Verifying Distributed Programs via Canonical Sequentialization

ALEXANDER BAKST, University of California, San Diego
KLAUS V. GLEISSENTHALL, University of California, San Diego
RAMI GÖKHAN KICI, University of California, San Diego
RANJIT JHALA, University of California, San Diego

We introduce canonical sequentialization, a new approach to verifying unbounded, asynchronous, message-passing programs at compile-time. Our approach builds upon the following observation: due the combinatorial explosion in complexity, programmers do not reason about their systems by case-splitting over all the possible execution orders. Instead, correct programs tend to be well-structured so that the programmer can reason about a small number of representative executions, which we call the program's *canonical sequentialization*. We have implemented our approach in a tool called Brisk that synthesizes canonical sequentializations for programs written in Haskell, and evaluated it on a wide variety of distributed systems including benchmarks from the literature and implementations of MapReduce, two-phase commit, and a version of the Disco distributed file-system. Brisk verifies *unbounded* versions of the benchmarks in tens of *milliseconds*, yielding the first concurrency verification tool that is fast enough to be integrated into a design-implement-check cycle.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program analysis**; **Distributed computing models**; *Concurrency*; • **Software and its engineering** → Software notations and tools;

Additional Key Words and Phrases: canonical sequentialization, asynchronous programs, concurrency, distributed programs, reductions, program verification, parameterized systems, message passing

ACM Reference Format:
Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. 2017. Verifying Distributed Programs via Canonical Sequentialization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 110 (October 2017), 27 pages.
https://doi.org/10.1145/3133934

1 INTRODUCTION

Concurrent and distributed message passing programs have remained viciously difficult to implement, as the programmer gets little feedback about the correctness of their system *at development time*. For classical single process applications, modern type systems and IDEs can provide instantaneous feedback about whether or not the individual parts compose correctly. In contrast, the distributed systems developer must painstakingly construct workloads and stress tests to tickle subtle concurrency errors like deadlocks without any guarantee that the tests actually cover all errors. Model checking [Desai et al. 2015; Killian et al. 2007; Yang et al. 2009] is helpful in systematically searching the space of executions for finding corner-case bugs. However, the exploration can take minutes or hours, and hence is only useful for post-facto validation and of little help during

2475-1421/2017/10-ART110
https://doi.org/10.1145/3133934

development. Worse, it is limited to *finite* systems: when the number of processes is *unbounded*, *e.g.*, if processes are spawned based on input parameters, model checking cannot guarantee correctness as the state space, and hence, the number of executions, is infinite. Consequently, the distributed systems developer is bereft of development- and compile-time tools that guarantee the absence of concurrency errors.

Our Approach In this paper, we propose *canonical sequentialization*, a new approach to verifying concurrency properties of unbounded, asynchronous, message-passing programs at compile-time. Our approach builds upon the following observation: due the combinatorial explosion in complexity, programmers do not reason about their systems by case-splitting over all the possible execution orders. Instead, correct programs tend to be *well-structured* so that the programmer can reason about a small number of representative executions which we call the program's *canonical sequentialization*.

Example: Two-Phase Commit Consider the classic two-phase commit protocol [Lampson and Sturgis 1976], which consists of a leader process trying to commit a transaction to a number of database nodes. The protocol proceeds in two phases. In the first phase, the leader issues a tentative transaction. The leader then waits for answers from the nodes who may decide to either accept or abort the transaction. This initiates the second phase: if all nodes agreed, the leader sends them a commit message; otherwise, it sends an abort message. Finally, each node sends its acknowledgement of the final decision.

Even though leader and database nodes execute in parallel, the concurrency is well-structured. First, due to the lack of a shared memory, most actions executed by different nodes are *independent* of each other, and thus commute. For example, a tentative proposal sent to one of the database nodes is independent of the messages other database nodes might send or receive. Thus, in the spirit of Lipton's movers [Lipton 1975], we can consider just the traces in which the proposal is received immediately after it is sent, effectively *moving* the receive to its matching send in any program trace.

However, this reasoning breaks down for the other parts of the protocol where there are *multiple* potentially matching sends, *e.g.*, when the leader is waiting to receive accept or abort messages from the database nodes. Our second crucial insight is that correct parameterized message passing programs often only contain *symmetric races*. For example, even though there is non-determinism with respect to which accept or abort message is received *first*, the states resulting from picking a particular winner are *symmetric i.e.*, they are *permutations* of each other [Norris IP and Dill 1996]. Thus, instead of reasoning about the original distributed program, we can reason about its canonical sequentialization which first delivers all tentative transactions to the database nodes in sequence, then receives all the replies, sends out the final decision and finally receives all acknowledgments.

Example: MapReduce As a second example, consider an implementation of MapReduce [Dean and Ghemawat 2004] which consists of (1) a number of *worker* processes that perform map/reduce tasks, (2) a *queue* process that distributes work, and (3) a *master* process that orchestrates the entire computation. Workers query the queue for assignments, perform the assigned task, and then send the results to the master; the queue waits for a request and answers with a work assignment; the master waits for results (see fig. 5.2).

Again, the apparently highly concurrent MapReduce protocol has a canonical sequentialization. All the races are symmetric: even though worker threads compete for work assignments, the states that result from picking a particular worker are equivalent modulo a shuffling of process identifiers. Similarly, even though the order in which results reach the master is non-deterministic, the resulting states are symmetric. Thus, instead of reasoning about the original distributed program, we can reason about its canonical sequentialization, which first sequentially assigns all tasks to the workers, and then passes the results to the master. We realize our approach via the following contributions.

1. Symmetric Non-Determinism & Canonical Sequentialization Our first contribution is to identify and formalize a property of message passing programs called *symmetric non-determinism* which serves as a pre-requisite for sequentialization (§ 2). We define a core language for message passing programs (§ 3) and use it to formalize canonical sequentialization as a set of *local rewriting rules* (§ 4). Each rewriting step produces a new program that consists of a *sequential prefix* and remainder term that still needs to be rewritten. We show that each rewrite preserves the halting states of all processes (§ 4.4). This allows us to use the sequentialization to not only prove local safety properties, but also *global* properties (*e.g.*, deadlock freedom) of the original program.

2. Synthesizing Sequentializations Our second contribution is to demonstrate that our rewriting rules can be turned into an method to *automatically synthesize* a canonical sequentialization from a symmetric non-deterministic program (§ 5.2). We use this synthesis algorithm to implement a distributed systems verification tool called Brisk[1] Brisk first compiles Haskell programs that use the Cloud Haskell library [Epstein et al. 2011] into our core language (§ 5.1). Brisk verifies that its input has only symmetric races, and computes its canonical sequentialization thereby checking absence of deadlocks and assertion failures.

3. Evaluation Our third contribution is an evaluation of our approach on a diverse range of benchmarks including distributed programs taken from the literature [Honda et al. 2008], a concurrent programming textbook [Marlow 2012], well known protocols such as two-phase-commit [Lampson and Sturgis 1976], MapReduce [Dean and Ghemawat 2004] and implementations of a key-value store, and a distributed file-system (§ 6). We show that unlike model checking, which gets prohibitively slow — *i.e.*, times out at one minute even with just 10 processes on our benchmark set — Brisk verifies the *unbounded* versions of the benchmarks in tens of *milliseconds*, yielding the first concurrency verification tool that is fast enough to be integrated into a design-implement-check cycle.

2 OVERVIEW

We start with an overview of how Brisk lets us write and verify a concurrent task distribution service in Haskell by synthesizing its canonical sequentialization (§ 2.1), then explain the main ideas underlying canonical sequentialization with a series of small examples (§ 2.2) and finally discuss its expressiveness (§ 2.3).

2.1 A Task Distribution Service

Figure 2.1 shows the implementation of a task distribution service. The program consists of n `clients`, each of which requests a task assignment from the server (line 12). Upon receiving the assignment (line 14), a client executes the assigned task and finally either sends an acknowledgment to the master (line 16), or fails in case no work was provided (line 18). The `server` uses the higher-order combinator `foldM` to repeat the function `serverLoop` once for each client. In each iteration, the server waits for a client to request a work item, (line 23), computes the next assignment (line 25), and finally sends the assignment to the client process (line 27). Finally, the `master` waits for acknowledgements from each client (line 34).

Verification Goals We want to show that (1) the program is deadlock free and (2) the clients never execute the fail statement in line 18. Even though these two properties seem obvious on inspection, proving them is far from trivial. First, since the program contains unboundedly many threads, a proof needs to track the number of processes that are yet to execute the sends in line 12 and 17 in order to ensure that, any time, there are still enough processes left to make sure that the receives in line 23 and 34 do not deadlock. Second and more problematically, the proof needs to reason about

[1] Brisk is available at `http://goto.ucsd.edu/~brisk`

```
1   import Brisk
2   data Msg = Request ProcessId | Ack
3   data Work = Task ProcessId Item | None
4
5   main n = do self ← getSelfPid
6               cs   ← spawnMany n (client self)
7               m    ← spawn (master cs)
8               server m cs
9
10  client sv = do self ← getSelfPid
11                 -- request a work item from server
12                 send sv (Request self)
13                 -- block until an item is assigned
14                 msg  ← receive
15                 case msg of
16                    Task m task → do process task
17                                     send m Ack
18                    None → fail
19  server m cs = foldM serverLoop () cs
20    where
21      serverLoop _ _ = do
22        -- wait for a request
23        Request p  ← receive
24        -- compute next item
25        item ← nextItem
26        -- send item to p
27        send p (Task m item)
28        return ()
29
30  master cs = foldM masterLoop () cs
31     where
32       masterLoop _ _ = do
33         -- wait for Ack
34         Ack ← receive
35         return ()
```

Fig. 2.1. A task distribution service.

the messages that are being sent and received by the processes, of which there may be unboundedly many. This difficulty is often avoided by reasoning about the *number* of messages sent rather than their *content* [Konnov et al. 2015].

This, however, is not enough for our example as the correctness of the program relies on 1) clients sending their own PID in line 12 as well as the server sending the master's PID in line 27 (if they sent some arbitrary value, the messages might vanish into the ether leaving their intended receives deadlocked) and 2) the server always sending a work item (if it sent None, the program would execute the failure branch in line 18). Thus, in general, reasoning about programs like our task service requires complex invariants that are *universally quantified* over the set of participating processes (*e.g.*, to track the contents of messages) and require auxiliary state (*e.g.*, to track how many processes have sent a certain type of message). Despite recent progress, automatic synthesis of such invariants remains a difficult challenge [Bjørner et al. 2013; Farzan et al. 2014; Gleissenthall et al. 2016], and thus, we are not aware of any automated verification method that an handle this simple example.

```
for c in cs do
     [Request p ← Request c]_server;
     [item ← nextItem]_server;
     [msg ← Task master item]_c;
     ⎡case msg of                   ⎤
     ⎢    Task m task → process task⎥
     ⎣    None → fail               ⎦_c
end ;
for c in cs do
     [Ack ← Ack]_master
end
```

Fig. 2.2. Canonical Sequentialization of the task distribution system in fig. 2.1 as computed by Brisk.

Verification via Canonical Sequentialization In this paper, we propose a new approach: rather than verifying the original distributed program, we synthesize and verify its canonical sequentialization. Writing a verified program in Brisk starts by importing the Brisk library (line 1) which provides primitives for sending and receiving messages (send and receive which are built on top of Cloud Haskell), process creation (spawnMany), and iterating over sets of processes (forM). When Brisk is invoked to verify the program, it compiles (§ 5.1) the higher-order Haskell source into a first-order core language called IceT in order to make explicit the control flow and process structure of the input program. Brisk then synthesizes the program's canonical sequentialization (§ 4) which we show in fig. 2.2. We use the notation $[s]_p$ to mean that process p executes statements s, we let

$\cdot;\cdot$ denote sequential composition, and use $\cdot \leftarrow \cdot$ for assignments. The canonical sequentialization consists of two `for`-loops over the clients cs. In each iteration of the first loop, the client issues a request, the server assigns a task to the respective client, and the client processes the tasks or fails. In the second loop, the master receives the clients' acknowledgements.

Correctness In the canonical sequentialization, proving both deadlock-freedom and safety becomes straightforward. As the sequentialization contains neither sends nor receives, the program cannot deadlock. Similarly, since the clients assign `Tasks` to `msg`, the failure branch cannot execute – a fact that can be easily proved (Brisk verifies this assertion automatically). Since our method guarantees that the original program and its canonical sequentialization are equivalent (or in general, the canonical sequentialization *over-approximates* the original program § 4.4), we can conclude that the original program is correct.

2.2 Main Ideas

Symmetric Nondeterminism The crux of our method, and thereby the reason why we can soundly represent a program through its canonical sequentialization, lies in the following observation: for any given trace, we can always move a receive up to its *matching send*. This transformation is an application of Lipton's theory of movers [Lipton 1975]. *Statically* moving a receive up to a matching send is only sound if either a *unique matching send* exists, or all matching sends are *symmetric*, *i.e.*, picking an arbitrary one will result in equivalent states up to permutations of PIDs. To make sure this requirement is met, Brisk checks that the program satisfies the following condition, which we call *symmetric non-determinism*: every receive in a given program location can only receive messages from either *i)* a single process, only, or *ii)* a set of symmetric processes (*i.e.*, processes running the same code) at the same program location.

$$\begin{bmatrix} \mathsf{send}(q, ping); \\ w \;\leftarrow \mathsf{recv}() \end{bmatrix}_p \parallel \begin{bmatrix} v \;\leftarrow \mathsf{recv}(); \\ \mathsf{send}(p, pong) \end{bmatrix}_q \qquad\qquad [v \leftarrow ping]_q; [w \leftarrow pong]_p$$

Fig. 2.3. Example ex1 (left) and its canonical sequentialization (right).

Example 1: Canonical Sequentialization Figure 2.3 shows program ex1 written in our core language IceT, in which two processes p and q exchange messages. Process p *asynchronously* sends a *ping* message to q and then waits for a reply. Process q waits for a message and, upon receipt, sends a *pong* message to p. Processes p and q are composed in parallel with the $\parallel$ operator. Since the sends and receives can only be executed in a *single order*, we can rewrite the above concurrent program into its canonical sequentialization shown next to ex1 in fig. 2.3. Crucially, both programs are equivalent in the sense that they terminate in the same state.

Example 2: Parametric Programs Next, consider program ex2 shown in fig. 2.4 which contains

$$\begin{bmatrix} \mathsf{for}\ q\ \mathsf{in}\ Q\ \mathsf{do} \\ \quad \mathsf{send}(q, ping); \\ \quad w \;\leftarrow \mathsf{recv}(q) \\ \mathsf{end} \end{bmatrix}_p \parallel \prod q{:}Q.\begin{bmatrix} v \;\leftarrow \mathsf{recv}(); \\ \mathsf{send}(p, pong) \end{bmatrix}_q \qquad\qquad \begin{array}{l} \mathsf{for}\ q\ \mathsf{in}\ Q\ \mathsf{do} \\ \quad [v \leftarrow ping]_q; \\ \quad [w \leftarrow pong]_p \\ \mathsf{end} \end{array}$$

Fig. 2.4. Example ex2 (left) and its canonical sequentialization (right).

an unbounded number of processes. In ex2 a single process p exchanges messages with a *set* of

$$\begin{array}{l}\text{for } q \text{ in } Q \text{ do}\\ \quad [v \leftarrow ping]_q;\\ \text{end};\end{array} \left[\boxed{\begin{array}{l}\text{for } q \text{ in } Q \text{ do}\\ \quad w \leftarrow \text{recv}()\\ \text{end}\end{array}}\right]_p \parallel \prod q{:}Q.\left[\boxed{\text{send}(p, pong)}\right]_q$$

$$\begin{array}{l}\text{for } q \text{ in } Q \text{ do } [v \leftarrow ping]_q \text{ end};\\ \text{for } q \text{ in } Q \text{ do } [w \leftarrow pong]_p \text{ end}\end{array}$$

Fig. 2.5. Intermediate rewriting step of example EX3 (left) and its canonical sequentialization (right).

processes Q that run the same program code. We call such processes *symmetric*. Process p executes a loop which iterates over all processes q in Q. For each q, it first sends a *ping* and subsequently waits for a reply *from* q. Each process in Q first waits for a message and, upon receipt, sends a *pong* message to p. We use $\prod$ to denote a parallel composition of symmetric processes. If we fix an iteration order over Q, the sends and receives are again constrained to execute in a single order. Thus, BRISK computes the canonical sequentialization shown right in fig. 2.4.

Example 3: Multiple Orders Next, consider fig. 2.6 which contains the program EX3 which allows for multiple execution orders. This program is a variant of EX2 where p's loop is split into two parts: first p sends out all *ping* messages, then it waits for the answers to arrive.

Different executions of EX3 may see messages sent and received in different orders. For example, *ping* messages sent by p can arrive at their respective processes in Q in any order, as messages to different processes may be transported at different speeds. By the same logic, p may receive *pong* messages in any order due to the speed of the underlying network or differences in execution times between members of Q.

$$\left[\begin{array}{l}\boxed{\begin{array}{ll}\text{for } q \text{ in } Q \text{ do} & \\ \quad \text{send}(q, ping) & a\\ \text{end}; & \end{array}}\\ \text{for } q \text{ in } Q \text{ do}\\ \quad w \leftarrow \text{recv}() \qquad \{b\}\\ \text{end}\end{array}\right]_p \parallel \prod q{:}Q.\left[\begin{array}{ll}\boxed{v \leftarrow \text{recv}();} & \{a\}\\ \text{send}(p, pong) & b\end{array}\right]_q$$

Fig. 2.6. Example EX3. Each send is annotated with a tag (red) and each receive is annotated with the *set* of all tags it can receive from (blue).

Checking Symmetric Non-Determinism

In order to rewrite EX3 into its canonical sequentialization, we first need to check that it satisfies symmetric non-determinism. For this, we annotate each syntactic occurrence of send with a unique *tag*. For each receive, we then compute an over-approximation of the set of tags it can receive from. BRISK computes these tags using a lightweight syntax-guided method (§ 5). Figure 2.6 shows send-tags in red and receive-tags in blue. In order to satisfy symmetric non-determinism, we require that each receive-set contains either *i)* only tags from a *single process*, or *ii)* a *single tag* from a process in set of *symmetric processes*. The tags for EX3 satisfy this requirement.

Moving Left: Eagerly Executing Receives We split the rewrite of EX3 into two steps. For the first step, consider a send to some process q in p's for-loop together with its matching receive in q. Since EX3 satisfies symmetric non-determinism, we know that there are no *additional* sends q could receive from. Moreover, the receive is *independent* of messages sent by p to other members of Q and of messages that other process might send to p. This means, we can eagerly execute it *directly after* its matching send in p without changing the behaviour of the program. We say that the receive can be *moved left*, up to its matching send. Applying the above reasoning, we can rewrite EX3 into the partially sequentialized program shown left in fig. 2.5. The program consists of a *sequential prefix* which is a rewriting of the boxed statements from fig. 2.6 and a *remainder term* that corresponds

$$\left[\boxed{\begin{array}{ll}\text{for } q \text{ in } Q \text{ do} & \\ \quad id \leftarrow \text{recv}(); & \{b\} \\ \quad \text{send}(id, ping) & a \\ \text{end} & \end{array}}\right]_p \parallel \prod q{:}Q.\left[\boxed{\begin{array}{ll} \text{send}(p,q); & b \\ v \leftarrow \text{recv}() & \{a\} \\ \text{send}(m, pong); & c \end{array}}\right]_q \parallel \left[\begin{array}{ll}\text{for } q \text{ in } Q \text{ do} & \\ \quad w \leftarrow \text{recv}() & \{c\} \\ \text{end} & \end{array}\right]_m$$

Fig. 2.7. Example ex4.

$$\begin{array}{l}\text{for } q \text{ in } Q \text{ do} \\ \quad [id \leftarrow q]_p; \\ \quad [v \leftarrow ping]_q \\ \text{end}; \end{array}$$
$$\prod q{:}Q.\left[\boxed{\text{send}(m, pong);}\right]_q \parallel \left[\boxed{\begin{array}{l}\text{for } q \text{ in } Q \text{ do} \\ \quad w \leftarrow \text{recv}() \\ \text{end}\end{array}}\right]_m$$

$$\begin{array}{lll}\text{for } q \text{ in } Q \text{ do} & \begin{array}{l}[id \leftarrow q]_p; \\ [v \leftarrow ping]_q\end{array} & \text{end}; \\ \text{for } q \text{ in } Q \text{ do} & [w \leftarrow pong]_m & \text{end}\end{array}$$

Fig. 2.8. ex4 after the first rewrite step (left) and its canonical sequentialization (right).

to the part of the program that still needs to be rewritten. For the second rewriting step, consider the boxed statements in fig. 2.5. In a given loop iteration, p can receive a send from *any* of the remaining processes in Q. However, all processes behave the same, and only differ in their PIDs. This means, moving the receive up to an *arbitrary* send will result in the same final state. As a result, we can rewrite the program shown left in fig. 2.5 into its canonical sequentialization shown on the right.

Example 4: Multi-Party Communication Finally fig. 2.7 shows example ex4 whose communication structure matches the task distribution service from fig. 2.1. Process p executes a loop in which it waits for a message, and upon receipt, sends a ping message to the process it received from. Each process in Q first sends its PID to process p, then sends a *pong* message to process m and finally waits for a reply. Process m executes a loop in which it receives messages from processes in Q. First, we check that ex4 is symmetrically non-deterministic. Figure 2.7 shows the tags which satisfy the requirements.

Next, we rewrite ex4 into its canonical sequentialization. The rewrite is split into two steps: a first step in which Brisk rewrites the communication between process p and set Q, and a second step in which it rewrites the communication between set Q and m. For the first step, Brisk rewrites the boxed statements in fig. 2.7. This rewrite step produces a sequential prefix shown left in fig. 2.9 and an additional *residual term* which contains the messages the processes in Q sent to m, shown to the right. In the second step, Brisk rewrites the parallel composition of the remainder program with the residual term, *i.e.*, the program shown left in fig. 2.8. Brisk rewrites the boxed statements in fig. 2.8 into the canonical sequentialization shown right in fig. 2.8.

$$\begin{array}{l}\text{for } q \text{ in } Q \text{ do} \\ \quad [id \leftarrow q]_p; \\ \quad [v \leftarrow ping]_q; \\ \text{end}\end{array} \qquad \prod q{:}Q.\left[\ \text{send}(m, pong)\ \right]_q$$

Fig. 2.9. Intermediate step in the rewrite of ex4. Prefix (left) and residual term (right).

$$
\mathrm{C} \triangleq \prod c{:}Cs. \left[\begin{array}{l}
\mathsf{if}\ *\ \mathsf{then} \\
\quad \mathsf{send}(s, \mathsf{Get}(k)) \\
\mathsf{else} \\
\quad \mathsf{send}(s, \mathsf{Set}(k, v)) \\
\mathsf{end}
\end{array}\right]_c
\quad
\mathrm{C'} \triangleq \prod c{:}Cs. \left[\begin{array}{l}
\mathsf{if}\ *\ \mathsf{then} \\
\quad msg \leftarrow \mathsf{Get}(k) \\
\mathsf{else} \\
\quad msg \leftarrow \mathsf{Set}(k, v) \\
\mathsf{end;} \\
\mathsf{send}(s, msg)
\end{array}\right]_c
\quad
\mathrm{S} \triangleq \left[\begin{array}{l}
\mathsf{while\ true\ do} \\
\quad op \leftarrow \mathsf{recv}(*); \\
\quad \mathsf{if}\ isGet(op)\ \mathsf{then} \\
\quad\quad \mathsf{doGetOp()} \\
\quad \mathsf{else} \\
\quad\quad \mathsf{doSetOp()} \\
\quad \mathsf{end} \\
\mathsf{end}
\end{array}\right]_s
$$

2.3 Expressiveness

Not all programs have a canonical sequentialization. We can characterize the systems where our method is applicable by describing its limits, *i.e.*, the four cases where Brisk *fails* to synthesize a sequentialization.

1. Asymmetric Non-determinism Brisk will reject programs if it cannot prove that they only exhibit symmetric non-determinism. In this case, Brisk outputs the sends and receives that are involved in the suspected asymmetric race. In our experience, this often is either a bug or easy to remedy by restructuring the receives. That said, there are algorithms that do break symmetry and hence cannot be sequentialized by Brisk, *e.g.*, process sets with asymmetric topology (§ 8).

Example Consider program C above, which is a set of clients of a key-value store S whose API supports retrieving the value of a key k with a $\mathsf{Get}(k)$ message, and setting the value v of a key k with a $\mathsf{Set}(k, v)$ message. Each $c \in Cs$ performs a non-deterministic choice (represented by the condition $*$) to either send a $\mathsf{Get}(k)$ or a $\mathsf{Set}(k, v)$ message to s. The composition C || S does not satisfy symmetric non-determinism, as there are *two* distinct sends of the same type to the store s. However, this is easily refactored to a program C' that first performs a non-deterministic *assignment* to a *msg* variable and then sends the contents of the variable to s. Hence, C' || S has symmetric non-determinism.

Example Next, consider a logging process that receives messages from *every* process in a system and logs the messages it receives to disk. Except in simple instances, programs including such a logger will have asymmetric non-determinism. In this example, there is no formula for refactoring the program into a variant with symmetric non-determinism.

2. Superfluous Sends The program must not contain any superfluous sends that *do not* have a matching receive. Brisk is unable to rewrite such programs into their sequentialization, but returns a counterexample in the form of a sequential prefix that ends in the superfluous send. While superfluous sends can be benign, (unlike superfluous receives which are deadlocks), they are dubious; we consider their detection to be a virtue of our approach.

3. Indiscriminate Communication When iterating over an (unbounded) set of processes Q, a process p must only talk to a *single process* in Q, in any iteration. Process p may however send messages to *other* processes not in Q. This is also not an onerous requirement as in well-structured programs, loops over unbounded sets Q are used primarily to "broadcast" or "gather" both of which are amenable to sequentialization.

4. Stateful Loop Termination Finally, while-loops that interact with other processes must not use loop-carried state to decide whether to loop again or exit. Sequentialization requires that the decision only depends on values computed in the *current iteration*. This requirement models a reactive pattern in which loop termination depends on external messages. We found that loops requiring loop-carried state can often be restructured into an iteration over sets (processes or otherwise) which our method supports.

$x, X \in$ Identifiers $\quad t \in$ MsgType

e	::=		**Expressions**
		c	*literal values*
	|	x	*variable*
	|	$f(\bar{e})$	*primitive operations*
w	::=		**Sender Specification**
		$*$	*any sender*
	|	e	*expression*
s	::=		**Statements**
		$x \leftarrow e$	*assignment*
	|	$x \leftarrow$ recv(w, t)	*receive typed message*
	|	send(t, e, e)	*send typed message*

P	::=		**Programs**
		skip	*empty process*
	|	$[s]_e$	*singleton process*
	|	$P; P$	*sequential composition*
	|	$P \parallel P$	*parallel composition*
	|	$\prod x{:}X.P$	*parallel iteration*
	|	for x in X do P end	*sequential iteration*
	|	if e then P else P	*branching*
	|	while true do P end	*unbounded iteration*
	|	break	*loop exit*
	|	$\sum x{:}X.P$	*nondeterministic value*

Fig. 3.1. Syntax of the IceT language.

Counterexamples Brisk provides useful feedback in each of the four cases of failure. When there is a (possible) asymmetric race, the programmer is pointed to the race condition that needs to be fixed. In the remaining cases, Brisk outputs the longest *sequential prefix* encountered in the failed rewrite attempt (together with the remaining, unsequentialized program) thereby pinpointing the exact conditions under which the relevant condition is violated. Since Brisk is fast enough (10s of milliseconds) to provide this feedback *during* development, we envision a use case where the coding discipline required by Brisk can nudge the developer towards well-structured programs that are easier to reason about for machines (and humans).

3 MESSAGE PASSING PROGRAMS

In this section, we present the syntax and semantics of IceT, a core language for representing message passing programs as in Erlang and Cloud Haskell [Epstein et al. 2011]. Figure 3.1 shows the syntax of IceT.

Processes Each process is associated with a unique process identifier (PID), which serves as an address for sending messages. We use $[s]_p$ to denote a single process with PID p executing statement s, and assume that distinct processes have disjoint variable sets. We let skip denote the empty process.

Programs Programs are obtained from single process statements through parallel and sequential composition. We allow grouping of consecutive statements of the same process, *i.e.*, for statements s_1 and s_2, we abbreviate $[s_1]_p; [s_2]_p$ to $[s_1; s_2]_p$. $\prod x{:}X.P$ denotes the *parallel* composition of all instantiations of P to values in X. Let $X = \{x_0, x_1, \ldots, x_k\}$ and let $t[u/x]$ denote the substitution (without capture) of term u for variable x in term t. We thus define $\prod x{:}X.P \triangleq P[x_0/x] \parallel P[x_1/x] \parallel \ldots \parallel P[x_k/x]$. Similarly, we use (for x in X do P end) to denote the *sequential* composition of all instantiations of P to values in X, *i.e.*, for $X = \{x_0, x_1, \ldots, x_k\}$ we get for x in X do P end $\triangleq P[x_0/x];\ P[x_1/x];\ \ldots;\ P[x_k/x]$. Finally, $\sum x{:}X.P$ nondeterministically chooses a value from X and then assigns x to that value in P.

Normal Forms We say that a program is in *normal form* if it consists of a parallel compositions of sequences of statements from *distinct processes*, and assume that input programs to our method satisfy this requirement. For two programs P and Q in normal form, we let $P \circ Q$ denote the result of sequencing Q after P, process-wise.

$$\textsc{R-Context}\quad \frac{\Gamma, \Delta, A, \Psi \rightsquigarrow \Gamma', \Delta', A', \Psi'}{\Gamma, \Delta, A \circ B, \Psi \rightsquigarrow \Gamma', \Delta', A' \circ B, \Psi'} \qquad \textsc{R-Congruence}\quad \frac{A \equiv B}{\Gamma, \Delta, A, \Psi \rightsquigarrow \Gamma, \Delta, B, \Psi}$$

$$\textsc{R-Send}\quad \frac{\begin{array}{ll} \Delta \models x = q & \\ q \text{ is a PID} & \Gamma(p, q, t) = m \\ \Gamma \not\models \mathsf{E}(q) & \Gamma' = \Gamma[(p, q, t) \leftarrow m \cdot n] \end{array}}{\Gamma, \Delta, [\mathsf{send}(t, x, n)]_p, \Psi \rightsquigarrow \Gamma', \Delta, \mathsf{skip}, \Psi} \qquad \textsc{R-Recv}\quad \frac{\begin{array}{ll} \Delta \models x = p & \\ p \text{ is a PID} & \Gamma' = \Gamma[(p, q, t) \leftarrow n] \\ \Gamma(p, q, t) = m \cdot n & \Delta' = \Delta; [y \leftarrow m]_q \end{array}}{\Gamma, \Delta, [y \leftarrow \mathsf{recv}(x, t)]_q, \Psi \rightsquigarrow \Gamma', \Delta', \mathsf{skip}, \Psi}$$

Fig. 4.1. Proof Rules (Basic Statements)

Example Consider programs A and B, where s_1 to s_5 are statements.

$$A \triangleq [s_1]_p \;\|\; \prod q{:}Q.[s_2]_q \qquad\qquad B \triangleq [s_3]_p \;\|\; \prod q{:}Q.[s_4]_q \,\|\, [s_5]_m$$

The composition $A \circ B$ is given by the program $[s_1; s_3]_p \;\|\; \prod q{:}Q.[s_2; s_4]_q \;\|\; [s_5]_m$. While the above program is in normal form, A ; B and $A \;\|\; B$ are not.

Typed Channels and Message Orders Processes communicate by sending and receiving messages over *typed channels*. There is a separate channel for each (ordered) pair of processes. Furthermore, each channel is split into sub-channels for different types of message. Messages on the same sub-channel are delivered in order whereas there are no guarantees for messages sent on separate (sub-)channels. This semantics is standard and models languages like Erlang and Cloud Haskell.

Sends and Receives The statement $\mathsf{send}(t, p, e)$ asynchronously sends the value of expression e to p's sub-channel for type t. Dually, $x \leftarrow \mathsf{recv}(p, t)$ *blocks* until it receives a value of type t from process p, and then assigns the received value to the variable x. A receive from a set X denotes a receive from any $x \in X$; a receive from $*$ represents a receive from *any* process. In contexts where there is only a single message type, we omit types from the statements, *i.e.*, we use $\mathsf{send}(p, e)$ to denote a send of value of e to p and $x \leftarrow \mathsf{recv}(p)$ to denote a receive from p. Finally, we use $x \leftarrow \mathsf{recv}()$ as an abbreviation for $x \leftarrow \mathsf{recv}(*)$.

Example Let $s_0 \triangleq \mathsf{send}(t, r, v_0)$ and $s_1 \triangleq \mathsf{send}(t, r, v_1)$, which send a message of type t to r:

$$A \triangleq [s_0 \;;\; s_1]_p \;\|\; \begin{bmatrix} x_0 \leftarrow \mathsf{recv}(*, t); \\ x_1 \leftarrow \mathsf{recv}(*, t) \end{bmatrix}_r \qquad B \triangleq [s_0]_p \;\|\; [s_1]_q \;\|\; \begin{bmatrix} x_0 \leftarrow \mathsf{recv}(*, t); \\ x_1 \leftarrow \mathsf{recv}(*, t) \end{bmatrix}_r$$

In A, s_0 and s_1 are both executed by the process p which is composed in parallel with process r. Since the messages are sent on the same sub-channel, they are delivered in order, and on termination we have $(x_0, x_1) = (v_0, v_1)$. However, in B, where s_0 and s_1 are executed by different processes, either $(x_0, x_1) = (v_0, v_1)$ or $(x_0, x_1) = (v_1, v_0)$.

4 CANONICAL SEQUENTIALIZATIONS

In this section, we formalize canonical sequentializations through a set of rewriting rules.

Symbolic States Each rewriting rule defines a relation between a pair of *symbolic states* consisting of the following components: A *context* (Γ); a sequential *prefix* (Δ); the program (P) to be rewritten; and a *residual* process (Ψ) comprising statements that interact with processes outside of P. A context Γ consists of a symbolic message buffer Buff and a set of assertions Asrt. Buff maps each channel to the sequence of pending messages on that channel. More concretely, Buff(p, q, t) returns the sequence of pending messages of type t sent from p to q. Asrt contains assertions

$$\mathsf{skip}; P \equiv P \quad P; \mathsf{skip} \equiv P \quad \textstyle\prod p{:}P.\mathsf{skip} \equiv \mathsf{skip}$$
$$P \parallel \mathsf{skip} \equiv P \quad P \parallel Q \equiv Q \parallel P \quad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$

Fig. 4.3. Congruence relation.

about process identifiers. We summarize the syntax in Figure 4.2. We sometimes use Γ to refer to one of its components, for example, we write $\Gamma(p, q, t)$ to mean $\textsc{Buff}(p, q, t)$. Finally, for context $\Gamma \triangleq (\textsc{Buff}, \textsc{Asrt})$ and assertion a, we write $\Gamma \vdash a$ to mean $a \in \textsc{Asrt}$, and $\Gamma \nvdash a$ to mean $a \notin \textsc{Asrt}$.

Rewriting Rules Each rewriting rule defines a judgment of the form $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$. The goal of each step is to move parts of program P into the sequential prefix Δ such that eventually we can rewrite P to skip. We now describe the main rules of our method, starting with basic rules, followed by rules for loops and conditionals and finally residual processes.

$\textsc{Buff}$	$\in$	$(PID \times PID \times \mathsf{MsgType}) \rightharpoonup \mathsf{Exp}^*$	**Buffers**
$\textsc{Asrt}$	$::=$		**Assertions**
		$\emptyset$	*Empty*
	$\mid$	$\textsc{Asrt} \cup \{x \in X\}$	*Membership*
	$\mid$	$\textsc{Asrt} \cup \{\emptyset \subseteq X \subseteq X\}$	*Bounds*
	$\mid$	$\textsc{Asrt} \cup \{\mathsf{E}(x)\}$	*External*
Γ	$::=$	$(\textsc{Buff}, \textsc{Asrt})$	**Contexts**

Fig. 4.2. Syntax for context Γ.

4.1 Basic Rules

Figure 4.1 contains the basic rules. We assume for now that programs do not contain wild card receives and show how to eliminate wild cards from programs that only contain symmetric races in (§ 5.2).

Sends and receives Rule R-Send treats sends. The rule rewrites a send from process p to some x into skip, if x is either a PID q, or a variable that maps to some PID q, and additionally, q is not external (*i.e.*, belongs to a residual process). We enforce the check that x corresponds to a PID through the condition $\Delta \models x = q$, where we write $\Delta \models \varphi$ to mean that formula φ is valid after executing Δ (see § 5.2 for a discussion of $\Delta \models \varphi$). The rule updates context Γ by adding expression n to the end of the buffer for the respective channel, where we use $\Gamma[a \leftarrow b]$ to denote the function that returns the same values as Γ on all inputs except for a where it returns b. Rule R-Recv rewrites a receive from some x into skip, if x corresponds to a PID p. R-Recv takes the most recent message m from the respective channel and adds the corresponding assignment to the prefix.

Context and congruence Rule R-Context allows rewriting individual program parts independently of program parts that occur later or in parallel. Rule R-Congr allows rewriting into congruent programs, where $\equiv$ is shown in Figure 4.3.

R-Loop-Upd

$$\frac{\begin{array}{l} q^* \text{ and } Q^* \text{ fresh} \\ \Gamma_0 \triangleq \Gamma \cup \{\emptyset \subset Q^* \subseteq Q\} \cup \{q^* \in Q^*\} \qquad \Delta' \triangleq \Delta\, ; \begin{bmatrix} \text{for } q \text{ in } Q \text{ do} \\ \quad \Delta^u[q/u] \\ \text{end} \end{bmatrix} \qquad \Psi' \triangleq \Psi \parallel \prod q{:}Q.[\Psi^u]_q \\ \Delta_0 \triangleq \text{havoc}(\Delta, A, B) \\ \Gamma_0, \Delta_0, [A[q^*/q]]_p \parallel \prod q{:}Q^*.[B]_q, \text{skip} \rightsquigarrow \Gamma_0, (\Delta_0; \Delta^u)\,, \left(\text{skip} \parallel \prod q{:}Q^*\backslash\{u\}.[B]_q\right), [\Psi^u]_u \end{array}}{\Gamma, \Delta, \begin{bmatrix} \text{for } q \text{ in } Q \text{ do} \\ \quad A \\ \text{end} \end{bmatrix}_p \parallel \prod q{:}Q.\left[B; C\right]_q, \Psi \rightsquigarrow \Gamma, \Delta', \prod q{:}Q.[C]_q, \Psi'}$$

Fig. 4.5. Rewrite Rules (Iteration over sets of processes).

$$\begin{bmatrix} \text{send}(q, \mathit{ping}); \\ w \ \leftarrow \text{recv}(q) \end{bmatrix}_p \parallel \begin{bmatrix} v \ \leftarrow \text{recv}(p); \\ \text{send}(p, \mathit{pong}) \end{bmatrix}_q$$

(a) ex1

$$\begin{array}{l} \text{skip}; \\ [w \ \leftarrow \text{recv}(q)]_p \end{array} \parallel \begin{bmatrix} v \ \leftarrow \text{recv}(p); \\ \text{send}(p, \mathit{pong}) \end{bmatrix}_q$$

(b) after send

$$\begin{array}{l} \text{skip}; \\ [w \ \leftarrow \text{recv}(q)]_p \end{array} \parallel \begin{array}{l} \text{skip}; \\ [\text{send}(p, \mathit{pong})]_p \end{array}$$

(c) and after receive.

Fig. 4.4. Example ex1: rewriting send and receive.

Example Consider again program ex1 from § 2 (shown in fig. 4.4a) where we replaced wild card receives with receives from the respective processes. We start the rewrite with the symbolic state given by $\Delta \triangleq \text{skip}$, $\Psi \triangleq \text{skip}$ and $\Gamma \triangleq (\text{Buff}_\emptyset, \emptyset)$, where $\text{Buff}_\emptyset$ maps every channel to the empty sequence ϵ. We also assume that there is only a single message type $\top$. In a first step, we apply the rules R-Context and R-Send to rewrite ex1 into the program shown in fig. 4.4b, where we update the buffer to $\text{Buff}_\emptyset[(p, q, \top) \leftarrow \mathit{ping}]$. Applying R-Context and R-Recv yields the program shown in fig. 4.4c with buffer $\text{Buff}_\emptyset$ and prefix $\Delta = [v \leftarrow \mathit{ping}]_q$. Applying R-Context and R-Congr twice yields $[w \ \leftarrow \text{recv}(q)]_p \parallel [\text{send}(p, \mathit{pong})]_p$. Finally, applying the same rules again yields skip with prefix $\Delta = [v \leftarrow \mathit{ping}]_q;\ [w \leftarrow \mathit{pong}]_p$.

4.2 Loops, Unfolding and Conditionals

Next, we present our rules for loops. We first present our rules for iterating over sets of *processes*, then our rules for iterating over sets of *indices*, and finally our rules for while-loops and if-statements.

Iterating over identical processes Figure 4.5 contains rule R-Loop-Upd for rewriting the interaction between a set of identical processes Q and a process p which iterates over Q. The rewrite succeeds if we can rewrite the interaction between an *arbitrary* iteration of p and a *single process* in Q, *independently* of previous iterations and other processes. This condition is enforced through the rewrite-step that appears in the pre-condition of the rule. The rule picks a fresh PID $q^* \in Q$ (corresponding to the value of q in the chosen iteration) and a subset $\emptyset \subset Q^* \subseteq Q$ (corresponding to the set of remaining processes, in that iteration) and shows that it is possible to *unfold* a process u from Q^* (u may or may not be equal to q^*) such that u and p can be rewritten to skip. Explicitly unfolding process u from Q^* ensures that the iteration talks to process u, only. To ensure iterations are independent, the rule modifies the prefix by havocing all variables that may be assigned in the loop by assigning a non-deterministic value. Likewise, the rule requires the context after the rewrite to be the same as before in order to rule out superfluous sends.

R-Send-Unfold

$$\frac{\Gamma \vdash q^* \in Q}{\begin{array}{l}\Gamma,\Delta,\left([\mathsf{send}(t, q^*, n)]_p \;\|\; \prod q{:}Q.A\right), \qquad \Psi \rightsquigarrow \\ \Gamma,\Delta,\left([\mathsf{send}(t, q^*, n)]_p \;\|\; \prod q{:}Q\setminus\{q^*\}.A \;\|\; [A]_{q^*}\right), \Psi\end{array}}$$

R-Recv-Unfold

$$\frac{\begin{array}{l} q^* \text{ fresh} \\ \Gamma \vdash \emptyset \subset Q' \subseteq Q \\ \Gamma, \Delta, \left([x \leftarrow \mathsf{recv}(q^*, t)]_p \,\|\, [A]_{q^*}\right), \Psi \rightsquigarrow \Gamma, \Delta', [A']_{q^*}, \Psi' \end{array}}{\begin{array}{l}\Gamma,\Delta,\left([x \leftarrow \mathsf{recv}(Q, t)]_p \;\|\; \prod q{:}Q'.[A]_q\right), \qquad \Psi \rightsquigarrow \\ \Gamma,\Delta,\left([x \leftarrow \mathsf{recv}(q^*, t)]_p \,\|\, \prod q{:}Q'\setminus\{q\}.A \;\|\; [A]_{q^*}\right), \Psi\end{array}}$$

R-Send-Resid

$$\frac{\begin{array}{ll} V \text{ fresh} & q \text{ is a PID} \\ \Delta \models x = q & \Delta' = \left(\Delta; [V \leftarrow V \cup \{n\}]_p\right) \\ \Gamma \vdash \mathsf{E}(q) & \Psi' = \left(\Psi \circ \sum v{:}V.[\mathsf{send}(t, q, v)]_p\right) \end{array}}{\Gamma, \Delta, [\mathsf{send}(t, x, n)]_p, \Psi \rightsquigarrow \Gamma, \Delta', \mathsf{skip}, \Psi'}$$

R-Compose-Resid

$$\frac{\begin{array}{ll} \Gamma_0 \triangleq \Gamma \cup \mathsf{E}(p) \text{ for } p \in \mathsf{Procs}(B) & \mathsf{rf}(B \,\|\, \Psi) \\ \Gamma_0, \Delta, A, \mathsf{skip} \rightsquigarrow \Gamma_0, \Delta', \mathsf{skip}, \Psi & \\ \Gamma, \Delta', B \,\|\, \Psi, \mathsf{skip} \rightsquigarrow \Gamma, \Delta'', \mathsf{skip}, \mathsf{skip} & \end{array}}{\Gamma, \Delta, A \,\|\, B, \mathsf{skip} \rightsquigarrow \Gamma, \Delta'', \mathsf{skip}, \mathsf{skip}}$$

Fig. 4.6. Rewrite Rules (Unfold, Residue)

Unfolding Figure 4.6 shows rules R-Send-Unfold and R-Recv-Unfold which unfold a single process from a set of identical processes. Rule R-Send-Unfold allows unfolding a process q^* from a set Q, if there is a send to q^*, and it follows from the context that $q^* \in Q$. Rule R-Recv-Unfold treats a situation in which a process p can receive from *any* of the processes in a set of identical processes Q', *i.e.*, there is a race between these processes. The rule picks a fresh $q^* \in Q'$ and unfolds it from the set. It then modifies the receive such that it can only receive from the freshly chosen PID. The rule has an additional precondition requiring that the receive can be rewritten to skip, *i.e.*, there is in fact a matching send in A. This precondition is required to ensure that the rewrite-step does not introduce any deadlocks (by over-specializing the receive from any process in Q' to just q^*).

Example Consider ex5 shown left below (this example is based on ex4 from § 2). As before, we eliminated wild card receives and assume that there is only a single message type. Our goal is to apply R-Loop-Upd to produce the sequentialization ex5$_{\text{Goal}}$ shown on the right.

$$\text{ex5} \triangleq \begin{bmatrix} \mathsf{for}\ q\ \mathsf{in}\ Q\ \mathsf{do} \\ \quad id \leftarrow \mathsf{recv}(Q); \\ \quad \mathsf{send}(id, ping) \\ \mathsf{end} \end{bmatrix}_p \;\|\; \prod q{:}Q. \begin{bmatrix} \mathsf{send}(p, q); \\ v \leftarrow \mathsf{recv}(p) \end{bmatrix}_q$$

$$\text{ex5}_{\text{Goal}} \triangleq \mathsf{for}\ q\ \mathsf{in}\ Q\ \mathsf{do} \quad \begin{array}{l} [id \leftarrow q]_p\ ; \\ [v \leftarrow ping]_q \end{array} \quad \mathsf{end}$$

In order to satisfy the precondition of rule R-Loop-Upd, we need to rewrite the program I shown below, which corresponds to an arbitrary iteration of p's loop (Q^* is a fresh set with $\Gamma \vdash \emptyset \subset Q^* \subseteq Q$), by unfolding a process from Q^* and rewriting p and the unfolded process to skip. Applying R-Recv-Unfold yields the program I' shown on the right, where q^* is a fresh PID. We can rewrite I' into skip $\| \prod q{:}Q^*\setminus\{q^*\}.[\cdots]$ with sequential prefix $\Delta \triangleq \left([id \leftarrow q^*]_p\ ;\ [v \leftarrow ping]_{q^*}\right)$ thereby satisfying the goal in the precondition of R-Loop-Upd. This allows us to rewrite the entire program to skip (using an additional application of Rule R-Congr) which produces the sequential prefix

$$\textsc{R-While-Repeat}\quad \frac{\Gamma, \Delta, [A]_p \;\|\; [B]_q, \Psi \rightsquigarrow \Gamma', \Delta', \mathsf{skip}, \Psi'}{\begin{array}{l}\Gamma,\ \Delta,\ [\mathsf{while\ true\ do}\ A\ \mathsf{end}]_p \;\|\; [B\,;C]_q, \Psi \rightsquigarrow \\ \Gamma', \Delta', [\mathsf{while\ true\ do}\ A\ \mathsf{end}]_p \;\|\; [C]_q, \quad \Psi'\end{array}}$$

$$\textsc{R-While-Remove}\quad \frac{\Gamma, \Delta, [A]_p \,\|\, [B]_q, \Psi \rightsquigarrow \Gamma', \Delta', \mathsf{break}, \Psi'}{\begin{array}{l}\Gamma,\ \Delta,\ \left[\mathsf{while\ true\ do}\ A\ \mathsf{end}\right]_p \;\|\; [B;C]_q, \Psi \rightsquigarrow \\ \Gamma', \Delta', [C]_q, \qquad\qquad \Psi'\end{array}}$$

$$\textsc{R-if-then}\quad \frac{\Delta \models e}{\Gamma, \Delta, \mathsf{if}\ e\ \mathsf{then}\ A\ \mathsf{else}\ B, \Psi \rightsquigarrow \Gamma, \Delta, A, \Psi} \qquad \textsc{R-if-else}\quad \frac{\Delta \models \neg e}{\Gamma, \Delta, \mathsf{if}\ e\ \mathsf{then}\ A\ \mathsf{else}\ B, \Psi \rightsquigarrow \Gamma, \Delta, B, \Psi}$$

Fig. 4.7. Rewrite Rules (Branch)

$\textsc{ex5}_{\textsc{Goal}}$.

$$I \triangleq \begin{bmatrix} id \leftarrow \mathsf{recv}(Q); \\ \mathsf{send}(id, ping) \end{bmatrix}_p \;\|\; \prod q{:}Q^*. \begin{bmatrix} \mathsf{send}(p, q); \\ v \leftarrow \mathsf{recv}(p) \end{bmatrix}_q$$

$$I' \triangleq \begin{bmatrix} id \leftarrow \mathsf{recv}(q^*); \\ \mathsf{send}(id, ping) \end{bmatrix}_p \;\|\; \begin{bmatrix} \mathsf{send}(p, q^*); \\ v \leftarrow \mathsf{recv}(p) \end{bmatrix}_{q^*} \;\|\, \prod q{:}Q^*\backslash\{q^*\}.[\cdots]$$

Example Consider example ex6, shown below. ex6 is a variant of the previous example that is *rejected* by our method. As before, in order to satisfy the precondition of rule R-Loop-Upd, we need to rewrite the program I corresponding to an iteration of p's loop.

$$\textsc{ex6} \triangleq \begin{bmatrix} \mathsf{for}\ q\ \mathsf{in}\ Q\ \mathsf{do} \\ \quad \mathsf{send}(q, ping)\ ; \\ \quad v \leftarrow \mathsf{recv}(Q) \\ \mathsf{end} \end{bmatrix}_p \;\|\; \prod q{:}Q. \begin{bmatrix} \mathsf{send}(p, pong); \\ w \leftarrow \mathsf{recv}(p) \end{bmatrix}_q$$

$$I \triangleq \begin{bmatrix} \mathsf{send}(q^*, ping)\ ; \\ v \leftarrow \mathsf{recv}(Q) \end{bmatrix}_p \;\|\; \prod q{:}Q^*. \begin{bmatrix} \mathsf{send}(p, pong); \\ w \leftarrow \mathsf{recv}(p) \end{bmatrix}_q$$

Applying rule R-Send-Unfold yields the program I'. However, our method fails to rewrite this program as this would require unfolding a *second* process to handle the receive from Q.

$$I' \triangleq \begin{bmatrix} \mathsf{send}(q^*, ping)\ ; \\ id \leftarrow \mathsf{recv}(Q) \end{bmatrix}_p \;\|\; \begin{bmatrix} \mathsf{send}(p, pong)\ ; \\ w \leftarrow \mathsf{recv}(p) \end{bmatrix}_{q^*} \;\|\cdots$$

Intuitively, this is because the receive in p can receive a *pong* message from *any* process (not just the one it just sent to), which violates the requirement that each loop iteration should only talk to a *single* process.

Iterating over sets of Indices Figure 4.8 contains rule R-Loop-Repeat. The rule rewrites the interaction between a set of processes Q and a process p which iterates over a set of indices I. Again, the rule contains a precondition that requires rewriting the interaction between an arbitrary iteration of p and a single process from Q. For this, the rule picks a fresh i^* and requires showing that we can unfold a process u from Q such that p's iteration can be rewritten to skip while process u remains *unchanged*, *i.e.*, executable after the interaction. The rule produces a sequential prefix by repeating the prefix produced in the iteration where the unfolded process u is substituted for an arbitrary process from Q.

While Loops and Conditionals

$$\textsc{R-Loop-Repeat}$$

$$\frac{\begin{array}{l} i^* \text{ fresh} \\ \Delta_0 \triangleq \text{havoc}(\Delta, A, B) \end{array} \qquad \Delta' \triangleq \Delta \,;\begin{bmatrix}\text{for } i \text{ in } I \text{ do} \\ \quad \sum q{:}Q.\,(\Delta^u[i/i^*][q/u]) \\ \text{end}\end{bmatrix} \qquad \Psi' \triangleq \Psi \parallel \begin{pmatrix}\text{for } i \text{ in } I \text{ do} \\ \quad \sum q{:}Q.[\Psi^u]_q \\ \text{end}\end{pmatrix} \qquad \Gamma, \Delta_0, \left([A[i^*/i]]_p \parallel \prod q{:}Q.[B]_q\right), \text{skip} \rightsquigarrow \Gamma, (\Delta_0;\Delta^u), (\text{skip} \parallel [B]_u \parallel \prod q{:}Q\backslash\{u\}.B), [\Psi^u]_u}{\Gamma, \Delta, \begin{bmatrix}\text{for } i \text{ in } I \text{ do} \\ \quad A \\ \text{end}\end{bmatrix}_p \parallel \prod q{:}Q.[B]_q, \Psi \rightsquigarrow \Gamma, \Delta', \prod q{:}Q.[B]_q, \Psi'}$$

Fig. 4.8. Rewrite Rules (Iteration over sets of indices).

Consider again Figure 4.7. Rule R-WHILE-REPEAT unrolls an iteration of a while loop, if the iteration, together with some prefix B of another process, can be rewritten to skip. Rule R-WHILE-REMOVE unrolls an iteration of a while loop, if the iteration together with some prefix B of another process can be rewritten to break. It then removes the while loop from the program. Rule R-IF-THEN allows to rewrite the then-branch, if the condition holds; R-IF-ELSE allows rewriting to the else-branch, if the condition does not hold. Our system additionally contains rules R-BRANCH that allows to rewrite an if-statement, if both branches, together with an additional context, can be rewritten to skip, R-NONDET-RECV which allows receiving from a non-deterministically chosen process in an identical set, and a rule for rewriting pairs of for-loops.

Example Consider example EX7 shown below, in which process p interacts with a set of processes Q. p executes a loop in which it receives a PID and then sends back the value 0. Each process in Q sends its PID to p and waits for a reply. Upon receipt, it assigns the received value to *stop* and breaks from the loop, if *stop* is one.

$$\begin{bmatrix}\text{for } i \text{ in } I \text{ do} \\ \quad id \ \leftarrow \text{recv}(Q) \ ; \\ \quad \text{send}(id, 0) \\ \text{end}\end{bmatrix}_p \parallel \prod q{:}Q. \begin{bmatrix}\text{while true do} \\ \quad \text{send}(p, q) \ ; \\ \quad stop \ \leftarrow \text{recv}(p) \ ; \\ \quad \text{if } stop = 1 \text{ then} \\ \quad\quad \text{break} \\ \quad \text{else skip} \\ \text{end}\end{bmatrix}_q$$

Our goal is to use rule R-LOOP-REPEAT to rewrite p to skip so that the remaining program consists only of the parallel composition over Q. Applying R-LOOP-REPEAT and R-RECV-UNFOLD yields the program shown below, where q^* is fresh:

$$\begin{bmatrix} id \ \leftarrow \text{recv}(q^*) \ ; \\ \text{send}(id, 0)\end{bmatrix}_p \parallel \begin{bmatrix}\text{while true do} \\ \quad \text{send}(p, q^*) \ ; \\ \quad stop \ \leftarrow \text{recv}(p) \ ; \\ \quad \text{if } stop = 1 \text{ then} \\ \quad\quad \text{break} \\ \quad \text{else skip} \\ \text{end}\end{bmatrix}_{q^*} \parallel \ldots$$

Using an application of R-WHILE-REPEAT and R-IF-ELSE, we can rewrite process p to skip, producing sequential prefix $\Delta \triangleq [id \leftarrow q^*]_p \ ; \ [stop \leftarrow 0]_{q^*}$, yielding the final sequential prefix:

$$\text{for } i \text{ in } I \text{ do} \quad \sum q{:}Q. \begin{pmatrix} [id \leftarrow q]_p \ ; \\ [stop \leftarrow 0]_q \end{pmatrix} \quad \text{end}$$

4.3 Residual Processes

Finally, Figure 4.6 shows our rules for residual processes. Rule R-Send-Resid moves a send of value n to process x into the residual process Ψ, if x is *external.* Since we are postponing the execution of the send, the rule takes a “snapshot” of n by inserting it into a fresh set V and adding the assignment to the sequential prefix (we assume that V is initialized to $\emptyset$). The residual process then non-deterministically sends a value from V (note that this step introduces an over-approximation). Our system contains an additional rule R-Recv-Resid for receives. Rule R-Compose-Resid allows a modular rewriting of programs. It takes two parallel programs A and B and first rewrites A while treating all processes in B as *external*, meaning that all sends to processes in B ($\mathsf{Procs}(B)$) are added to the residual process Ψ. The rule then composes the residual process with B and rewrites the composition. The rule requires the composition of Ψ and B be symmetrically non-deterministic, indicated by $\mathsf{rf}(B \,||\, \Psi)$.

Example Consider again example ex4 from Figure 2.7. We apply rule R-Compose-Resid, rewriting process p and the processes in Q, treating process m as external. This produces the residual term shown in Figure 2.9, where we remove the non-deterministic choice as the set only contains one element. Next, we compose the residual process with process m as shown left in 2.8. The program is race-free up to symmetry as each receive in m can only receive messages from processes in Q that are at the same program location. Rewriting the resulting program yields sequentialization show right in fig. 2.8.

4.4 Correctness

Termination Each rewrite rule, with the exception of R-Congruence, decreases the size of the input program. Moreover, there are finitely many ways to instantiate each rule. Therefore, we guarantee termination of our rewriting by restricting the use of R-Congruence to situations where it *decreases* the size of the input program.

Rewrite Soundness We now present our main correctness theorem. For this, we need the following additional definitions. We define a *program state* as a triple (σ, μ, P), where $\sigma \in (PID \times \mathsf{Var} \rightharpoonup \mathsf{Val})$ is a (partial) map such that $\sigma(p, x)$ is the value of the variable x in process p, $\mu \in (PID \times PID \times \mathsf{MsgType} \rightharpoonup \mathsf{Val}^*)$ is a map from channels to sequences of values, and P is a program. We define an interpretation on prefixes and contexts such that $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ when σ and μ are a store and message buffer consistent with the states reachable by executing Δ and the assumptions in Γ. Let $\sigma|_P$ denote the store whose domain is restricted to the variables of the processes in P. We denote the set of processes that are halted in a state (and will never become enabled) as $\mathsf{halted}(\sigma, \mu, P)$.

Theorem 4.1. *Let P be a program in normal form. If*

(1) $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$

(2) $\mathsf{rf}(P \circ E)$ *for some* extension E *and* $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ *such that* $(\sigma, \mu, \Psi; P \circ E) \rightarrow (\sigma_Q, \mu_Q, Q)$

Then there exists $(\sigma', \mu') \in [\![\Delta', \Gamma']\!]$ *such that* $(\sigma', \mu', \Psi'; P' \circ E) \rightarrow (\sigma_{Q'}, \mu_{Q'}, Q')$ *and* $\sigma_Q|_H = \sigma_{Q'}|_H$ *where* $H = \mathsf{halted}(\sigma_Q, \mu_Q, Q)$.

The inverse direction does not hold since our method over-approximates values sent by residual processes.

Proof (Sketch). The proof is by induction on the derivation of $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$, splitting cases on the final step. Left-movers such as sends may be sequentialized while preserving the states of halted processes, since they commute with other actions by definition [Lipton 1975]. Importantly, the case for R-Recv uses the fact that receives are left movers, up to their matching

$$\begin{array}{rcll} e & ::= & x \mid \lambda x : t . e \mid e \;\; e \mid C \;\; \overline{e} \mid \textbf{case} \;\; e \;\; \textbf{of} \;\; \overline{\mid C \; \bar{x} \;\; \rightarrow e} \mid \textbf{let} \;\; x \;=\; e \;\; \textbf{in} \;\; e \mid \textbf{fix} \;\; f . e & \textit{Pure Terms} \\ & \mid & \texttt{send} \;\; e \;\; e \mid \texttt{receive} \mid \texttt{spawn} \;\; e \mid \texttt{getSelfPid} \mid \texttt{return} \;\; e \mid e \;\; \texttt{>>=} \;\; e \mid \texttt{foldM} \;\; e \;\; e \;\; e & \textit{Effectful Terms} \\ t & ::= & \texttt{()} \mid \texttt{Int} \mid \cdots \mid \texttt{ProcessId} \mid \texttt{Process} \;\; t \mid t \rightarrow t & \textit{Types} \end{array}$$

Fig. 5.1. Syntax of λ_P terms

```
data Message = Work Int |
    Term

mapper :: ProcessId
       → ProcessId
       → Process ()
mapper q r = mapperLoop
  where
    mapperLoop = do
      me ← getSelfPid
      send q me
      w  ← receive
      case w of
        Work i → do
          send r i
          mapperLoop
        Term   → return ()
```

```
queue :: Int
      → [Int]
      → ProcessId
      → Process ()
queue n work r = do
  me      ← getSelfPid
  workers ← spawn n (mapper
       me r)
  foldM distribute () work
  foldM terminate () workers
 where
  distribute _ i = do
    x ← receive
    send x (Work i)
  terminate  _ _ = do
    x ← receive
    send x Term
```

```
reducer :: Int
        → Int
        → Process ()
reducer k n = do
  me ← getSelfPid
  spawn 1 (queue n work me)
  foldM waitForUnit () work
  return ()
 where
  work = [1..k]
  waitForUnit _ _ = receive

main :: Process ()
main = do n ← readNumWorkers
          k ← readWorkUnits
          reducer n k
```

Fig. 5.2. MapReduce implemented in Haskell

send. The case for R-Recv-Unfold relies on introducing a prophecy variable [Abadi and Lamport 1991] that guesses the sender of the eventually received message. □

Theorem 4.1 implies that we can check reachability for any halted subset of processes, consistent with results from finite-state partial order reduction techniques (*e.g.*, Siegel and Avrunin [2005]). Thus, if we can rewrite a program P into prefix Δ, then (1) P is deadlock free and (2) process-local safety properties of Δ are enjoyed by P.

5 IMPLEMENTATION

Next, we describe the implementation of our approach in a tool called Brisk, that takes as input a program written in Haskell using the Cloud Haskell libraries [Epstein et al. 2011]. Brisk first compiles it to an IceT term (§ 5.1) and then rewrites the IceT term to its canonical sequentialization (§ 5.2) in order to verify concurrency properties of the input Haskell source.

5.1 From Haskell to IceT

First, we describe how Haskell terms are compiled to IceT programs.

MapReduce in Brisk Figure 5.2 contains an implementation of MapReduce in Haskell which we use as running example. MapReduce comprises a `reducer`, a `queue`, and a set of `mapper` processes. The program is parameterized by `k`, the amount of work, and `n`, the number of `mapper` processes. The `mapper` process (implemented in `mapperLoop`) queries the queue for work. If the response is a unit of work, it sends a result (in this case, just the work unit `i` itself) to the `reducer` process and calls `mapperLoop` to begin the next iteration; if the response is `Term`, it terminates. The work queue first spawns `n` `mapper` processes, and then uses `foldM`[2] to wait for `k` requests which it answers with a unit of work `Work i`. Having distributed `k` units of work, the queue waits for each `mapper` to send a request, and responds with a `Term` message, causing the `mapper` to exit. The `reducer`

[2] The `foldM` combinator is a standard left-fold where the "fold function" can be effectful.

```
λq.λr.fix loop.
  getSelfPid >>=
  λme. send q me >>=
  λ_.  receive >>=
  λw.  case w of
         Work i →  send r i >>=
                λ_. loop
         Term   →  return ()
```

Fig. 5.3. λ_{P} term corresponding to the body of mapper from fig. 5.2. (λ binds tighter than >>=.)

$$
\begin{aligned}
\mathsf{wf}_f(x) &= \mathsf{true}\\
\mathsf{wf}_f(\lambda x.\ e) &= \mathsf{wf}_f(e)\\
\mathsf{wf}_f(e_1\ e_2) &= f \notin \mathsf{fv}(e_2) \land \mathsf{wf}_f(e_1)\\
\mathsf{wf}_f(\mathsf{case}\ e\ \mathsf{of}\ \overline{p_i \to e_i}) &= f \notin \mathsf{fv}(e) \land \forall\, i.\ \mathsf{wf}_f(e_i)\\
\mathsf{wf}_f(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) &= f \notin \mathsf{fv}(e_1) \land \mathsf{wf}_f(e_2)\\
\mathsf{wf}_f(e_1 \mathbin{>\!\!>\!=} e_2) &= f \notin \mathsf{fv}(e_1) \land \mathsf{wf}_f(e_2)
\end{aligned}
$$

Fig. 5.4. Well-formedness of bodies of λ_{P} fix expressions. We denote the set of free variables of the expression e by $\mathsf{fv}(e)$.

process first spawns the work queue and then waits to receive k messages from the mappers. The main function first reads in the number of workers and work units and then executes the program.

Abstract Syntax We now describe how to extract a IceT program from a Haskell program. We formalize our translation in terms of a language λ_{P}, which models Cloud Haskell programs. The language consists of a lambda calculus extended with algebraic data types and let bindings (fig. 5.1). General recursive functions are implemented using the fixed-point combinator fix. The language is extended with the effectful primitives send and receive for sending and receiving messages. Processes are spawned by calling spawn e p where e is an expression evaluating to the *number* of processes to spawn and p is an expression[3] evaluating to the program of the new process. The primitive getSelfPid returns the identifier of the currently running process. Effectful programs are built using these primitives and the monadic combinators return, >>=, and foldM. Figure 5.3 shows the desugared λ_{P} term corresponding to mapper from fig. 5.2.

Pure and Impure expressions. λ_{P} expressions are typed. For ease of presentation, we consider a type system that is much simpler than that of Haskell, as our translation is oblivious to most of its features. If the expression e has type t, then we write $e :: t$. Effectful expressions m have types of the form Process t, where t is the type of the value returned by executing m. We say that Process t is *impure*, and call $t' \to t$ impure, if t is.

Translation to IceT In this section, we consider first-order terms m, where functions appear as arguments only to primitives. Higher-order programs can be handled by *defunctionalization* (*e.g.*, Mitchell and Runciman [2009]). Next, we traverse the abstract syntax tree (AST) of m to compute the corresponding IceT program, using the types of the child expressions to guide the translation. If m :: Process t, then the term corresponds to a tree of expressions where each node is an application of >>=.

To translate an impure λ_{P} term m, we define the mapping $\mathcal{T}\{m\}$. The function $\mathcal{T}\{m\}$ traverses the tree structure of m. Sub-expressions of the form m >>= $(\lambda x.n)$ correspond to IceT programs of the form $x \leftarrow s_m; s_n$, where s_m and s_n are IceT terms corresponding to m and n. Let-bound functions are inlined, and their subsequent applications are β-reduced before translating to IceT. We translate algebraic data types to tuples, and hence translate **case** expressions to a sequence of **if** statements [Cardelli 1984].

Since our goal is to produce IceT programs in normal form, (§ 3), we limit the occurrences of spawn. Our rewrite rules consider a flat, parallel composition of processes, so we require that uses of spawn occur (1) *before* any message sends or receives and (2) *outside* of any loops (*i.e.*, **fix** and foldM) or conditionals.

[3] We refer the interested reader to Epstein et al. [2011] for a discussion of how to handle closures.

```
R ≜ [ for $p$ in K do
        $x \leftarrow$ recv($*$, Int) $\{b\}$ (1)
      end ]$_{\mathrm{m}}$

W($p$) ≜ [ while true do
             send(PID, q, $p$);               $a$
             $w \leftarrow$ recv($*$, Message) $\{c, d\}$ (2)
             if $w$ = Work $i$ then
               send(Int, m, $i$)              $b$
             else
               break
           end ]$_p$

Q ≜ [ for $i$ in K do
        $x \leftarrow$ recv($*$, PID);             $\{a\}$ (3)
        send(Message, $x$, Work $i$)         $c$
      end;
      for $p$ in P do
        $x \leftarrow$ recv($*$, PID);             $\{a\}$ (4)
        send(Message, $x$, Term)           $d$
      end ]$_{\mathrm{q}}$
```

$$\textsc{MapReduce} \triangleq Q \parallel \textstyle\prod p{:}P.W(p) \parallel R$$

Fig. 5.5. Tagged IceT model of MapReduce program from fig. 5.2

We restrict the translation of recursive, impure computations by defining a notion of well-formedness that corresponds to tail-recursion. The intuition is that any recursive computation, *i.e.*, an expression of the form `fix` $f.e$, the recursive call f should be the *last action*. The definition of $\mathrm{wf}_f(e)$ in fig. 5.4 closely follows the definition of a tail recursion for a function f. We thus require that $\mathrm{wf}_f(e')$ for all impure sub-expressions `fix` $f.e'$ of e. With this restriction, it is straightforward to translate recursive, effectful functions into while loops. The body of the `fix` term in fig. 5.3 is well-formed: the recursive call to `loop` is the last action.

MapReduce in IceT The Haskell code for `mapper` from fig. 5.2 is thus translated to a while loop in the extracted IceT program shown in fig. 5.5, which also shows the send and receive tags, derived from the types as described in § 5.2. Both m and q contain receives that are tagged with a single location in the set of processes P, while the receive in the P process is tagged only with sends from the q process. Since the receive at (1) is tagged with b, which appears in the worker process, the statement is equivalent to $x \leftarrow \mathrm{recv}(\mathrm{P}, \texttt{Int})$. Likewise, the receive at (2) is equivalent to $w_p \leftarrow \mathrm{recv}(\mathrm{q}, \texttt{Message})$ and wildcards in (3) and (4) may be instantiated to P.

5.2 Canonical Sequentialization of IceT Programs

We synthesize canonical sequentializations by implementing our rewrite rules § 4 as a Prolog predicate `rewrite`, shown below. Intuitively, predicate `rewrite`$(P, \Gamma, \Delta, \Psi, P', \Gamma', \Delta', \Psi')$ holds if $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$. In order to rewrite a program P we can use the query

$$\texttt{rewrite}(P, (\textsc{Buff}_\emptyset, \emptyset), \mathrm{skip}, \mathrm{skip}, \mathrm{skip}, (\textsc{Buff}_\emptyset, \_), \Delta, \mathrm{skip})$$

which rewrites P to skip producing its canonical sequentialization Δ.

```
rewrite(P,Γ,Δ,Ψ,P″,Γ″,Δ″,Ψ″) :-  (   P = P″,Γ = Γ″,Δ = Δ″,Ψ = Ψ″
                                 ;   rewrite_step(P,Γ,Δ,Ψ,P′,Γ′,Δ′,Ψ′),!,
                                     rewrite(P′,Γ′,Δ′,Ψ′,P″,Γ″,Δ″,Ψ″)).
```

Predicate `rewrite` is defined recursively: either the rewrite is complete (*i.e.*, $P = P'$), or the predicate applies a predicate `rewrite_step` which implements a *disjunction* over all rewrite rules (*i.e.*, R-Send, R-Recv, and so on) and recursively rewrites the results. Crucially, `rewrite` performs a *cut* (using the notation !) after each successful rewrite step: once a rewrite step succeeds, the program cannot backtrack to try alternative steps. In general, this may eliminate valid rewrite

sequences unless the rules are confluent (*i.e.*, the order in which rules are applied does not matter). Unfortunately, our rules are not confluent in a limited number of cases. For example, in some situations both R-Send and R-Send-Unfold are applicable. In this situation, applying R-Send will cause the rewrite to get stuck: R-Send consumes the send to a member of some set, and hence it is impossible to unfold the corresponding set. We address this difficulty by fixing an ordering on these rules. For example, our implementation will always try to apply R-Send-Unfold before R-Send. The resulting system *is* confluent: the cuts do not eliminate valid rewrites. This step is crucial for ensuring that Brisk is fast enough for interactive use.
Predicate `rewrite` can also be used to rewrite a program to something other than skip, *e.g.*, for programs where some reactive components (*e.g.*, servers) may not terminate and we only want to rewrite the finite interaction between clients and server. Here, we rewrite the client to skip and keep the server as a remainder term.

Semantic Entailment Our proof system makes use of the entailment relation, $\Delta \models \varphi$, *e.g.*, in R-Send and R-Recv. As Δ may contain loops, entailment is undecidable. Computing approximations of $\Delta \models \varphi$ is orthogonal to this paper, however, we found that for our benchmarks (§ 6), loop invariants that track constants were sufficient.

Checking Symmetric Non-Determinism We check symmetric non-determinism through the following effective over-approximation: for each receive, we add tags of all sends of the matching type, where we treat sends to variables as sends to *any* PID and assume processes do not send messages to themselves. While this approximation sufficed for all our examples, our method is orthogonal to the method employed for race detection and can easily be extended to a more precise analysis.

Wildcard Instantiation We replace each wildcard with the expression implied by its tags. This is possible since symmetric non-determinism ensures that each receive is matched with a single process or a symmetric set.

6 EVALUATION

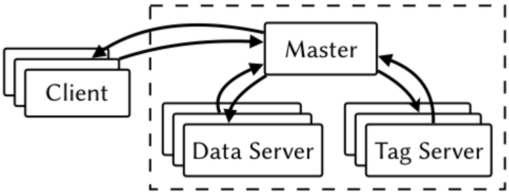
To test the effectiveness of our approach, we ported programs from related work to Haskell. Our evaluation demonstrates that a large variety of idiomatic, asynchronous message passing programs satisfy symmetric non-determinism. Even though programs satisfying symmetric nondeterminism are nontrivial to verify, Brisk can rapidly compute their canonical sequentializations, implying that our method can be easily integrated in an iterative design-implement-check loop.

Methodology To evaluate Brisk, we recorded the time it takes to rewrite each benchmark into its canonical sequentialization. For programs that should terminate, we compute the canonical sequentialization by rewriting the entire program to skip. For systems with reactive components, we closed the system by adding clients that execute a finite program (*e.g.*, non-deterministically call RPCs). We then ran Brisk to compute the canonical sequentialization of the interaction of clients and the reactive components. For both classes of programs, beyond summarizing the input program's behavior, computing a canonical sequentialization serves as proof that (1) the program does not deadlock (modulo reactive components) and (2) processes do not call `fail`. We only check for simple assertion failures such as the one in fig. 2.1 by treating fail as a special primitive that cannot be rewritten. Checking arbitrary safety properties is an undecidable problem in general, however, one can use the computed sequentialization to check more expressive properties using off-the-shelf techniques for verifying sequential programs.

To gauge the complexity of verifying programs that satisfy symmetric non-determinism, we have manually written a Promela model for each benchmark by instantiating its parameters and suitably abstracting its data values. We found the smallest instantiation such that Spin was unable

Name	#Param	#LOC	Spin N	IceT #Term	Brisk time (ms)
ex2	1	14	-	69	20
ex3	1	13	11	57	20
PingDet	1	17	13	83	20
PingIter	1	19	11	63	20
PingSym	1	13	10	44	30
PingSym2	1	43	7	140	30
ConcDB	1	54	6	265	20
DistDB	2	42	2	218	20
Firewall	1	45	9	201	30
LockServer	1	28	12	109	30
MapReduce	2	64	4	205	30
Parikh	0	35	-	173	20
Registry	1	40	10	171	30
TwoBuyers	0	59	-	332	20
2PCommit	1	47	6	281	50
WorkSteal	2	39	5	141	40
Theque	3	576	3	1443	100

Table 6.1. Brisk and Spin results. The first grouping is a set of micro-benchmarks, the second are ported from related work, and Theque is our own case study. It takes at most 100 ms for Brisk to rewrite a program into its canonical sequentialization.



Master:
AllocBlob(name)
PutBlob(name, data)
GetBlob(name)
AddTag(tag, refs)
GetTag(tag)

Tag Server:
AddTag(name)
GetTag(tag)

Data Server:
PutBlob(name, data)
GetBlob(name)

Fig. 6.1. Theque messages (top) and remote procedure calls (bottom). An arrow from A to B indicates that A sends B messages. The dashed line surrounds the services comprising Theque.

to run without consuming more than 8 GB of RAM or taking more than 60 seconds to complete. For programs containing *multiple* parameters, we instantiate each parameter with a value of the same size. We ran Spin (optimized for safety checks with `spin -run -safety`) to to check for invalid end states. We ran our experiments on a 2.8GHz Intel® Xeon® computer.

Benchmarks We evaluate our approach with the following benchmarks:

- ex3, ex2, PingDet, PingSym, PingIter, and PingSym2 are microbenchmarks from § 2.
- ConcDB [D'Osualdo et al. 2012], Parikh [D'Osualdo et al. 2012] and DistDB [Marlow 2012] are implementations of key-value stores.
- Firewall [D'Osualdo et al. 2012] is a firewall mediating communication between a worker and a server.
- Registry [Tasharofi et al. 2012] contains a master process, n workers and a registry server that is used to keep track of the processes registered in the system.
- WorkSteal is the first phase of MapReduce from fig. 5.5, where n workers compete for k jobs.
- LockServer [Wilcox et al. 2015] implements a lock service.

- 2PCommit [Lampson and Sturgis 1976] is the classic two-phase-commit protocol for atomic commitment.
- TwoBuyers [Honda et al. 2008] is a protocol where two "buyers" negotiate a purchase from a "seller."
- Theque is a prototype distributed file store that we developed, inspired by Disco [Disco-Project 2017].

Case Study: Theque Cluster File System To demonstrate that realistic systems can be developed using Brisk, we built a prototype cluster storage system, inspired by the Disco cluster file system, called Theque. A Theque instance stores two types of data: (1) immutable, write-once *blobs* of data and (2) mutable *tags*, which are metadata. Tags are references to blobs, other tags, or both. Figure 6.1 shows the processes and remote procedure calls. Clients issue requests to a master process. Operations on blobs are forwarded to a set of data servers, tag operations are handled by a set of tag servers.

Theque is naturally symmetrically non-deterministic. Each type of process is responsible for a *class* of operations: the master for coordination, the data servers for data operations and the tag servers for meta-data operations. The Theque RPCs are partitioned into different Haskell data types. For example, the data type `MasterAPI` defines the types of messages that the master process expects in its main event handler (*i.e.*, `AllocBlob`, `PutBlob`, `GetBlob`, `AddTag`, and `GetTag` from fig. 6.1). Likewise, `TagNodeAPI` defines the types of messages for tag nodes. Since the respective event handlers expect messages of different types, it is easy to prove Theque has symmetric non-determinism. For instance, tag nodes trivially cannot receive messages sent by clients, as the messages are of type `MasterAPI` which does not match the expected type `TagNodeAPI`.

In order to create a closed system, we created a module that spawns all of the necessary processes, including an unbounded number of clients that issue a non-deterministically chosen request to the Master server. We then ran Brisk on this module to verify first that its implementation is indeed non-deterministic up to symmetry and second to compute its canonical sequentialization.

Results In Table 6.1, we compare the results of running Brisk to those of verifying deadlock freedom on finite instances with Spin. The column labeled **#Param** indicates the number of parameterized components (sets of processes or indices). **Spin N** indicates the *smallest* parameter instantiation such that the Spin verification exceeds either its memory or time bound. We omit this number for the benchmarks that are not parametric, and for ex2, which is essentially deterministic. **IceT #Term** indicates the size of the intermediate IceT program (compiled from Haskell source) measured as the number of constant and function symbol occurrences. Finally, in the column labeled **Brisk Time**, we report the time for Brisk to compute the program's canonical sequentialization. Our results show that for protocols involving asynchronous communication, the Spin verification run suffers from the expected state space explosion for modest instantiations. In contrast, Brisk computes canonical sequentializations in less than 100 ms. In order to test how Brisk performs on faulty programs, we manually added errors such as missing sends, missing receives and sends to non-existent PID's to our benchmarks. We found that Brisk reports those errors in around the same amount of time.

7 RELATED WORK

Next, we situate our approach with related techniques for verifying distributed programs.

Actors and Process Calculi Various techniques have been designed to reason about actor programs. Tools for model checking Erlang programs, *e.g.*, McErlang [Fredlund and Svensson 2007], are inherently limited to finite state systems. To tackle the infinite state case, [Huch 1999] creates finite models using abstract interpretation. However, this work does not handle programs with *infinite* *control*, *i.e.*, an unbounded number of *processes*. In contrast, [D'Osualdo et al. 2013, 2012] convert an Erlang program into a (infinite-state) vector addition system which can only be checked for *coverability* properties, thus excluding *e.g.*, deadlock-freedom. [Summers and Müller 2016] describes a logic for reasoning about safety and liveness properties of actor programs. Their setting differs from ours in that they consider actors that are *always* ready to receive, whereas our programs *block* until a suitable message arrives.

In an actor setting, programs often violate symmetric non-determinism as they receive messages from different non-symmetric processes at the same program location: usually, each process consists of a single message handler with multiple continuations for different senders and message types. Moreover, both actor programs and process calculi feature dynamic process creation. It would be interesting to investigate whether symmetric non-determinism can be exploited in such a setting. For instance, we speculate that, if the underlying protocol logic does not crucially depend on bona fide races, one might often be able to (automatically) refactor the programs to only contain symmetric races, *e.g.*, along the lines described in § 2.3. Similarly, we conjecture that our notion of unfolding can be generalized to handle dynamic process creation.

Verification of MPI programs It has been shown [Siegel 2005; Siegel and Avrunin 2005; Siegel and Gopalakrishnan 2011; Siegel and Zirkel [n. d.]] that finite-state MPI programs, subject to some restrictions, may be efficiently verified by model checkers that consider only synchronous communications. The "source-specific" condition of [Siegel 2005] is related to symmetric non-determinism. Their condition allows receiving sends from various locations, if at each point of an execution there is only *one* possible source that can be received from. In contrast, symmetric non-determinism allows symmetric races which permit receiving from *multiple sources*. Again, all above methods do not consider systems with infinite state or infinite control.

Session Types The Session Types [Charalambides et al. 2016; Deniélou et al. 2012; Honda 1993; Honda et al. 2012, 2008] family of work projects a user-provided global protocol into a set of types for the various local processes of a program. Session types have found applications in many settings, including MPI programs [Honda et al. 2016], GO [Ng and Yoshida 2016] and HASKELL [Orchard and Yoshida 2016]. Unlike our approach, session types require *the user* to specify the global interaction protocol. Moreover, session types do not allow for *symmetric races* and hence lack the expressiveness needed to check some of our benchmarks (e.g., EX3 or MapReduce).

Partial-Order Reduction Exploiting commutativity in state transition graphs is crucial in mitigating the state-explosion problem in model checking. Partial-order reduction methods [Abdulla et al. 2014; Flanagan and Godefroid 2005; Godefroid et al. 1996] explore representative traces in acyclic state spaces. These methods are based on the idea that often a concurrent or distributed system only defines a *partial ordering* of events. Thus, whenever there is no ordering between two given events, it is sufficient to explore an arbitrary representative one. This idea is embodied in the concept of *persistent sets* [Godefroid et al. 1996] which is closely related to our use of left-movers. A persistent set, for a given state, is a *subset* of all enabled transition (*i.e.*, those that can be executed in this state) such that (1) the transition cannot be disabled by other transitions and (2) it *commutes* with all transitions reachable through transitions that are *not* in the set. Assuming one can efficiently compute persistent sets, it is sufficient to explore, for each state, only the transitions in its persistent set in order to cover all halted states. A related notion is that of *sleep sets* [Godefroid et al. 1996] which avoid re-exploration of previously explored transitions.

The above partial-order techniques do not directly translate to parameterized or infinite state systems, as persistent or sleep sets in these settings would be *unbounded* in size. While some work has addressed the challenge of infinite state systems [Cimatti et al. 2011; Wachter et al. 2013], these techniques focus on systems with a *bounded* number of processes. In contrast, canonical sequentialization applies to *parameterized* as well as infinite state systems.

Reductions The theory of reductions [Lipton 1975] has been used to identify groups of program statements that appear to act atomically with respect to other threads [Elmas et al. 2009; Flanagan and Qadeer 2003]. Our work exploits the insights from [Lipton 1975] in order to rewrite programs into their canonical sequentialization. The work in [Desai et al. 2014] is similar to ours in that our rules explore traces where buffers are small (by moving receives right after sends). However, our work checks *global* system configurations whereas [Desai et al. 2014] is concerned with *local* states, only and hence cannot account for deadlocks. Moreover, [Desai et al. 2014] does not consider unbounded numbers of processes. A condition for transforming asynchronous finite-state programs into synchronous programs is presented in [Basu et al. 2012], however, it is too restrictive for our benchmarks.

Parameterized Verification Recent work focuses on automatically inferring counting arguments in the form of counting automata [Farzan et al. 2014] or by synthesizing descriptions of sets and referring to their cardinalities [Gleissenthall et al. 2016]. There has been much work on inferring universally quantified invariants [Bjørner et al. 2013; Gleissenthall et al. 2016; Monniaux and Alberti 2015; Sanchez et al. 2012]; despite recent progress, reliably synthesising both quantified invariant and auxiliary state remains a challenge.

In general, clever abstractions are required to verify parameterized systems by model checking. Counter abstractions [Pnueli et al. 2002] are a classic way to exploit symmetry in parameterized systems. The work described in Konnov et al. [2015] encodes threshold-based distributed algorithms into counter systems and uses acceleration and an SMT-based model checker to verify them. However, the programs we consider require tracking the *contents* of messages (e.g., PID's as in example ex4) which is challenging for counter-based approaches. The method of invisible invariants [Arons et al. 2001; Pnueli et al. 2001] guesses candidate invariants from small instantiations, but is limited to finite data domains.

We exploit the symmetry of groups of processes by treating identifiers as scalar sets [Norris IP and Dill 1996]. Our *unfold* can be seen to perform a data type reduction [McMillan 1999] (an instance of abstract interpretation [Cousot and Cousot 1977]).

User-Guided Verification Several recent papers focus on proving high-level correctness properties of distributed systems. Protocols specified in the Verdi framework [Wilcox et al. 2015] require the user to provide an inductive invariant. This quite significant burden is somewhat mitigated by [Padon et al. 2016], in which the user guides the system toward an inductive invariant by examining counterexamples. [Drăgoi et al. 2014] develops a logic based on the HO model (a synchronous execution model with benign faults) and [Drăgoi et al. 2016] implements a DSL for developing and verifying systems in this model. Like our approach, the synchronous semantics of the HO model simplify verification. However, unlike the above, which require user-provided invariants, we focus on generic lower-level properties (*e.g.*, deadlock-freedom) and verify programs *automatically*.

8 CONCLUSIONS AND FUTURE WORK

In this paper we have described *canonical sequentialization*, a new approach to verifying parameterized distributed programs. We have implemented canonical sequentialization in Brisk, which rewrites distributed programs written in Haskell. Brisk verifies the *unbounded* versions of distributed programs in tens of *milliseconds*, yielding the first concurrency verification tool that is fast enough to be integrated into a design-implement-check cycle. For future work, we would like to address errors due to node failures or unreliable networks. Second, we do not yet handle programs with parameterized topologies such as rings and dynamic thread creation. We expect to be able to extend our notion of *unfolding* to handle these types of programs. Finally, it would be interesting to compare how our approach helps programmers understand and ultimately fix errors in their code versus more traditional approaches.

ACKNOWLEDGMENTS

This work was supported by NSF grants CCF-1422471, CCF-1223850, CNS-1514435 and a generous gift from Microsoft Research. We thank Heidy Khlaaf, Ismail Kuru, Eric Seidel, Nik Sultana, and the anonymous referees for their helpful comments and suggestions for improving the paper.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (May 1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 373–384.

Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. 2001. *Parameterized Verification with Automatically Computed Inductive Assertions.* Springer Berlin Heidelberg, Berlin, Heidelberg, 221–234.

Samik Basu, Tevfik Bultan, and Meriem Ouederni. 2012. Synchronizability for verification of asynchronously communicating systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 56–71.

Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS.*

Luca Cardelli. 1984. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming.* ACM, 208–217.

Minas Charalambides, Peter Dinges, and Gul Agha. 2016. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Science of Computer Programming* 115 (2016), 100–126.

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. 2011. Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In *TACAS.*

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 238–252.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI.*

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:6)2012

Ankush Desai, Pranav Garg, and P Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 709–725.

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.

Disco-Project. 2017. Disco MapReduce. http://discoproject.org. (April 2017).

E. D'Osualdo, J. Kochems, and C.-H. L. Ong. 2013. Automatic Verification of Erlang-Style Concurrency. In *Proceedings of the 20th Static Analysis Symposium (SAS'13).* Springer-Verlag.

Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. 2012. Soter: An Automatic Safety Verifier for Erlang. In *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions (AGERE! 2012).* ACM, New York, NY, USA, 137–140. https://doi.org/10.1145/2414639.2414658

Cezara Drăgoi, Thomas A Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A logic-based framework for verifying consensus algorithms. In *International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 161–181.

Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. 2016. PSYNC: A partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices* 51, 1 (2016), 400–415.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A Calculus of Atomic Actions. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09).* ACM, New York, NY, USA, 2–15. https://doi.org/10.1145/1480881.1480885

Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. 2011. Towards Haskell in the cloud. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 118–129.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2014. Proofs that count. *ACM SIGPLAN Notices* 49, 1 (2014), 151–164.

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, Vol. 40. ACM, 110–121.

Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 338–349.

Lars-Åke Fredlund and Hans Svensson. 2007. McErlang: a model checker for a distributed functional programming language. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 125–136.

Klaus v Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. 2016. Cardinalities and universal quantifiers for verifying parameterized systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 599–613.

Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. 1996. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Vol. 1032. Springer Heidelberg.

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR*.

Kohei Honda, Eduardo RB Marques, Francisco Martins, Nicholas Ng, Vasco T Vasconcelos, and Nobuko Yoshida. 2012. Verification of MPI programs using session types. In *European MPI Users' Group Meeting*. Springer, 291–293.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types.. In *POPL*.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (March 2016), 67 pages. https://doi.org/10.1145/2827695

Frank Huch. 1999. Verification of Erlang Programs using Abstract Interpretation and Model Checking. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.* 261–272. https://doi.org/10.1145/317636.317908

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 *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*

Igor Konnov, Helmut Veith, and Josef Widder. 2015. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *International Conference on Computer Aided Verification*. Springer, 85–102.

Butler Lampson and Howard E. Sturgis. 1976. Crash Recovery in a Distributed Data Storage System. In *Technical report XEROX Palo Alto Research Center.*

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. https://doi.org/10.1145/361227.361234

Simon Marlow. 2012. Parallel and concurrent programming in Haskell. In *Central European Functional Programming School*. Springer, 339–401.

Kenneth L. McMillan. 1999. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. 219–234. https://doi.org/10.1007/3-540-48153-2_17

Neil Mitchell and Colin Runciman. 2009. Losing functions without gaining data: another look at defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. ACM, 13–24.

David Monniaux and Francesco Alberti. 2015. *A Simple Abstraction of Arrays and Maps by Program Translation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 217–234. https://doi.org/10.1007/978-3-662-48288-9_13

Nicholas Ng and Nobuko Yoshida. 2016. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *CC*.

C. Norris IP and David L. Dill. 1996. Better verification through symmetry. *Formal Methods in System Design* 9, 1 (1996), 41–75. https://doi.org/10.1007/BF00625968

Dominic Orchard and Nobuko Yoshida. 2016. Effects as sessions, sessions as effects. In *POPL*.

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*. ACM, 614–630.

Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. 2001. Automatic deductive verification with invisible invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 82–97.

Amir Pnueli, Jessie Xu, and Lenore Zuck. 2002. Liveness with (0, 1, ∞)-counter abstraction. In *International Conference on Computer Aided Verification*. Springer, 107–122.

Alejandro Sanchez, Sriram Sankaranarayanan, César Sánchez, and Bor-Yuh Evan Chang. 2012. *Invariant Generation for Parametrized Systems Using Self-reflection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 146–163. https://doi.org/10.1007/978-3-642-33125-1_12

Stephen F Siegel. 2005. Efficient verification of halting properties for MPI programs with wildcard receives. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 413–429.

Stephen F Siegel and George S Avrunin. 2005. Modeling wildcard-free MPI programs for verification. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 95–106.

Stephen F Siegel and Ganesh Gopalakrishnan. 2011. Formal analysis of message passing. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2–18.

Stephen F. Siegel and Timothy K. Zirkel. [n. d.]. Automatic Formal Verification of MPI-Based Parallel Programs. 309–310.

Alexander J. Summers and Peter Müller. 2016. Actor Services. In *ESOP*.

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 *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems (FMOODS'12/FORTE'12)*. Springer-Verlag, Berlin, Heidelberg, 219–234. https://doi.org/10.1007/978-3-642-30793-5_14

Bjorn Wachter, Daniel Kroening, and Joel Ouaknine. 2013. Verifying multi-threaded software with Impact. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 210–217.

James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 357–368.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. 213–228.