can be done by iterating over the indices of the list. The element at each index is checked for equality to the desired value, and if it is equal, that index is the solution. The current index *idx* is maintained during iterations. Under the conditions (i.e., guard) that the index has not been found, the process continues:

$$\textsc{Get-Idx-Helper-Empty}\ g, L_c[id] \vdash getIdxHelper([], vs, idx) \downarrow \{(g, \bot)\}$$

$$\textsc{Get-Idx-Helper-Base}\ \frac{g, L_c[id] \vdash x = vs \downarrow res \qquad g' = g \wedge getGuardFor(res, \bot) \qquad g' \Longrightarrow \bot}{g, L_c[id] \vdash getIdxHelper(x :: xs, vs, idx) \downarrow (idx | g \wedge getGuardFor(res, \top))}$$

$$\textsc{Get-Idx-Helper}\ \frac{\begin{array}{c} g, L_c[id] \vdash x = vs \downarrow contained \\ g' = g \wedge getGuardFor(contained, \bot) \qquad g' \Rightarrow\!\!\!\!\!\!\bot \qquad g', L_c[id] \vdash idx + 1 \downarrow idx' \\ g', L_c[id] \vdash getIdxHelper(xs, vs, idx') \downarrow intRes \qquad res = M(intRes, (idx | g \wedge getGuardFor(res, \top))) \end{array}}{g, L_c[id] \vdash getIdxHelper(x :: xs, vs, idx) \downarrow res}$$

$$\textsc{Get-Idx}\ \frac{g, L_c[id] \vdash getIdxHelper(xs, vs, \{(g, 0)\}) \downarrow res}{g, L_c[id] \vdash getIdx((p, xs), vs) \downarrow res}$$

Getting whether or not a list contains an element is performed similarly, but no extra *idx* value needs to be mtaintained:

$$\textsc{Contains-Helper-Empty}\quad g, L_c[id] \vdash contains([], vs) \downarrow \{(g, \bot)\}$$

$$\textsc{Contains-Helper-Base}\quad \frac{g, L_c[id] \vdash x = vs \downarrow res \qquad g' = g \wedge getGuardFor(contained, \bot) \qquad g' \Rightarrow \bot}{g, L_c[id] \vdash contains(x :: xs, vs) \downarrow res}$$

$$\textsc{Contains-Helper}\quad \frac{\begin{array}{c} g, L_c[id] \vdash x = vs \downarrow contained \qquad g' = g \wedge getGuardFor(contained, \bot) \\ g' \nRightarrow \bot \qquad g', L_c[id] \vdash contains(xs, vs) \downarrow contained' \qquad g, L_c[id] \vdash contained \vee contained' \downarrow res \end{array}}{g, L_c[id] \vdash contains(x :: xs, vs) \downarrow res}$$

$$\textsc{Contains}\quad \frac{g, L_c[id] \vdash contains(xs, vs) \downarrow res}{g, L_c[id] \vdash contains((p, xs), vs) \downarrow res}$$

The containment check can be performed by first getting index of an element, and then returning the following Boolean value summary, where *res* is the result of getting the index of the element:

$$\{(D(res), \top), (\neg D, \bot)\}$$

B.4 Sets

Set value summaries can implemented using the same representation as list value summaries (e.g., containment), but additional invariants must be maintained for the representation (s, ls) to guarantee that the representation is canonical.

Canonicity can be maintained by making sure that (1) every addition of an element to the set is done only under guards for which the underlying set does not already contain the element, and (2) elements are added so that the list is sorted according to some total order on its elements. This corresponds to a standard implementation of sets on top of lists, but instead using lifted lists and lifted list operations.

B.5 Maps

We can encode a map $mp : \tau_0 \rightarrow \tau_1$ as a map mp_{vs} from τ_0 to $VS(\tau_1)$ as is mentioned in the paper. However, in practice we choose to encode map $mp : \tau_0 \rightarrow \tau_1$ as a pair $(\ell(keys(m)), mp_{vs})$, where mp_{vs} is a map from τ_0 to $VS(\tau_1)$. The first element of the pair gives key set under different guards, and the map mp_{vs} maps concrete values within the key set to their corresponding value summary values. Each key is stored as a primitive value summary. Because of this, if keys are composite data structures, there may be some resulting redundancy that we cannot recognize, but in practice, keys tend to be primitive data structures that are naturally represented using primitive value summaries.

An invariant for the map representation (ks, mp_{vs}) is that if $contains(ks, k)$ contains the value **true** under guard g, then $g = \{D(mp_{vs}[key]) \mid key \in Values(k)\}$.

Get. Getting an element from a map involves getting all the value summaries for the all possible key values in the provided key value summaries (recall that keys are primitive value summaries, so no additional computation is needed for this) and merging them under the appropriate guards:

$$\textsc{Map-Get}\quad \frac{\{(g_0, k_0), \ldots, (g_n, k_n)\} \qquad res = M((mp[k_0]|g_0), \ldots, (mp[k_n]|g_n))}{g, L_c[id] \vdash (keys, mp)[vs_k] \downarrow res}$$

where $M(x_0, \ldots, x_n) = M(x_0, M(x_1, M(x_2, \ldots, x_n) \cdots)$.

Update. The lifted update of a map $map[vs_k] \leftarrow vs_v$ can be performed by making the following assignment for each $(g, k) \in vs_k$:

$$mp_v[k] \leftarrow updateUnderG(mp_v[k], vs_v, g \wedge D(vs_v))$$

where mp_v is the second component of the value of $map[vs_k]$ in the current configuration:

$$\textsc{Map-Update}\quad \frac{\begin{array}{c}(oldKs, oldMap) = L_c[map] \\ newKs = \{k \mid (g, k) \in vs_k\} \qquad \forall k \in oldKs \setminus newKs. L_c'[map].2[k] = oldMap[k] \\ \forall k \in newKs. L_c'[map].2[k] = updateUnderG(oldMap[k], vs_v, g \wedge D(vs_v)) \\ g, L_c[id] \vdash oldKs \cup vs_k \downarrow L_c[map].1\end{array}}{g, id \vdash (map[vs_k] \leftarrow vs_v, c) \rightarrow (\textbf{skip}, (L_c', B_c, SO_c))}$$

Equality. Equality of maps can be checked by making sure the domains are the same (using the set equality check) and then making sure that keys map to the same values (using the equality check for those value summaries):

$$\textsc{Map-Equality}\quad \frac{\begin{array}{c}g, L[id] \vdash keys = keys' \downarrow domEq \qquad \{k_0, \ldots, k_n\} = Domain(mp) \cap Domain(mp') \\ g, L[id] \vdash (mp[k_0], \ldots, mp[k_n]) = (mp'[k_0], \ldots, mp'[k_n]) \downarrow valEq \\ g, L[id] \vdash domEq \wedge valEq \downarrow res\end{array}}{g, L[id] \vdash (keys, mp) = (keys', mp') \downarrow res}$$

C LIFTED SEMANTICS FOR P

Fig. 8 gives the remaining rules for the lifted semantics for P, including rules for the top-level schedule step, statements, and the choose expression. Expressions inside event handlers, which may contain data-structure-specific operations, as well as assignments for composite data structures, have lifted semantics as described above.

Schedule Step

$$\frac{\begin{array}{c} choices = \ell(\lambda x.\{y \mid \texttt{enabled}(y,x)\})(c) \qquad g = \phi \wedge \bigvee\{D(choice) \mid choice \in choices\} \\ g, \{\} \vdash \textbf{choose}\ choices \downarrow m \qquad c'' = (L_c, B_c - \{m\}, SO_c) \\ \{g_0, \ldots, g_n\} = \{\phi \mid \exists t, ev.\phi = getGuardFor(m.tgt, t) \wedge getGuardFor(m.ev, ev)\} \\ \forall 0 \le i \le n.m_i = (m|g_i) \wedge H[L_c[m.tgt_i], m_i.ev] = h_i \\ g_0, m_0.tgt \vdash (h_0(m_0), c'') \rightarrow^* (\textbf{skip}, c'_0)\ \ldots\ g_n, m_n.tgt \vdash (h_n(m_n), c'_{n-1}) \rightarrow^* (\textbf{skip}, c'_n) \\ c' = (c'_n | D(m)) \end{array}}{H, \phi \vdash c \xRightarrow{m} c'}$$

Sequence

$$\frac{g, id \vdash (S_0, c) \rightarrow^* (\textbf{skip}, c') \qquad g, id \vdash (S_1, c') \rightarrow^* (\textbf{skip}, c'')}{g, id \vdash (S_0; S_1, c) \rightarrow (\textbf{skip}, c'')}$$

Assign-Var

$$\frac{\begin{array}{c} L_c[id] \vdash e \downarrow v \\ c' = (L_c[id][x \mapsto updateUnderG(x, v, g)], B_c, SO_c) \end{array}}{g, id \vdash (x := e, c) \rightarrow (\textbf{skip}, c')}$$

Send

$$\frac{\begin{array}{c} m = ((id, ev, v, tid, \texttt{fresh}(mid, c))|g) \qquad c' = (L_c, B_c \cup \{m\}, SO_{c'}) \\ SO_{c'} = SO_c \cup \ell(\lambda x, y.\{(m', x) \mid m'.src = x.id \wedge m' \in y\})(m, (B_c|g)) \end{array}}{g, id \vdash (\textbf{send}(tid, ev, v), c) \rightarrow (\textbf{skip}, c')}$$

New

$$\frac{\begin{array}{c} id' = \texttt{newInstance}(machineType, c) \\ c' = (updateUnderG(L_c, id', defaultLocals(id'), g), B_c, SO_c) \\ g, id \vdash (\textbf{send}(id', \text{init}, v), c') \rightarrow^* (\textbf{skip}, c'') \end{array}}{g, id \vdash (\textbf{new}\ machineType(v), c) \rightarrow (\textbf{skip}, c')}$$

Raise

$$\frac{m = (id, ev, v, id, \texttt{fresh}(mid, c)) \qquad handlers, g \vdash c \xrightarrow{m} c'}{g, id \vdash (\textbf{raise}\ ev(v), c) \rightarrow (\textbf{skip}, c')}$$

If

$$\frac{\begin{array}{c} L_c[id], g \vdash e \downarrow v \qquad g_0 = g \wedge getGuardFor(v, \top) \\ g_1 = g \wedge getGuardFor(v, \bot) \qquad g_0, id \vdash (S_0, c) \rightarrow^* (\textbf{skip}, c') \\ g_1, id \vdash (S_1, c') \rightarrow^* (\textbf{skip}, c'') \end{array}}{g, id \vdash (\textbf{if}\ e\ \textbf{then}\ S_0\ \textbf{else}\ S_1, c) \rightarrow (\textbf{skip}, c'')}$$

Choose

$$\frac{\begin{array}{c} B = getFreshGuardVariables(\lceil \log(|V|) \rceil) \\ V = \{v_1, \ldots, v_{|V|}\} \\ v = M(\{(v_i | encodeUsing(i, B, |V|))\}_{i=1..|V|}) \end{array}}{g, L_c[id] \vdash \textbf{choose}\ V \downarrow (v|g)}$$

Fig. 8. Value summary semantics for P programs

D POR REDUCTION FOR P: PRED

Here we describe PRED, a sound partial-order reduction for P programs. Our motivation for PRED comes from the well-known approach of persistent sets [Wolper and Godefroid 1993] and how the send-order relation used for defining P macro-step semantics relates to the independence of transitions. We achieve PRED reduction by restricting the set of enabled transitions at each configuration to explore only those transitions that form a persistent set [Wolper and Godefroid 1993].

A transition t_m for a message m refers to the execution of $m.tgt$'s event handler on $m.v$. The set of all possible transitions for a system is given by the set $\mathcal{T}$. We overload enabled and let $\texttt{enabled}(c, t_m) = \texttt{enabled}(c, m)$.

Independence of transitions. We use the notion of independence of transitions similar to that defined in the TransDPOR reduction for actor systems [Tasharofi et al. 2012].

Definition D.1 (Dependence). Two transitions $t_m,\ t_{m'} \in \mathcal{T}$ are *independent* iff for all configurations $c \in C$:

- If t_m is enabled in c and $(c, c') \in t_m$, then $t_{m'}$ is enabled in c' iff it is enabled in c (i.e., independent transitions cannot disable or enable eachother). Formally, $\texttt{enabled}(c, t_{m'}) \Leftrightarrow \texttt{enabled}(c', t_{m'})$.
- If t_m and $t_{m'}$ are both enabled in c, then there exists a unique configuration c' such that $c \overset{m}{\Longrightarrow} k \overset{m'}{\Longrightarrow} c'$ and $c \overset{m'}{\Longrightarrow} k' \overset{m}{\Longrightarrow} c'$ for some configurations k, k' (i.e., enabled independent transitions commute). Formally, $\texttt{enabled}(c, t_m) \wedge \texttt{enabled}(c, t_{m'}) \Rightarrow t_{m'}(t_m(c)) = t_m(t_{m'}(c)) = c'$

Transitions that are not independent are *dependent*.

POR exploits independence to determine interleavings that are *Mazurkiewicz-equivalent* [Mazurkiewicz 1986], i.e., a class of interleavings that can be obtained by commuting adjacent independent actions. Exploring only one representative interleaving from each equivalence class is sound for checking safety properties, e.g., local assertions, or absence of data races and deadlocks.

D.1 Persistent-set-based reduction

Many POR algorithms [Wolper and Godefroid 1993] rely on computing persistent sets. Persistent sets give a set of transitions for a configuration such that it is sound to explore only interleavings of transitions in the persistent set [Wolper and Godefroid 1993]. In particular, all configurations reachable by executing transitions outside a persistent set do not interact with the persistent set, so it is sound not to explore interleavings of transitions within the persistent set with those outside of it.

Definition D.2 (Persistent Set). A set of enabled transitions $P_{c_i} \subseteq \mathcal{T}$ in a configuration c_i are *persistent* in c_i iff, for all executions starting from c_i in which only transitions $t_{m_i} \notin P_{c_i}$ are taken $(0 \leq i \leq n)$, $t_{m_{i+n}}$ is independent with all transitions in P_{c_i} [Wolper and Godefroid 1993].

Reduced P semantics. Motivated by the idea of persistent sets, we present a restricted semantics of P that explores only transitions that are within persistent sets for the current configuration. This semantics thus can be considered to be a sound reduction of the P macro-step semantics. The justification for the persistent sets is based on the P semantics, and thus the reduction achieved applies for all P programs.

This restricted semantics makes use of a version of the SO_c relation, augmented with additional causality information: The $<_c$ relation is the transitive closure of the relation $<_c = SO_c \cup \{m_0 \triangleleft m_1 \mid m_0, m_1 \in \mathcal{M}\}$, where any two message m_0, m_1 are such that $m_0 \triangleleft m_1$ if the event handler that

receives m_0 is responsible for sending m_1. This relation accounts for the order in which messages were sent by a given machine, including the send order $\triangleleft$ of any newly sent messages, and the order between messages sent at different steps.

For any three transitions t_m, $t_{m'}$, $t_{m''}$, we have that $t_{m''}$ is not enabled in configuration c if the following hold: **(1)** $t_m, t_{m'}$ enabled in c; **(2)** $t_{m'}$ sends message m'' (i.e., $m'' \in \texttt{sends}(c, t_{m'})$); **(3)** $m <_c m'$ (i.e., m was sent before m'); **(4)** t_m and $t_{m''}$ are not independent. From this, we can conclude that for $t_m, t_{m'}$ enabled in configuration c with $m <_c m'$, taking the transition for the "earlier" message m will not interfere with transitions for any messages sent as a consequence of taking transition $t_{m'}$.

In the restricted semantics, for each configuration c, we thus only explore only those transitions whose corresponding received messages are least according to the relation $<_c$. Note that these transitions do not interfere with the remaining enabled transitions in c. Let predicate $\texttt{sufficient}(c, t_m)$ enforce this restriction. It is defined as:

$$m \in B_c \wedge \forall m' \in B_c.m' \neq m \Rightarrow m' \not<_c m$$

As in the `enabled` predicate, we require that no other message m' in the global buffer was sent before m was sent, but unlike `enabled`, here we require this for *all* m' in the global buffer, rather than just m' sent to the same target as m. For a configuration c, the restricted set of enabled transitions that we use for reduction is given by the following set:

$$\{t_m \mid \forall t_m \in \mathcal{T}.\texttt{sufficient}(c, t_m)\}$$

Note that $\texttt{sufficient}(c, t_m)$ implies $\texttt{enabled}(c, t_m)$. By only choosing transitions in this restricted set we can achieve more efficient exhaustive exploration of P programs.

To illustrate the reduction achieved, we revisit the P program from Fig. 1 in the paper. Fig. 2 in the paper shows all executions allowed by the P macro-step semantics, where each path through the tree corresponds to an execution where each node is a configuration and each edge label is a step in the execution. The executions allowed by the restricted P semantics are shown by the full tree. From this example, we can see the benefits of using Pred to explore P programs; Pred allows fewer interleavings, leading to more efficient exploration of program behaviors. Further reductions can be achieved using additional partial-order reductions (see §6.2 of the paper).

D.2 Soundness of Pred

Theorem D.3 (Pred Computes Persistent Sets). *In* P *semantics, for any sequence of steps from an initial state E_c ending in configuration c, the set of transitions given by the following is a persistent set:*

$$\{t_m \mid \forall t_m \in \mathcal{T}.\texttt{sufficient}(c, t_m)\}$$

Proof. Let E_c, c be such an sequence of steps and configuration, and let *Persistent* denote the set of transitions that we need to show is a persistent set. Let $t_m \in$ *Persistent* be an arbitrary transition enabled in c. Consider an arbitrary nonempty sequence of steps S resulting from taking allowed transitions from $c = c_j$, where $t_{m_i} \notin Persistent_C$:

$$c_j \rightarrow_{m_j} c_{j+1} \rightarrow_{m_{j+1}} c_{j+2} \rightarrow_{m_{j+2}} \ldots \rightarrow_{m_{j+n}} c_{j+n+1}$$

To show that $Persistent_c$ is persistent, we need to show that $t_{m_{j+n}}$ and t_m are independent.

We will show the stronger property that $t_{m_{j+n}}$ and t_m are independent and t_m is enabled in c_{j+n} with $m <_{c_{j+i}} m_{j+i}$ for all $0 \leq i \leq n$.

By strong induction on n:

Base Case. For $n = 0$, from the definition of `sufficient` and `enabled`, we know that $m <_{c_j} m_j$ and that $m_j.tgt \neq m.tgt$, so t_{m_j} and t_m are independent. We also have from the definition of *Persistent* that t_m is enabled.

Induction Step. We assume the inductive hypothesis that for all $0 \leq k < n$, $t_{m_{j+k}}$ and t_m are independent and t_m is enabled in c_{j+k} with $m <_{c_{j+n}} m_{j+1}$ for all $0 \leq i \leq k$. From this inductive hypothesis, we know that t_m is enabled in c_{j+n} From E_c, we know that $t_{m_{j+n}}$ is also enabled in configuration c_{j+n}. There thus must be a $0 \leq i < n$ such that m_{j+n} was sent by $t_{m_{j+k}}$ in configuration c_{j+k}. From our inductive hypothesis, we have that that $m <_{c_{j+k}} m_{j+k}$. It follows that $m <_{c_{j+n}} m_{j+n}$ also holds. If t_m and $t_{m_{j+n}}$ have the same actor, then only one of them can be enabled in c_{j+n} as per the definition of `enabled`. Thus, we know that t_m and $t_{m_{j+n}}$ have different actors. Furthermore, we also know that neither m is sent by $t_{m_{j+n}}$ nor m_{j+n} is sent by t_m because both transitions are enabled at the same configuration, so t_m and $t_{m_{j+n}}$ are independent. □