Title
Sequentialization and Synchronization for Distributed Programs

Permalink
https://escholarship.org/uc/item/0hh6k2zn

Author
Bakst, Alexander Goldberg

Publication Date
2017

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA, SAN DIEGO

Sequentialization and Synchronization for Distributed Programs

A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Alexander Goldberg Bakst

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Sorin Lerner
Professor Todd Millstein
Professor Jeffrey Remmel
Professor Geoffrey Voelker

2017
The Dissertation of Alexander Goldberg Bakst is approved and is acceptable in quality and form for publication on microfilm and electronically:

________________________________________

________________________________________

________________________________________

________________________________________

Chair

University of California, San Diego

2017
DEDICATION

For Mom, Dad, and Sarey.
EPIGRAPH

The key to understanding complicated things is knowing what not to look at.

_Gerald Jay Sussman_
# TABLE OF CONTENTS

Signature Page  ................................................................. iii
Dedication  ................................................................. iv
Epigraph  ................................................................. v
Table of Contents  ......................................................... vi
List of Figures  ........................................................ viii
List of Tables  ........................................................ x
Acknowledgements  ..................................................... xi
Vita  ............................................................. xiii
Abstract of the Dissertation  ............................................... xiv

Chapter 1  Introduction  ................................................... 1
  1.1 Verification of Distributed Systems And Protocols  ............ 2
  1.2 Toward Automated Verification  .................................... 4
    1.2.1 Iterating over a list  ........................................ 5
    1.2.2 Verifying Distributed Programs  ......................... 9
    1.2.3 Discussion  ................................................ 10
  1.3 Related Approaches to Automated Verification ............... 11
  1.4 Contributions  .................................................. 15

Chapter 2  Message Passing Programs  .................................. 17
  2.1 Overview  ....................................................... 17
  2.2 Syntax and Semantics  ........................................... 18
    2.2.1 Syntax  .................................................. 18
    2.2.2 Semantics  ............................................. 20
  2.3 Tagged Programs  ................................................. 23
  2.4 Reductions for $I_S$  ........................................... 24
  2.5 Summary  ....................................................... 27

Chapter 3  Canonical Sequentializations  .............................. 29
  3.1 Overview  ....................................................... 31
  3.1.1 A Task Distribution Service  ............................... 31
  3.1.2 Main Ideas  ................................................ 36
  3.1.3 Expressiveness  ............................................ 40
  3.2 Symmetric Non-determinism  ................................... 43
  3.3 Rewrite Rules  .................................................. 44
    3.3.1 Basic Rules  ............................................ 45
    3.3.2 Loops, Unfolding and Conditionals  .................... 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3 Residual Processes</td>
<td>52</td>
</tr>
<tr>
<td>3.3.4 Correctness</td>
<td>55</td>
</tr>
<tr>
<td>3.4 Canonical Sequentialization of I₅ Programs</td>
<td>56</td>
</tr>
<tr>
<td>3.5 From HASKELL to I₅</td>
<td>57</td>
</tr>
<tr>
<td>3.5.1 Syntax and Semantics of λₘ</td>
<td>57</td>
</tr>
<tr>
<td>3.5.2 Translation to I₅</td>
<td>60</td>
</tr>
<tr>
<td>3.5.3 Correctness of the Translation</td>
<td>66</td>
</tr>
<tr>
<td>3.6 Evaluation</td>
<td>69</td>
</tr>
<tr>
<td>3.7 Summary</td>
<td>73</td>
</tr>
<tr>
<td>Chapter 4 Pretend Synchrony</td>
<td>75</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>77</td>
</tr>
<tr>
<td>4.2 Verification with I₅</td>
<td>82</td>
</tr>
<tr>
<td>4.3 Rewrite Rules</td>
<td>84</td>
</tr>
<tr>
<td>4.3.1 Rewriting loop-free programs</td>
<td>85</td>
</tr>
<tr>
<td>4.3.2 Communication between a Single Process and a Group</td>
<td>86</td>
</tr>
<tr>
<td>4.3.3 Communication between Groups of Processes</td>
<td>90</td>
</tr>
<tr>
<td>4.3.4 Rewrite Rule Correctness</td>
<td>92</td>
</tr>
<tr>
<td>4.3.5 Extensions</td>
<td>94</td>
</tr>
<tr>
<td>4.4 Generating Verification Conditions</td>
<td>95</td>
</tr>
<tr>
<td>4.5 Case Studies: Synchronous Invariants for Distributed Protocols</td>
<td>100</td>
</tr>
<tr>
<td>4.5.1 Raft Leader Election</td>
<td>101</td>
</tr>
<tr>
<td>4.5.2 Single-decree Paxos</td>
<td>104</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>107</td>
</tr>
<tr>
<td>Chapter 5 Conclusion</td>
<td>109</td>
</tr>
<tr>
<td>5.1 Future Directions</td>
<td>110</td>
</tr>
<tr>
<td>Appendix A Translation from λₘ to I₅</td>
<td>113</td>
</tr>
<tr>
<td>Appendix B Sequentialization</td>
<td>120</td>
</tr>
<tr>
<td>Bibliography</td>
<td>130</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.1</td>
<td>Program ManyPings: A server collects &quot;ping&quot; messages from a set of clients.</td>
</tr>
<tr>
<td>1.2</td>
<td>Printing a list of messages.</td>
</tr>
<tr>
<td>1.3</td>
<td>Closing a list of files with heap summaries corresponding to labeled program points.</td>
</tr>
<tr>
<td>1.4</td>
<td>Simplification of ManyPings.</td>
</tr>
<tr>
<td>2.1</td>
<td>$I_S$ Syntax.</td>
</tr>
<tr>
<td>2.2</td>
<td>Definition of $A : ✓$.</td>
</tr>
<tr>
<td>2.3</td>
<td>Definition of $\text{Procs}(A)$.</td>
</tr>
<tr>
<td>2.4</td>
<td>Definition of $A \times B$.</td>
</tr>
<tr>
<td>2.5</td>
<td>Congruence relation $A \equiv B$.</td>
</tr>
<tr>
<td>2.6</td>
<td>$I_S$ Semantic Definitions</td>
</tr>
<tr>
<td>2.7</td>
<td>$I_S$ Semantics: $C \xrightarrow{T_{1_S}} C$</td>
</tr>
<tr>
<td>3.1</td>
<td>A task distribution service.</td>
</tr>
<tr>
<td>3.2</td>
<td>Canonical Sequentialization of the task distribution system in Figure 3.1 as computed by BRISK.</td>
</tr>
<tr>
<td>3.3</td>
<td>Syntax for context $Γ$.</td>
</tr>
<tr>
<td>3.4</td>
<td>Rewrite Rules (Basic Statements)</td>
</tr>
<tr>
<td>3.5</td>
<td>Rewrite Rules (Iteration over sets of processes).</td>
</tr>
<tr>
<td>3.6</td>
<td>Rewrite Rules (Iteration over sets of indices).</td>
</tr>
<tr>
<td>3.7</td>
<td>Rewrite Rules (Unfold)</td>
</tr>
<tr>
<td>3.8</td>
<td>Rewrite Rules (Residue)</td>
</tr>
<tr>
<td>3.9</td>
<td>Rewrite Rules (Branch)</td>
</tr>
<tr>
<td>3.10</td>
<td>$λ_m$ syntax</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.11</td>
<td>Semantics of $\lambda_m$ terms</td>
</tr>
<tr>
<td>3.12</td>
<td>MapReduce implemented in HASKELL</td>
</tr>
<tr>
<td>3.13</td>
<td>Transformation of $\lambda_m$ to $I_S$</td>
</tr>
<tr>
<td>3.14</td>
<td>Well-formed $\lambda_m$ terms</td>
</tr>
<tr>
<td>3.15</td>
<td>Tagged $I_S$ model of MAPREDUCE program from Figure 3.12</td>
</tr>
<tr>
<td>3.16</td>
<td>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.</td>
</tr>
<tr>
<td>4.1</td>
<td>Two phase commit. Arrows denote the flow of messages during each logical phase of the protocol. The boxes of code at either end of an arrow will be synchronized by our method. Each for loop is annotated with an invariant $[I]$.</td>
</tr>
<tr>
<td>4.2</td>
<td>Synchronization of two phase commit.</td>
</tr>
<tr>
<td>4.3</td>
<td>$I_S$ Syntax. Changes with respect to Figure 2.1 highlighted in gray.</td>
</tr>
<tr>
<td>4.4</td>
<td>Syntax for context $\Gamma$. Changes with respect to Figure 3.3 highlighted in gray.</td>
</tr>
<tr>
<td>4.5</td>
<td>Rewrite Rules (Basic Statements)</td>
</tr>
<tr>
<td>4.6</td>
<td>Rewrite Rules (Iteration over sets of processes and unfolding)</td>
</tr>
<tr>
<td>4.7</td>
<td>Selected rules establishing control flow independence.</td>
</tr>
<tr>
<td>4.8</td>
<td>Rewrite Rules (case splitting on a single process)</td>
</tr>
<tr>
<td>4.9</td>
<td>Reactive while loop.</td>
</tr>
<tr>
<td>4.10</td>
<td>Weakest precondition computation. $V(A)$ denotes the variables that may be assigned in $A$.</td>
</tr>
<tr>
<td>4.11</td>
<td>Simplified Raft leader election algorithm (asynchronous version)</td>
</tr>
<tr>
<td>4.12</td>
<td>Single-decree Paxos (asynchronous version)</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3.1.  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.  ...........................................  70

Table 4.1.  CHAI benchmark results. ................................................................. 101
ACKNOWLEDGEMENTS

By a series of fortunate events, the past 6 years have graced me with the profound pleasures that are wise mentors, distinguished colleagues, and dear friends.

First, great thanks are due to my advisor, Ranjit Jhala. With his particular blend of patience, frankness, and enthusiasm, Ranjit has above all else instilled in me an invaluable sense of taste by which to understand and evaluate problems and their solutions. Thanks are due to Sorin Lerner, not only for his mentorship and cheap laughs at his expense, but also for stubbornly refusing to hang up the phone until I agreed to visit UCSD. I am additionally thankful to Todd Millstein, Jeffrey Remmel, and Geoff Voelker for serving on my committee. I had the distinct pleasure of working with Chris Hawblitzel, who offered precious tutelage early in my graduate career.

A source of nightmares is imagining a second go at graduate school, but without the great collaborators and mentors. Thank you to Pat Rondon and Ming Kawaguchi, not only for deftly bringing me up to speed on all things Liquid Types, but also for nurturing a light-hearted approach to curmudgeonery. Thanks to Ravi Chugh for patiently watching me stare at the whiteboard, vexed by my own mess, and for removing all obstacles to giving yoga a shot. To Zach Tatlock, thank you for all of the goofs and spoofs, nurturing my physical dependence on caffeine, and your sometimes downright frustrating commitment to honest and logical dissection of my offhand remarks. To Klaus Gleissenthall and Rami Kici, thank you for collaborating on the bulk of the work in this dissertation.

The UCSD CSE community is a treasure trove of wonderful people whom I am fortunate to call friends. In particular, Dimitar Bounov, Neha Chachra, Alan Leung, Valentin Robert, Eric Seidel, Niki Vazou, and Panagiotis Vekris all made graduate school more than bearable.

It is impossible to overstate the overwhelming support of Mom, Dad, and Sis. Finally, thank you to Maya, who refuses to let my brain rest for even a moment, and positively ruins all of my bad moods.
Published Works Adapted in this Dissertation

Chapter 2, in part, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is adapted from the material as it appears in Klaus v. Gleissenthall, Alexander Bakst, Rami Gökhan Kıcı, and Ranjit Jhala. Pretend Synchrony. In submission to the ACM SIGPLAN International Conference on Principles of Programming Languages, POPL 2018. The dissertation author was the primary investigator and author of this paper.
VITA

2008 Bachelor of Science, Massachusetts Institute of Technology
2009 Master of Engineering, Massachusetts Institute of Technology
2011–2017 Research Assistant, University of California, San Diego
2017 Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017


ABSTRACT OF THE DISSERTATION

Sequentialization and Synchronization for Distributed Programs

by

Alexander Goldberg Bakst

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Ranjit Jhala, Chair

Distributed systems are essential for building services that can handle the ever increasing number of people and devices connected to the internet as well as the associated growth in data accumulation. However, building distributed programs is hard, and building confidence in the correctness of an algorithm or implementation is harder still. One fundamental reason is the highly asynchronous nature of distributed execution. Timing differences caused by network delays and variation in compute power can trigger behaviors that were unanticipated by the programmer.

Unfortunately, techniques for building confidence are all up against the same problem: the combinatorial explosion in the number of behaviors of a distributed system.
Testing and model checking techniques can not hope to weed out all behaviors when the state space is infinite. At the other end of the spectrum, constructing proofs by hand is a daunting task, as inductive invariants must simultaneously account not only for the algorithm-specific properties, but also the asynchronous nature of distributed computation.

In this dissertation, we take the view that many distributed programs are actually designed in such a way that the developer can safely reason about a small number of representative, synchronous executions. To support this claim, we first identify a property of message passing programs called symmetric non-determinism. Intuitively, the observable behavior of a program with symmetric non-determinism is insensitive to underlying network and processor non-determinism. Second, we develop an algorithm for transforming programs with symmetric non-determinism into an equivalent sequential program, called its canonical sequentialization. Our experiments show that canonical sequentializations can be computed quickly enough for eventual use in the design-implement-check cycle. Finally, we generalize the previous approach and present an algorithm for proving the correctness of distributed programs by automatically transforming them into an equivalent synchronized program. In this setting, rather than reasoning about message buffers, the developer writes loop invariants as synchronous assertions. We have implemented both of these algorithms as tools that demonstrate a reduction in the manual annotation burden for distributed systems.
Chapter 1

Introduction

Asynchronous distributed systems are notoriously hard to implement correctly as timing differences caused by network delays or variations in execution time may trigger behaviours that were neither intended nor anticipated by the programmer. In case the design of algorithms and protocols for concurrent, distributed execution was not already tricky business, ensuring the correctness of implementations tailored to a particular application adds even more complexity. One approach to mitigate the challenge of developing correct protocols and implementations is to construct workloads and stress tests to coax out corner cases hiding behind unanticipated schedules [19, 83, 37]. Unfortunately, these efforts are doomed to come up short. Due to infinite data domains and the fact that programs are often parameterized, i.e., the number of participating nodes is not known at compile time, the number of possible schedules is infinite, making their enumeration a hopeless endeavor. As a result, programmers are left to guess, without any guarantee that their tests actually cover all errors.

A principled alternative is to construct mathematical proofs of correctness, in which one formally verifies that a system adheres to its specification. However, formal verification of parameterized systems is an entirely different kettle of fish. Verifying a system amounts to (1) defining a good set of executions, known as the specification, and then (2) proving that all of the executions of the system are contained within the specification. Often this takes the shape of an inductive argument, in which the program verifier (an algorithm or a human) attempts to discover a program invariant, which is a
set of states closed under all of the state transitions that the program can make. If the invariant contains the initial state of the program and is contained in the specification, then the system is guaranteed to satisfy its specification. Due to the asynchronous nature of communication, invariants must often explicitly enumerate possible schedules by case splitting over the joint global states of all participants. This makes the invariants both unwieldy and hard to come up with. Even worse, asynchronous invariants must reason about the contents of (unbounded) message buffers which renders even checking the correctness of candidate invariants undecidable.

1.1 Verification of Distributed Systems And Protocols

**Specification in proof assistants.** Specifying distributed algorithms requires extreme care. Machine-checkable proofs of correctness are essential for improving confidence in the correctness of a system, as even meticulously hand-checked proofs can be incorrect [84]. Thus, a number of approaches have been developed to aid in developing protocols and algorithms, and in particular proving their correctness. For example, PVS [57] has been used to verify Paxos [41], and and TLA+ [42] and TLAPS [14] have been used to specify and prove the correctness of Multi-Paxos [9]. This strategy imposes relatively few restrictions on the properties that can be specified and hence, verified. The trade-off is that the proof must be painstakingly constructed by hand: the specification in [9] is only 158 lines but the proof is 1136 lines. To mitigate this problem somewhat, Ivy [58] allows users to discover inductive invariants by examining counterexamples produced by bounded checking of algorithms specified in a custom modeling language. The language, RML, is restricted in order to guarantee that the verification conditions for (loop-free) programs fall into the effectively propositional fragment and are hence decidable [60].

**Frameworks for verifying distributed systems.** Another approach is to build custom program logics and frameworks specifically for reasoning about and constructing distributed systems, and then discharge proof obligations through prover backends.
The Verdi framework [81], which uses Coq [49] as a proof assistant, has been used to prove linearizability for Raft [82]. EventML [62], using the NuPRL proof assistant [13], has been used to verify the correctness of a Paxos implementation [67]. Recently, a type theoretic approach based on Hoare Type Theory [80] was proposed, which enables the user to compose different protocols and hence make correctness proofs modular. Related deductive techniques exploit advances in automated theorem proving to reduce the manual proof burden. Typically, the user provides an inductive invariant, which the system then uses to construct a proof of correctness. For example, the Dafny [44] language was used to build and verify a replicated state machine library based on Paxos in [31]. Dafny automatically proves the system correct, given substantial annotation by the user (about 9 lines of annotation for each line of code).

Programming model simplification. One of the principal obstacles to proving a system correct is the asynchronous nature of communication and execution. The techniques thus far require the programmer to reason about message buffers or to manually prove refinement of intermediate layers of abstraction between the specification and implementation. Thus, one alternative strategy is to design programming models specifically purposed for simplifying reasoning. The Heard-Of (HO) model, proposed in [11], introduced a new model of distributed computation in which communication is divided into rounds. Messages that arrive late are considered lost, and so the model effectively induces synchronous communication. The authors argue that this leads to simpler proofs. A logic for reasoning about the HO model is presented in [22] and a domain-specific language for developing and verifying programs in the HO model is described in [23]. The SCons language described in [46] is designed to be expressive enough to implement consensus algorithms such as Paxos while obtaining small cutoff bounds: finite systems whose correctness implies the correctness of the parameterized system. Finally, the work in [40, 38, 39] focuses on threshold-based fault tolerant distributed algorithms, which are algorithms that use threshold guards (i.e., requirements that a message be received from at least \( t \) distinct processes). This approach models algorithms as threshold automata, which
use integer counters to track the number of processes in a given state, and hence can represent parameterized systems. These approaches reduce verifying a threshold automata to checking a finite set of representative executions that can be checked automatically using a satisfiability-modulo-theory (SMT) solver.

1.2 Toward Automated Verification

Ultimately, the challenge is to devise how to manage the complexity of the number of possible program executions in an asynchronous, concurrent system. One approach asks the user to provide summaries of the executions in the form of program invariants, the other restricts the problem domain to (small) finite systems. In either case, verification crucially depends on reasoning about the protocol structure, i.e., which messages are destined for which processes, and the contents of messages. Thus, the verifier must reason simultaneously about message contents and messaging structure. Doing both at the same time complicates reasoning when proving even shallow properties.

Consider the program ManyPings in Figure 1.1, in which each member of an unbounded set ps of client processes asynchronously send a “ping” message to a server process. The server uses a blocking recv() statement to retrieve the first available

```c
process server(pid_set ps) {
    string x;

    for (p : ps) {
        x = recv();
    }
}

process client(pid server) {
    send(server, "ping");
}
```

**Figure 1.1.** Program ManyPings: A server collects “ping” messages from a set of clients.
message, binding it to $x$. How do we prove that the composition of the server and clients does not deadlock – that is, that the server does not get “stuck” at a particular `recv()`?

For even small instantiations of $ps$, the state space can grow quite large. For example, for a single client $p$ in $ps$, either (a) $p$ has not sent a message to the server yet, (b) it has and the message has not been received, or (c) the server has received $p$’s message. To make matters worse, in the general case we must explore the states resulting from the order in which messages queue up at the server. However, these details are irrelevant as the behavior of the server does not depend on the order in which it receives messages from the clients.

The intuition is as follows. We can partition the clients into two disjoint sets: those whose messages the server has yet to receive and those who the server has heard from. Initially, the latter set is empty. On each iteration of the loop, a single process “moves” from the first set to the second set, since the union of the two sets must equal $ps$ and the server could not have received a message from the latter set.

The ability to partition the clients in this way is a consequence of exploiting the structure of the (implicit) messaging protocol, in which the clients only send messages to the server (and not to each other, for example). Having partitioned the clients in this way, we then summarize the interaction of the server with an unbounded number of clients by considering an arbitrary process in isolation. Remarkably, this strategy resembles techniques for automated reasoning about heap-manipulating programs, which we call next.

### 1.2.1 Iterating over a list

Consider the short program in Figure 1.2. The program is a function `print_msgs` that takes a pointer to a linked list of tasks, each a message to be printed only once. The function asserts that the message at the head of the list has not been printed, and then proceeds to print the message string with `printf()`. To mark that the task has been processed, the `done` field is set to 1. Finally, the remaining messages are printed by a
struct list {
  int done;
  char *msg;
  struct list *next;
}

void print_msgs(list *p) {
  if (p != null) {
    assert(!p->done);
    printf(p->msg);
    p->done = 1;
    print_msgs(p->next);
  }
  return;
}

Figure 1.2. Printing a list of messages.

recursive call of print_msgs() on p->next, the tail of the list.

**Specification.** We would like to ensure that on return, every node reachable from p has been processed (i.e., its done field is set to 1). As the program will crash if the function encounters an element whose done field is set to 1 before the message is printed, the specification should additionally require that on entry to print_msgs(), every node reachable from p has its done field set to 0.

**Verification.** To verify the correctness of print_msgs(), we first devise a finite representation or abstraction of the heap, as the input to print_msgs() is a list of unbounded size. One family of approaches (e.g., [4, 79]) abstracts heaps by a composition of disjoint sub-heaps which are either (a) a precise description of values stored at a finite set of addresses or (b) a summary of an unbounded number of memory locations. These abstractions factor out the so-called “physical” description of the heap, describing the internal pointers of a data structure, and the “logical” description of the values stored within the data structure.

We simulate the effect of each statement in print_msgs() by manipulating the
void print_msgs(list *p) {
    if (p != null) {
        assert(!p->done);
        printf(p->msg);
        p->done = 1;
        print_msgs(p->next);
    } else {
        return;
    }
}

Figure 1.3. Closing a list of files with heap summaries corresponding to labeled program points.

abstract heap accordingly. Figure 1.3 contains the close_files() function as before, in addition to several labeled program points. On the right is the abstraction of the heap rooted at p corresponding to each program point. On entry to the function, when p is not null, we assume that p points to a list of unprocessed tasks. We represent this by the abstract heap corresponding to program point 1, which denotes a collection of list nodes whose data values are equal to 0.

Before the first dereference of p at the assertion at program point 2, we unfold the summary to reveal the head element of the list (represented by a box with two cells to denote contiguous addresses in memory), which is disjoint from the remainder of the list. Hence, after the update, the abstract heap at program point 3 records the fact that the first task is complete, but the remainder are not, as there is no danger of
aliasing. Assuming inductively that `print_msgs()` obeys its specification, and since the heap rooted at `p->next` is a list of unprocessed tasks, the heap after the recursive call at point 4 is updated to contain a list of completed tasks. Finally, before exiting the `if` statement, we compute a new summary by `folding` the head node back into the rest of the list. Since the value of `done` at the head and all of the nodes in the tail is 0, the new summary indicates that the list is completely processed.

```
server:
  for (p : ps) { x = recv(); }  // p ∈ ps:
                                  send(server,"ping");
(a) Initial summary.

server:
  x = recv();  // _p:
  send(server,"ping");  // p ∈ _ps:
                        send(server,"ping");
(b) Unfold _p from _ps, where _ps ⊆ ps.

  x = "ping" ;
      p ∈ ps\{_p}:
          send(server,"ping");
(c) Same final states as (b).

  for (p:ps) { x = "ping"; }
(d) Same final states as (a) by generalizing the step from (b) to (c).
```

**Figure 1.4.** Simplification of ManyPings. The initial summary (a) says that each process is at its starting position. To simplify the loop, we consider an arbitrary subset of processes and unfold an arbitrary element in (b). The interaction between the server and _p is reaches the same states as the program in (c). Generalizing this reasoning produces (d).
1.2.2 Verifying Distributed Programs

The verification of `print_msgs()` considers a step in the recursive traversal of a list corresponding to some arbitrary element of the list and generalizes the effect (marking the element “done”) to the entire list. Let us now see how to verify `ManyPings` by employing a similar style of symmetric reasoning.

In distributed systems with no shared memory, most pairs of actions commute with each other, that is, given an initial state they result in the same state when executed in either order. A particular client’s message is independent of all others: by an application of Lipton’s theory of movers [45] we can consider only the program traces in which the message is received immediately after it is sent. However, this reasoning breaks down when there are multiple possible messages that could be received at a given program point. Our key insight is that, although there is non-determinism with respect to whose message the server will receive as its first message, for example, the states resulting from picking a particular client are merely permutations of each other [34]. This observation leads to a property of programs called symmetric non-determinism, which allows us to treat a single message exchange as representative of a set of possible exchanges. Without delving into the details of symmetric non-determinism, let us see how symmetric reasoning allows us to verify `ManyPings` in the same way as `print_msgs()`.

**Specification.** The specification of `ManyPings` is that it does not deadlock. In each iteration of its loop, the server should receive a message from some client. To verify that `ManyPings` meets its specification, we must show that in every halted state, every process has terminated successfully.

**Verification.** For the verification task we adopt the following strategy. We will iteratively apply proof rules that rewrite the program, transforming it into a simpler program – that is, a program with fewer behaviors. Each rewriting step preserves the halted states of the program. We denote the parameterized program as the composition of the concrete server process and a group of client processes.

The initial proof state is shown in Figure 1.4a, which is the composition of the
server program and the set \( ps \) of client processes. The parallel bars (\( || \)) separate distinct groups of processes. While there are many possible transitions in the concrete program, in the proof we focus first on the server’s loop.

To rewrite the server’s loop, we consider an execution of the iteration corresponding to some arbitrary process \( _p \) in the subset of the client processes that the server has not yet interacted with, \( _p \). Unfolding \( _p \) from \( _ps \) produces the new program in Figure 1.4b. Focusing on only the single server loop iteration and the process \( _p \), we conclude that the program in Figure 1.4c deadlocks if Figure 1.4b does, since we can execute the server’s receive immediately after the message is sent by \( _p \). Symmetric non-determinism is critical here, as it allows us to argue that \( _p \) summarizes the possible behaviors of all clients.

We can then generalize the step from Figure 1.4b to Figure 1.4c to every iteration of the loop, producing the final program in Figure 1.4d. Finally, it is easy to see that Figure 1.4d does not get stuck as it contains no blocking operations. Because Figure 1.4d halts in the same states as the initial program, we conclude that the program is deadlock free.

### 1.2.3 Discussion

In the presence of aliasing (in the verification of print_msgs()), it would be unsound to perform the step labeled \( \text{Update} \) in Figure 1.3, in which only the description of the head tail is updated. The heap abstraction, together with the unfold and fold operations, maintains the critical invariant that, for any two nodes \( n_1 \) and \( n_2 \) belonging to a list summary, \( n_1.\text{next} \neq n_2.\text{next} \). Thus, the guaranteed separation between the head and the tail enable us to perform local reasoning, reasoning about updates to the head and the tail in isolation.

Likewise, there are many possible transitions from a given state in a concurrent system. In general, visiting all of the reachable states would involve backtracking in order to consider all of the enabled transitions from an initial state. Symmetric non-
determinism allows us to avoid backtracking in the step from Figure 1.4b to Figure 1.4c, and hence permits a form of local reasoning between a finite set of processes (as with the server and \(_p\)). The dual folding and unfolding steps preserve the symmetric non-determinism of the input program.

In both settings, the complexity of verification is managed by factoring out structure from the property to be proved. In the case of \(\text{printmsgs}()\), the structure is the linked list itself. The nodes summarized as a \(\text{list}\langle n\rangle\) are always organized as a proper linked list (there is no aliasing). The heap abstraction is updated only at locations corresponding to a single \textit{concrete} memory address, allowing us to avoid the precision lost in the presence of aliasing. As for \texttt{ManyPings}, representing all of the possible senders by a \textit{single} (albeit arbitrary) sender reduces an model with an unbounded number of a processes to a finite instance, allowing us to produce a simpler program by symbolic execution. We will see that symmetric non-determinism is the property that enables this reasoning.

1.3 Related Approaches to Automated Verification

While we have already discussed important work relating to distributed systems in, we now discuss how the ideas described in the previous section relate to, and borrow from, existing work which addresses message passing and model checking in general.

**Session Types.** There is tension between automation and strength of guarantee – this is by no means unique to the problem of distributed systems. However, the problem is certainly exacerbated by the complexities of the general execution model. While (mainstream) type systems are not expressive enough to prove full functional correctness, they do provide \textit{guarantees} that certain classes of errors are absent at run time. \textit{Session Types} [32, 33, 18, 10, 17] aim to bring these guarantees to developers of distributed systems. A session type is essentially a linearly typed resource denoting the protocol (or protocol fragment) executed by the corresponding typed term [78]. In general, the developer provides a \textit{global type}, which corresponds to a specification of the
protocol. The relevant pieces of the specification are then projected out as types for each participant’s program. If each process typechecks with respect to this projection, then the system as a whole is guaranteed to implement the protocol (and hence never get “stuck”). Session types have been implemented for mainstream programming languages such as Haskell [56], Go [54], and MPI [48].

**Reduction.** The ideas discussed in the previous section are based on Lipton’s theory of movers [45], which designates certain program actions as either *left* or *right* movers, depending on how they commute with actions performed by concurrently executing threads. Left and right movers can be composed to produce a *single* action, resulting in a program with fewer possible interleavings of actions. This idea was further developed in [24] to build a system in which the user iteratively applies mover reasoning to *rewrite* a program into a simpler variant with fewer atomic actions. In a given trace, a receive commutes to the left of every action except the matching send. Thus, we can not reduce a program (by replacing the matching send and receive pair with an assignment) unless we know that (a) the receive will eventually be enabled (b) the matching send is the *unique* matching send.

**Model Checking.** Model checkers, which explore the reachable states of a system either exhaustively or up to some bound, are effective at finding unanticipated corner cases. Because they explore every reachable state, exhaustive coverage (and hence verification) is only possible with finite, (and in practice, small) systems. Nevertheless, model checking techniques offer a good tradeoff to developers by improving confidence in the correctness of a system while offering high degrees of automation. The Mace language [37] features a model checker and has been used to implement well known distributed systems such as Chord [72] and Pastry [64]. In contrast, MoDIST [83] does not require modification to existing systems, and has been used to test systems such as Berkeley DB and MPS. Amazon recently published a report [53] on experiences applying TLA+ to some of its core services and uncovering design bugs using the TLC model checker.
Partial-Order Reduction. The verification problem amounts to checking if the set of program executions (or traces) is contained within the specification (a set of “good” traces). One way to simplify the task, then, is to reduce the set of program executions under consideration by arguing that certain traces are representative.

For example, model checkers a combinatorial explosion in the number of reachable states. As a result, typically only instantiations with small values are suitable for model checking, and as such do not provide a proof of correctness. Mitigating the so-called state-space explosion problem has been a topic of copious research. One of the most well-known family of techniques is partial-order reduction [29, 28, 3]. Partial-order reduction methods 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 [29]. A persistent set, for a given state, is a subset of all enabled transitions (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. These techniques do not directly translate to the parameterized (or infinite-state) setting, as persistent sets would be unbounded in size. Although some work has addressed the challenge of infinite state systems [12, 77], these techniques focus on systems with a bounded number of processes.

Synchronization. As synchronous systems have fewer behaviors than their asynchronous counterparts, they are, in principle, easier to verify. Verification results for a program assuming synchronous communication do not necessarily carry over to the asynchronous mode in the general case. However, substantial research has targeted this problem to determine which properties can be verified, and under what circumstances, by considering only synchronized communication. A condition for transforming asynchronous finite-state programs into synchronous programs is presented in [6]. However, this condition excludes even our ManyPings example. Similarly, the reduction proposed
by [19] explores only system configurations where the message buffers are small, by eagerly executing receives. However, this is not guaranteed to visit all global state configurations, and can thus not be used to prove the absence of deadlocks, for example.

This problem has also been considered in the more specific context of (finite-state models of) MPI programs [68, 69, 71, 70]. If each message receive point has only one possible sender, then it is possible to explore all of the halted states of the program by only considering \textit{synchronous} communication. Our approach incorporates this kind of reasoning. Crucially, symmetric non-determinism allows a process to receive from multiple sources.

\textbf{Parameterized Verification.} The model checking techniques discussed thus far are restricted to systems with a finite number of processes. In order to handle parameterized models, one generally constructs an \textit{abstract model} that approximates the behavior of the concrete system. Each of the (finite number of) abstract states denotes a subset of the concrete program’s state space. Counter abstractions [61] are a classic way to exploit symmetry by \textit{counting} the number of processes in a given state. Recent work focuses on automatically inferring counting arguments in the form of counting automata [27] or by synthesizing descriptions of sets and referring to their cardinalities [76]. For verifying distributed systems, however, we often require tracking the \textit{contents} of messages (e.g., when they contain process identifiers that are used as arguments in later sends), which is challenging for counter-based approaches, which is challenging for counter-based approaches.

The notion of unfolding is related the idea of data type reductions [50] (itself an instance of abstract interpretation [15]). Here, the data domain is collapsed into a distinguished representative and the “rest,” which summarize the remainder of the domain. Verification typically requires stating and proving lemmas to reclaim precision lost by creating these summaries.

There has been much work on inferring universally quantified invariants [7, 52, 66, 76]; despite recent progress, reliably synthesising both quantified invariant and
The thesis of this dissertation is that the verification of distributed systems is simplified by automatically eliminating the non-determinism of the messaging system. To that end, the dissertation makes the following contributions.

1. We formalize a property of message passing programs called Symmetric Non-determinism, which enables local reasoning about actions;

2. We then describe Canonical Sequentialization, an algorithm for transforming a message passing program into its canonical sequentialization, which is a sequential program that preserves the final states of its input program. We evaluate canonical sequentialization by implementing it as a tool, BRISK, which compiles a HASKELL program to a IS program and then computes its canonical sequentialization.

3. Finally, we present Pretend Synchrony, which generalizes the idea of canonical sequentialization to transform a message passing program into a synchronized program with coarse-grained concurrency. Pretend synchrony allows the user to treat an asynchronous program as if it were synchronous, which simplifies the annotations required to prove a program correct. We evaluate the approach on several case studies, including the classic two-phase commit protocol, Raft leader election, and single-decree Paxos.

We will see that canonical sequentializations can be computed fast enough to be integrated tightly into the design-implement-check cycle for modern, high-level programming languages. Moreover, by again exploiting symmetric non-determinism, we will see how to drastically simplify proofs of correctness by first automatically (and efficiently) computing the synchronization of a system, should a sequentialization not exist. We first describe a language, IS, in Chapter 2 for formalizing message passing programs. In Chapter 3 we describe Canonical Sequentialization and in Chapter 4 we
discuss *Pretend Synchrony*. In Chapter 5 we propose ideas for future research based on the sequentialization and synchronization techniques discussed in the preceding chapters.
Chapter 2

Message Passing Programs

2.1 Overview

In the following chapters we will be discussing approaches to verifying properties of distributed systems. Programmers have at their disposal a rich set of languages and tools for constructing these systems. While they make programming convenient, features like (higher-order) procedures and dynamic process creation complicate analysis by obscuring both control flow and process structure.

To make progress, in this chapter we formalize $I_s$, a core Imperative language for Symmetric reasoning. $I_s$ programs are procedure-less (but have unbounded iteration) and feature explicit parallel composition rather than dynamic process creation. Crucially, $I_s$ explicitly supports symmetric by allowing programs to be parameterized by scalar sets of values (such as process identifiers) [34], and including forms for iterating parallel and sequential composition. We shall see in Chapter 3 how this simplifies reasoning about message passing programs.

In this chapter, we present the syntax and semantics of $I_s$, and then prove an essential theorem about reducing the complexity of exploring the halted states of $I_s$ programs. In this chapter and those that follow we make heavy (ab)use of mathematical notation. $A[e/x]$ substitutes the variable $x$ with the expression $e$. We let $\emptyset$ denote an empty map (in addition to the empty set) and write $m[x \leftarrow y]$ to denote the map $m'$ that is equivalent to $m$ at every point except $x$ which is mapped to $y$. 
### Syntax and Semantics

#### Syntax

**Expressions.** $I_s$ is designed to be parametric in the language of purely local computations. We assume that the local process state at least includes integers and process identifiers (PIDs). Primitive expressions in $I_s$ are constants $c$ and variables $i.x$. In $I_s$, each process is associated with a unique process identifier (PID), which serves as an address for sending messages. Variables are qualified by the PID of their owning process, $i.e., i.x$ refers to $i$'s variable $x$. When this is clear from context or unimport, we often drop the prefix and just write $x$. We assume disjoint sets of identifiers to name program variables, parameters, and types. The language of expressions includes $f(\bar{e})$, which denotes a purely local computation – that is, a computation that does not send or receive any messages.
Figure 2.2. Definition of $A : ✓$.

Figure 2.3. Definition of $\text{Procs}(A)$

Programs. We use skip to denote the empty process, which does nothing. The process $[s]_p$ is a singleton that denotes the process with PID $p$ executing statement $s$. Processes are further constructed by sequential ($P; Q$) and parallel composition ($P | Q$), and when a single process executes an entire program, we abuse notation and abbreviate, e.g., $[s_1]_p;[s_2]_p$ as $[s_1; s_2]_p$ or for $(x : X) \{ [s_1]_p;[s_2]_p \}$ as $\{ (x : X) \{ s \} \}_p$. We write for $(i : I) \{ A \}$ and $\prod_{i : I} A$ to denote the sequential and parallel instantiations of $A$ to the values in $I$, respectively of $A$ to the values in $I$, respectively.

Normal Form. A program is in normal form, written $A : ✓$, if it consists of parallel compositions of sequences of statements from distinct processes. The definition
of $A : ✓$ is found in Figure 2.2.

**Example 2.1.** Consider $A$ and $B$ defined below.

$$A \triangleq [s_1]_p \parallel \prod_{q \in Q} [s_2]_q$$

$$B \triangleq [s_3]_p \parallel \prod_{q \in Q} [s_4]_q \parallel [s_5]_m$$

$A$ and $B$ are both in normal form, but $A ; B$ and $A \parallel B$ are not.

**Composition.** We define $A \triangleq B$ to be the process-wise composition of $A$ and $B$.

$$A \triangleq B \triangleq \begin{cases} \prod_{x:X} C; D & A = \prod_{x:X} C, B = \prod_{x:X} D, \text{ and } \text{Procs}(C) = \text{Procs}(D) = \{x\} \\ A ; B & \text{Procs}(A) \cap \text{Procs}(B) \neq \emptyset \\ A \parallel B & \text{otherwise} \end{cases}$$

**Figure 2.4.** Definition of $A \triangleq B$.

Intuitively, $A \triangleq B$ sequences the statements in $B$ after the corresponding statements in $A$, assuming $A : ✓$ and $B : ✓$. The definition is found in Figure 2.4.

**Example 2.2.** Recall $A$ and $B$ defined in Example 2.1. Their composition is given by

$$A \triangleq B = [s_1 ; s_3]_p \parallel \prod_{q \in Q} [s_2 ; s_4]_q \parallel [s_5]_m$$

### 2.2.2 Semantics

Next we discuss the semantics of $I_s$. The operational semantics are shown in Figure 2.7. First we discuss the semantics at a high level, and then formally define the properties that will be of interest in this dissertation.

**Message Channels and Ordering.** In $I_s$, processes communicate by sending and receiving messages over *typed* channels: There is a separate channel for each (ordered) pair of processes, and each channel is split into sub-channels for different message types. Messages on the same sub-channel are delivered in order, whereas there are no
guarantees for messages sent on separate sub-channels. Message delivery can be delayed arbitrarily, but messages are never dropped or duplicated. These semantics are modeled after the guarantees provided by high-level languages such as ERLANG.

Sends and Receives. The process \([send(t,q,e)\ell]_p\) asynchronously enqueues message \(e\) in the sub-channel \((p,q,t)\). The tag \(\ell\) is a unique identifier for the syntactic send statement. We denote a set of tags as \(\ell\). Sets of tags are used to annotate receive statements, but have no effect on the semantics. Dually, if the sub-channel \((p,q,t)\) is not empty, then \([q.x := recv(t,p)\tau]_q\) dequeues the first message on the channel and binds it to \(x\); otherwise, the process blocks. \([q.x := recv(t,I)\tau]_q\) blocks until there is some non-empty sub-channel \((i,q,t)\) and then behaves as \([q.x := recv(t,i)\tau]_q\). The form \([q.x := recv(t,I)\tau]_q\) behaves similarly except the sub-channel is constrained to be an element of \(I\).

Example 2.3. Consider the processes \(A\) and \(B\) defined below:

\[
A \triangleq [send(t,r,v_0); send(t,r,v_1)]_p \parallel [r.x := recv(t,*) ; r.y := recv(t,*)]_r
\]

\[
B \triangleq [send(t,r,v_0)]_p \parallel [send(t,r,v_1)]_q \parallel [r.x := recv(t,*) ; r.y := recv(t,*)]_r
\]

In \(A\), both sends are executed by the process \(p\), and hence the messages are sent on the same sub-channel. They are therefore delivered to \(r\) in order, and on termination we always have \(r.x = v_0\) and \(r.y = v_1\). On the other hand, in \(B\), the sends are executed by different processes and hence the messages may be delivered in any order. On termination, either \(r.x = v_0\) and \(r.y = v_1\) or \(r.x = v_1\) and \(r.y = v_0\).

Operational Semantics. Definitions required for the semantics of \(I_S\) are shown in Figure 2.6, and the semantics itself is in Figure 2.7. A configuration \(C\) is either a triple of a program store, buffer, and program \((\sigma, \mu, P)\) or the distinguished crash denoting an error state. The semantics is defined by the relation \(C \xrightarrow{T} C'\) where \(T\) is a trace (which we also call a transition). The relation \(A \equiv B\), shown in Figure 2.5 is used to perform trivial reductions: if \(A \xrightarrow{T} B\) and \(B \equiv B'\), then \(A \xrightarrow{T} B'\) (and similarly if \(B \xrightarrow{T} A\)). Finally, we
will be discussing reachable configurations and enabled transitions in the remainder of the dissertation.

**Definition 2.1 (Enabled).** A transition \( T \) is enabled in configuration \( C \) if and only if there exists \( C' \) such that \( C \xrightarrow{\_} T \_ C' \).

**Definition 2.2 (Reach Set).** We define \( \text{Reach}(C) \) to be the reachable configurations of \( C \):

\[
\text{Reach}(C) \triangleq \left\{ C' \mid \exists T. C \xrightarrow{T} C' \right\}
\]

**Parameterized Programs.** Programs are parameterized by disjoint sets of PIDs, denoting finite but unbounded collections of process identifiers. To discuss properties of these programs, we first discuss valid instantiations of parameterized programs. An instantiation of a program \( A \) is a program store \( \sigma \) that gives a valuation to all of the PID and PID set-valued variables in \( A \): we call these initial stores.

**Definition 2.3 (Initial Stores).** The store \( \sigma_0 \) is an initial store of \( A \) if

1. For all \( X \in \text{PidSetParams}(A) \), \( |\sigma(X)| > 0 \);
2. For all distinct \(x, y \in \text{PidParams}(A)\), \(\sigma(x) \neq \sigma(y)\); and

3. For all distinct \(X, Y \in \text{PidSetParams}(A)\), \(\sigma(X) \cap \sigma(Y) = \emptyset\).

We lift the definition of initial stores to define the initial configurations of \(A\):

\[
\text{Initial}(A) \triangleq \{(\sigma, \emptyset, A) \mid \sigma \text{ is an initial store of } A\}
\]

**Properties.** A property \(\varphi\) is a formula over a program store: a store \(\sigma\) satisfies \(\varphi\) (we write \(\sigma \models \varphi\)) if and only if \(\varphi\) is true when its free variables are given their valuation from \(\sigma\). Satisfaction is lifted to configurations in the obvious way.

**Halting properties.** In this dissertation we are concerned with verifying halting properties of programs, that is, properties over halted states. Intuitively, the halting property of \(A\) is the set of halted configurations reachable from the initial configurations of \(A\), where a configuration is halted if its reachable set is empty.

**Definition 2.4 (Halted Configurations).** We define the halted configurations of \(A\), written \(\text{Halted}(A)\), as

\[
\text{Halted}(A) \triangleq \{C \mid C \in \text{Initial}(A) \text{ and } \text{Reach}(C) = \emptyset\}
\]

Examples of halting properties range from local assertion safety to global properties such as deadlock freedom or agreement as in the two-phase commit protocol. We write \(P \models_H \varphi\) to denote that \(\varphi\) holds in all halted configurations reachable from the initial configuration.

**Definition 2.5 (Halting Properties).** We say that \(\varphi\) is a halting property of \(A\), written \(A \models_H \varphi\) if

\[
\forall C \in \text{Halted}(A). \ C \models \varphi
\]

### 2.3 Tagged Programs

As described in Section 2.2.1, sends and receives are annotated with tags and sets of tags, respectively. The intention is that a receive’s tag set corresponds to the set of
syntactic send statements whose messages may be received.

**Definition 2.6.** A program $A$ is **tagged consistently** if for all initial stores $\sigma_0$, whenever

$$(\sigma_0, \emptyset, A) \xrightarrow{T_{\ell}} (r, \mu, B) \xrightarrow{[\text{recv}(t, p)_{\ell}]} (\sigma', \mu', C)$$

then, letting $\mu(p, q, t) = v^f \cdot v'$, $\ell \in \ell$.

We will see in Chapter 3 that being able to determine the possible senders for a given receive is essential for reducing the complexity of message passing programs.

### 2.4 Reductions for $I_S$

Ultimately, reasoning about the behavior of concurrent systems is complicated because the behavior program, that is, its set of program executions or traces, by definition includes all possible interleavings of actions by each of constituent threads or processes. However, in practice, we are often interested in only a subset of the program’s behaviors, e.g., a property of the program’s halting executions. Recalling Example 2.3, the non-determinism of the message delivery system is observed as different final values for the variables $x$ and $y$, depending on whether $p$ or $q$’s message is delivered first. On the other hand, consider $C$ below, modified from $B$ by a deft change of sub-channels:

$$C \triangleq [\text{send}(t_0, r, v_0)]_p \parallel [\text{send}(t_1, r, v_1)]_q \parallel [r.x = \text{recv}(t_0, *); r.y = \text{recv}(t_1, *)]_r$$

The traces of $B'$ exhibit multiple interleavings of the statements in $p, q, r$: for example, $r$’s receive may execute as soon as $p$ has sent its message or $r$ may wait until both messages have been sent.

On the other hand, $B'$ has only a single **halted** configuration, in which $x_r = v_0$ and $y_r = v_1$. This is because the first receive in $r$ commutes with $q$’s send: given an initial state, executing both statements in either order results in equivalent states. Considering only halted configurations, then, we would not be able to distinguish $C$ from $D$ (which is not
in normal form):

\[
D \doteq \left[ \text{send}(t_0, r, v_0) \right]_p ; \left[ \text{send}(t_1, r, v_1) \right]_q ; [r.x := \text{recv}(t_0, *)]_r ; [r.y := \text{recv}(t_1, * )]_r
\]

or indeed even from \( E \):

\[
E \doteq \left[ \text{send}(t_0, r, v_0) \right]_p ; [r.x := \text{recv}(t_0, *)]_r ; \left[ \text{send}(t_1, r, v_1) \right]_q ; [r.y := \text{recv}(t_1, * )]_r.
\]

If we focus our attention on halted configurations, then it is impossible to distinguish two executions that start in some configuration \( C \) and terminate in the configuration \( C' \). For example if \( C \xrightarrow{t_0; t_1} t_1 \) and \( C \xrightarrow{t_1; t_0} t_0 \), then \( t_0 \) and \( t_1 \) commute with respect to each other. We can exploit the fact that some transitions commute in order to build \textit{simpler} programs (\ie, have \textit{fewer} traces) that have the same \textit{observational} behavior. We reduce complexity by eliminating program interleavings, \ie, by replacing parallel composition with sequential composition. Intuitively, if two statements \( s_1 \) and \( s_2 \) in different processes commute, then we can consider only those executions in which, say, \( s_1 \) occurs before \( s_2 \).

We formalize this intuition by defining \textit{left movers}, after \cite{24, 45}.

We define a notion of \textit{left movers}. Intuitively, a left mover \textit{in a configuration} is an enabled transition \( T_\lambda \) that, given a sequence \( T \) of transitions, (1) remains enabled after \( T \), (2) commutes to the left of \( T \), and (3) if \( T \) leads to a crash, then running \( T_\lambda \) first still leads to a crash (note that \( T \) should therefore not contain \( T_\lambda \)).

Our definition closely resembles that of \textit{ample} or \textit{persistent sets} \cite{29, 59}, however we additionally require that \( T_\lambda \) can be preprended to a trace that results in a crash.

**Definition 2.7** (Left Movers). Let \( C \) be a configuration and \([s]_p \) be a transition enabled in \( C \). Then \([s]_p \) is a \textit{left mover} in \( C \) if:

1. For all \( C' \neq \text{crash} \), \( [s]_p \) if \( C \xrightarrow{T; [s]_p}_I t_1 C' \) then \( C \xrightarrow{[s]_p; T}_I t_1 C' \); and

2. For all \( T \) not containing \([s]_p \), if \( C \xrightarrow{T}_I t_1 C' \) and \( C' \) is halted, then either \([s]_p \) is enabled in \( C' \) or \( C' = \text{crash} \).
3. If $C \xrightarrow{T_{1_s}} \text{crash}$ and $T$ does not contain $[s]_p$, then $C \xrightarrow{[s]_pT_{1_s}} \text{crash}$.

Theorem 2.1 says that halted states are preserved by executing left movers in a particular configuration. This theorem is a straightforward consequence of our definition (indeed, this result is the reason for our particular definition).

**Theorem 2.1 (Left Mover Reduction).** Assume $[s]_p$ is a left mover in configuration $C$. Then

$$\text{Halted}(C) = \bigcup \left\{ \text{Halted}(C_s) \mid C \xrightarrow{[s]_p} C_s \right\}$$

**Proof.** The theorem is a consequence of Definition 2.7. Suppose $C \xrightarrow{T_{1_s}} C'$ where $C'$ is halted. Further suppose that $[s]_p$ is enabled in $C$. There are two cases to consider, depending on if $T$ contains $[s]_p$.

1. Case $T = T_0; [s]_p; T_1$ for some $T_0$ and $T_1$:

   Then, by Definition 2.7, $C \xrightarrow{[s]_pT_0; T_1} C'$, and hence $C \xrightarrow{T_0; T_1} C''$ and $C \xrightarrow{[s]_pT} C'$ by inversion. Thus, $C' \in \text{Halted}(C'')$.

2. Case $T \neq T_0; [s]_p; T_1$ for all $T_0$ and $T_1$:

   Then, $C' = \text{crash}$ since otherwise $[s]_p$ would be enabled and hence $C'$ wouldn't be halted. Then, by Definition 2.7, $C \xrightarrow{[s]_pT} C'$ and hence $C \xrightarrow{[s]_pT} C'' \xrightarrow{T_{1_s}} C'$ Thus, $C' \in \text{Halted}(C'')$.

As a consequence of Theorem 2.1, it is easy to see that sends and enabled receives, for whom there is only one possible sender, are left movers. To explore reachable halted states, it suffices to execute these statements first, without backtracking to explore the states reachable from co-enabled statements.

**Fact 2.1.** Given $C = (\sigma, \mu, A \parallel [\text{send}(t, q, v)]_p; s)$, if $A \parallel [\text{send}(t, q, m)]_p; s : \checkmark$ then

$$\text{Halted}(C) = \text{Halted}((\sigma, \mu[(p, q, t) \leftarrow \mu(p, q, t) \cdot v], A \parallel s))$$
Fact 2.2. Given $C = (\sigma, \mu, A \parallel [p.x := \text{recv}(t, q)]_p; s)$, if $A \parallel [p.x := \text{recv}(t, q)]_p; s : \checkmark$ and $\mu(q, p, t) = \nu \cdot \nu'$, then

$$\text{Halted}(C) = \text{Halted}((\sigma[(p, x) \leftarrow \nu], \mu[(q, p, t) \leftarrow \nu'], A \parallel s))$$

2.5 Summary

In this chapter we have described $I_S$, a first order language which will serve as the intermediate verification language in the remainder of the dissertation. Importantly, we have given a $I_S$ a semantics that allows us to connect in-flight messages to their syntactic point of origin, which is crucial to the approaches described later. Finally, we have defined our own notion of left movers. Because left movers commute to the left in any trace in which they appear, to explore the halted states reachable from a given configuration, one need only explore the halted states reachable after executing the left movers enabled in the current configuration. This observation will be exploited in the two chapters that follow.

Acknowledgments

Chapter 2, in part, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. The dissertation author was the primary investigator and author of this paper.
Figure 2.7. $I_5$ Semantics: $C \xrightarrow{T} I_5 C$
Chapter 3

Canonical Sequentializations

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. The distributed systems developer on the other hand is bereft of development- and compile-time tools that guarantee the absence of concurrency errors.

In this chapter, we describe 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.

Consider an implementation of MapReduce [16] 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 Figure 3.12).
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.

In this chapter we describe 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 (Section 3.2). We formalize canonical sequentialization as a set of *local rewriting rules* (Section 3.3) on I_5 programs. 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. 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 a method to *automatically synthesize* a canonical sequentialization from a symmetric non-deterministic program (Section 3.4). We use this synthesis algorithm to implement a distributed systems verification tool called *BRISK*. BRISK first compiles *HASKELL* programs that use the CLOUD HASKELL library [25] into our core language (Section 3.5). 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 [33, 21], a concurrent programming textbook [47], well known protocols such as two-phase-commit [43], MapReduce [16] and implementations of a key-value store, and a distributed file-system (Section 3.6). We show that unlike model checking, which gets prohibitively slow — i.e., times out at one minute even with just 10 processes — 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.

3.1 Overview

Before discussing the technical details behind canonical sequentializations, we first give an overview of how BRISK lets us write and verify a concurrent task distribution service in HASKELL by synthesizing its canonical sequentialization (Section 3.1.1). Next, we explain the main ideas underlying canonical sequentialization with a series of small examples (Section 3.1.2) designed to give an intuition for the reasoning that BRISK performs. Finally, we make some remarks on its expressiveness (Section 3.1.3).

3.1.1 A Task Distribution Service

Figure 3.1 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 \( \text{foldM} \) to repeat the function \( \text{serverLoop} \) 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 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 [38].

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 [27, 76, 7] and thus, we are not aware of any automated verification method that can handle this simple example.

**Verification via Canonical Sequentialization.** In this chapter, 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, and iterating over sets of processes. When BRISK is invoked to verify the program, it compiles (Section 3.5) the higher-order HASKELL source into the first-order Iₚ in order to make explicit the control flow and process structure of the input
program. BRISK then synthesizes the program’s canonical sequentialization (Section 3.3) which we show in Figure 3.2. 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 Section 3.3.4), we can conclude that the original program is correct.
import Brisk

data Msg = Request ProcessId | Ack

data Work = Task ProcessId Item | None

main n = do
  self ← getSelfPid
  cs ← spawnMany n (client self)
  m ← spawn (master cs)
  server m cs

client sv = do
  self ← getSelfPid

  -- request a work item from server
  send sv (Request self)

  -- block until an item is assigned
  msg ← receive

  case msg of
    Task m task → do
      process task
      send m Ack
    None → fail

server m cs = foldM serverLoop () cs
  where
    serverLoop _ _ = do
      -- wait for a request
      Request p ← receive
      -- compute next item
      item ← nextItem
      -- send item to p
      send p (Task m item)
      return ()

master cs = foldM masterLoop () cs
  where
    masterLoop _ _ = do
      -- wait for Ack
      Ack ← receive
      return ()

Figure 3.1. A task distribution service.
for (c: cs) {
    [Request p := Request c]server;
    [item := nextItem]server;
    [msg := Task master item]c;
    case msg of
    Task m task → process task
    None → fail
    }; c
for (c: cs) {
    [Ack := Ack]master
}
3.1.2 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 [45]. 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.

Canonical Sequentialization. EX1 below is written in our core language $I_\Sigma$, in which two processes $p$ and $q$ exchange messages. In the interest of clarity, in this section we omit the type arguments to send and receive statements when there is no room for confusion.

$$\begin{align*}
\text{EX1} & \equiv \begin{array}{l}
\text{send}(q, \text{ping})$
\end{array} \\
\text{p} & \parallel \\
\text{w} & := \text{recv}(\ast)
\end{align*}$$

$$\begin{align*}
\text{send}(p, \text{pong})$
\end{align*}$$

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 $p$. Since the sends and receives can only be executed in a single order, we can rewrite the above concurrent program into its canonical sequentialization:

$$[v := \text{ping}]_q ; [w := \text{pong}]_p$$

Crucially, both programs are equivalent in the sense that they terminate in the same state.
**Parametric Programs.** Next, consider program EX2, which contains a (statically) unbounded number of processes:

\[
\text{EX2} ≥ \left[ \begin{array}{c}
\text{for } (q : Q) \{ \\
\quad \text{send}(q, \text{ping}); \\
\quad w := \text{recv}(q) \\
\} \\
\end{array} \right]_p \bigg\| \prod_{q \in Q} \left[ \begin{array}{c}
\quad v := \text{recv}(); \\
\quad \text{send}(p, \text{pong})_q \\
\end{array} \right]
\]

In EX2 a single process \( p \) exchanges messages with a set of 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 \( p \). 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:

\[
\text{for } (q : Q) \{ \\
\quad [v := \text{ping}]_q; \\
\quad [w := \text{pong}]_p \\
\}
\]

**Multiple Orders.** Next, consider 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.

\[
\text{EX3} ≥ \left[ \begin{array}{c}
\text{for } (q : Q) \{ \text{send}(q, \text{ping})_a \} \\
\end{array} \right] \bigg\| \prod_{q \in Q} \left[ \begin{array}{c}
\quad v := \text{recv}(*)_a; \\
\quad \text{send}(p, \text{pong})_q \\
\end{array} \right]_b
\]

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 \textit{pong} messages in any order due to the speed of the underlying network or differences in execution times between members of \( Q \).

\textbf{Checking Symmetric Non-Determinism.} In order to rewrite \textsc{ex3} into its canonical sequentialization, we first need to check that it satisfies symmetric non-determinism. For this, we annotate each syntactic occurrence of \textit{send} with a unique \textit{tag} as described in Section 2.2.1. For each \textit{receive}, we then compute an \textit{over-approximation} of the set of \textit{tags} it can receive from. \textsc{brisk} computes these \textit{tags} using a lightweight syntax-guided method (Section 3.2). \textsc{ex3} shows \textit{send}-tags in \textcolor{red}{red} and \textit{receive}-tags in \textcolor{blue}{blue}. In order to satisfy symmetric non-determinism, we require that each receive-set contains either \( i \) only \textit{tags} from a \textit{single process}, or \( ii \) a \textit{single tag} from a process in set of \textit{symmetric processes}. The \textit{tags} for \textsc{ex3} satisfy this requirement.

\textbf{Moving Left: Eagerly Executing Receives.} We split the rewrite of \textsc{ex3} into two steps. For the first step, consider a \textit{send} to some process \( q \) in \( p \)'s \textit{for}-loop together with its matching \textit{receive} in \( q \). Since \textsc{ex3} satisfies symmetric non-determinism, we know that there are no \textit{additional} \textit{sends} \( q \) could receive from. Moreover, the \textit{receive} is \textit{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 \textit{directly after} its matching \textit{send} in \( p \) without changing the behaviour of the program. We say that the \textit{receive} can be \textit{moved left}, up to its matching \textit{send}. Applying the above reasoning, we can rewrite \textsc{ex3} into the partially sequentialized program:

\[
\text{for } (q : Q) \{ [v := \text{ping}]_q \};
\]

\[
(\text{for } (q : Q) \{ [w := \text{recv}(\ast)]_p \} \parallel \bigcap_{q \in Q} [\text{send}(p, \text{pong})]_q)
\]

The program consists of a \textit{sequential prefix} which is a rewriting of the boxed statements and a \textit{remainder term} that corresponds to the part of the program that still needs to be rewritten. For the second rewriting step, consider the boxed statements in the remainder term. In a given loop iteration, \( p \) can receive a \textit{send} from \textit{any} of the remaining processes in \( Q \). However, each process runs the same program (with the notable exception that the
processes 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 to its canonical sequentialization:

\[
\text{for } (q : Q) \{ [v := \text{ping}]_q ; \text{for } (q : Q) \{ [w := \text{pong}]_p \} \}
\]

**Example 4: Multi-Party Communication.** Finally, we turn to the example \textsc{ex4}, whose communication structure matches the task distribution service from Figure 3.1:

\[
\text{EX4} \equiv \begin{cases}
\text{for } (q : Q) \{ \\
\quad \begin{cases}
\quad \text{id} := \text{recv}(\ast)_b ; \\
\quad \text{send(id, ping)}_a \\
\quad \end{cases} \\
\quad \text{send(p, pong)}_b \\
\quad \text{send(m, pong)}_c \\
\quad \text{send(m, pong)}_c \\
\quad \text{send(m, pong)}_c \\
\quad \text{send(m, pong)}_c \\
\end{cases}
\end{cases}
\]

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 \textsc{ex4} is symmetrically non-deterministic: the annotated tags show that this requirement is met. Next, we rewrite \textsc{ex4} into its canonical sequentialization. The rewrite is split into two steps: a first step in which \textsc{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, \textsc{brisk} rewrites the boxed statements. This rewrite step produces a sequential prefix:

\[
\text{for } (q : Q) \{ [id := q]_p ; \\
\quad [v := \text{ping}]_q ; \\
\}
\]

(3.1)
and an additional residual term contains the messages the processes in $Q$ sent to $m$:

$$\bigcap_{q \in Q} \left[ \text{send}(m, \text{pong}) \right]_q$$  \hspace{1cm} (3.3)

In the second step, BRISK rewrites the parallel composition of the remainder program with the residual term, shown below.

\[
\begin{align*}
\text{for} \ (q : Q) \ {\{} \\
\quad [id := q]_p ; \\
\quad [v := \text{ping}]_q \\
\} \\
\end{align*}
\]

$$\bigcap_{q \in Q} \left[ \text{send}(m, \text{pong}) \right]_q \parallel \left[ \begin{array}{c}
\text{for} \ (q : Q) \ {\{} \\
\quad w := \text{recv}(*) \\
\} \\
\end{array} \right]_m$$

BRISK rewrites the boxed statements into the canonical sequentialization:

\[
\begin{align*}
\text{for} \ (q : Q) \ {\{} \\
\quad [id := q]_p ; \\
\quad [v := \text{ping}]_q \\
\} \\
\end{align*}
\]

$$\text{for} \ (q : Q) \ {\{} \\
\quad [w := \text{pong}]_m \\
\}$$  \hspace{1cm} (3.5)

### 3.1.3 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 such as rings.

Example 3.1. Consider program C below, which is a set of clients of a key-value store S whose API supports retrieving the value of a key \( k \) with a Get\((k)\) message, and setting the value \( v \) of a key \( k \) with a Set\((k,v)\) message. Each \( c \in Cs \) performs a non-deterministic choice (represented by the condition \( * \)) to either send a Get\((k)\) or a Set\((k,v)\) message to \( s \).

\[
C \triangleq \bigcap_{c \in Cs} \begin{cases} 
\text{if } * \{ \\
\quad \text{send}(s, \text{Get}(k)) \\
\} \\
\text{else} \{ \\
\quad \text{send}(s, \text{Set}(k,v)) \\
\} 
\end{cases}
\]

\[
C' \triangleq \bigcap_{c \in Cs} \begin{cases} 
\text{if } * \{ \\
\quad \text{msg} \coloneqq \text{Get}(k) \\
\} \\
\text{else} \{ \\
\quad \text{msg} \coloneqq \text{Set}(k,v) \\
\quad \text{send}(s, \text{msg}) \\
\} 
\end{cases}
\]

\[
S \triangleq \begin{cases} 
\text{while true} \{ \\
\quad \text{op} \coloneqq \text{recv}(*) \\
\quad \text{if isGet}(\text{op}) \{ \\
\quad\quad \text{doGetOp()} \\
\quad\quad \text{else} \{ \\
\quad\quad\quad \text{doSetOp()} \\
\quad\quad \} \\
\quad \} \\
\end{cases}
\]

The composition \( C \parallel S \) 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, $C' \parallel S$ has symmetric non-determinism.

**Example 3.2.** 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.

**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).

3.2 Symmetric Non-determinism

In this section, we make precise the definition of symmetric non-determinism as introduced in Section 3.1. Let $\text{Recvs}(A)$ denote the set all of the syntactic receive statements occurring in $A$, and let $\text{SendTags}(A)$ denote all of the tags associated with the syntactic send statements occurring in $A$.

**Definition 3.1 (Symmetric Non-determinism).** Let $A$ be a program that is tagged consistently (Definition 2.6). Then $A$ has symmetric non-determinism if, for all $\text{recv}(t, w) \in \text{Recvs}(A)$, $A \equiv A_s \parallel A_r$ where $\bar{t} \in \text{SendTags}(A_s)$, $\bar{t} \cap \text{SendTags}(A_r) = \emptyset$ and either

1. $\text{Procs}(A_s) = \{p\}$ where $p$ is a PID variable ($A_s$ is a concrete process); or
2. $|\bar{t}| = 1$ and $\text{Procs}(A_s) = \{P\}$ where $P$ is a PID set variable ($A_s$ is a symmetric group of process).

**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. This approximation suffices for the examples considered in this dissertation. The method described in the following sections is orthogonal, and could be extended to a more precise analysis if deemed necessary.

**Receive 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.

3.3 Rewrite Rules

Next, 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 ($\Gamma$); a sequential prefix ($\Delta$); the program ($P$) to be rewritten; and a residual process ($\Psi$) comprising statements that interact with processes outside of $P$. A context $\Gamma$ 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 about process identifiers. We summarize the syntax in Figure 3.3. We sometimes use $\Gamma$ to refer to one of its components, for example, we write $\Gamma(p,q,t)$ to mean $BUFF(p,q,t)$. Finally, for a context $\Gamma \in (BUFF, ASRT)$ and an assertion $a$, we write $\Gamma \vdash a$ to mean $a \in ASRT$, and $\Gamma \not\vdash a$ to mean $a \notin ASRT$.

Rewriting Rules. Each rewriting rule defines a judgment of the form

$$\Gamma, \Delta, P, \Psi \leadsto \Gamma', \Delta', P', \Psi'.$$

The goal of each step is to move parts of program $P$ into the sequential prefix $\Delta$ 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.

3.3.1 Basic Rules

Figure 3.4 contains the basic rules. We assume that programs do not contain wildcard receives, having removed them according to the method described in Section 3.2.

Sends and receives. Rule R-SEND treats sends. The rule rewrites a send from process \( p \) to some \( x \) into \( \text{skip} \), 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 \( \Delta = x = q \), where we write \( \Delta \models \varphi \) to mean that formula \( \varphi \) is valid after executing \( \Delta \) (see Section 3.4 for a discussion of \( \Delta \models \varphi \)). The rule updates context \( \Gamma \) 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 \( \Gamma \) on all inputs except for \( a \) where it returns \( b \). Rule R-RECV rewrites a receive from some \( x \) into \( \text{skip} \), 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.

Example 3.3. Consider again program \( \text{ex1} \) from Section 3.1 where we replaced wildcard receives with receives from the respective processes:

\[
\begin{align*}
\left[ \text{send}(q, \text{ping}); \right]_{p} & \parallel \left[ v = \text{recv}(p); \right]_{q} \\
P & = \left[ w = \text{recv}(q) \right]_{p} \left[ \text{send}(p, \text{pong}) \right]_{q}
\end{align*}
\]

We start the rewrite with the symbolic state given by \( \Delta \models \text{skip} \), \( \Psi \models \text{skip} \) and \( \Gamma \models (\text{BUFF}\emptyset, \emptyset) \), where \( \text{BUFF}\emptyset \) maps every channel to the empty sequence \( \epsilon \). 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 below, where we update the buffer to \( B_{\phi}[(p, q, t) \leftarrow \text{ping}] \):

\[
\begin{align*}
\text{skip}; & \quad [w := \text{recv}(q)]_p \quad \begin{cases} v := \text{recv}(p); \quad \text{send}(p, \text{pong})_q \end{cases} \\
\end{align*}
\]

Applying R-CONTEXT and R-RECV yields the following, with buffer \( B_{\phi} \) and prefix \( \Delta = [v := \text{ping}]_q \).

\[
\begin{align*}
\text{skip}; & \quad \text{skip;} \\
[w := \text{recv}(q)]_p & \quad [\text{send}(p, \text{pong})]_p
\end{align*}
\]

Applying R-CONTEXT and R-CONGR twice yields \([w := \text{recv}(q)]_p \parallel [\text{send}(p, \text{pong})]_p\).

Applying the same rules again yields \text{skip} with prefix \( \Delta = [v := \text{ping}]_q ; [w := \text{pong}]_p \).

### 3.3.2 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 3.5 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 \( \emptyset \subset Q^* \subset Q \) (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 \text{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. Likewise, the rule requires the context after the rewrite to
\begin{align*}
\text{R-CONTEXT} & \\
\Gamma, \Delta, A, \Psi \sim \Gamma', \Delta', A', \Psi' & \\
\Gamma, \Delta, A \circ B, \Psi \sim \Gamma', \Delta', A' \circ B, \Psi' & \text{R-CONGRUENCE} \\
\Gamma, \Delta, A, \Psi & \sim \Gamma, \Delta, B, \Psi \\
\text{R-SEND} & \\
\Delta & \vdash x = q \\
q & \text{is a PID} & \Gamma(p, q, t) = m & p \text{ is a PID} \\
\Delta & \not= E(q) & \Gamma' = \Gamma[(p, q, t) \leftarrow m \cdot n] & \Gamma'[p, q, t] = m \cdot n \\
\Gamma, \Delta, [send(t, x, n)]_p, \Psi & \sim \Gamma', \Delta, \text{skip}, \Psi & \Gamma, \Delta, [y = recv(t, x)]_q, \Psi & \sim \Gamma', \Delta', \text{skip}, \Psi \\
\text{R-RECV} & \\
\Delta & \vdash x = p & \Gamma(p, q, t) & \Gamma[(p, q, t) \leftarrow n] \\
\Delta & \not= E(x) & \Gamma'[p, q, t] = m \cdot n & \Delta' = \Delta; [y := m]_q \\
\Gamma, \Delta, [y := \text{recv}(t, x)]_q, \Psi & \sim \Gamma', \Delta', \text{skip}, \Psi & \Gamma, \Delta, [y := \text{recv}(t, x)]_q, \Psi & \sim \Gamma', \Delta', \text{skip}, \Psi \\
\end{align*}

Figure 3.4. Rewrite Rules (Basic Statements)

\begin{align*}
\text{R-LOOP-UPD} & \\
q & \text{fresh} & \Delta' & \Delta; \text{for } (q : Q) \{ \Delta''[q/u] \} & \\
& & \Psi' & \Psi \parallel \prod_{q \in Q} [\Psi'']_q \\
\Delta_0 & \vdash \text{havoc}(\Delta, A, B) & \Gamma_0, \Delta_0, [A[q^*/q]]_p \parallel \prod_{q \in Q^*} [B]_q, \text{skip} & \sim & \Gamma_0, (\Delta_0; \Delta''), (\text{skip} \parallel \prod_{q \in Q^* \setminus \{u\}} [B]_q), [\Psi'']_u \\
\end{align*}

\begin{align*}
\Gamma, \Delta, [\text{for } (q : Q) \{ A \}]_p \parallel \prod_{q \in Q} [B; C]_q, \Psi & \sim \Gamma, \Delta', \prod_{q \in Q} [C]_q, \Psi' \\
\end{align*}

Figure 3.5. Rewrite Rules (Iteration over sets of processes).
be the same as before in order to rule out superfluous sends.

**Unfolding.** Figure 3.7 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 3.4.** Consider EX5 shown left below (this example is based on EX4 from Section 3.1). As before, we eliminated wild card receives and assume that there is only a single message type.

\[
\text{for } (q : Q) \{ \\
\text{id} := \text{recv}(Q); \\
\text{send(id, ping)} \\
\} \\
\text{send}(p, q) \\
\text{recv}(p) \}
\]

Our goal is to apply R-LOOP-UPD to produce the sequentialization EX5\text{GOAL}.

\[
\text{EX5}_\text{GOAL} \overset{\text{EX5}}{=} \text{for } (q : Q) \{ \\
\text{id} := q_p; \\
\text{ping}_q \\
\}
\]

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 $\Gamma \vdash \emptyset \subseteq Q^* \subseteq Q$), by unfolding a process from $Q^*$ and rewriting $p$ and the unfolded
process to skip.

\[
I = \begin{Cases}
  id := recv(Q); \\
  send(id, ping) \\
\end{Cases}_p \parallel \prod_{q \in Q^*} \begin{Cases}
  send(p, q); \\
  v := recv(p) \\
\end{Cases}_q
\]

Applying R-RECV-UNFOLD yields the program \(I'\), where \(q^*\) is a fresh PID. We can rewrite \(I'\) into \(skip \parallel \prod_{q \in Q^* \setminus \{q^*\}} \ldots\) with sequential prefix \(\Delta = \left[ [id := q^*]_p ; [v := ping]_{q^*} \right] \) 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 \(EX5_GOAL\).

\[
I' = \begin{Cases}
  id := recv(q^*); \\
  send(id, ping) \\
\end{Cases}_p \parallel \begin{Cases}
  send(p, q^*); \\
  v := recv(p) \\
\end{Cases}_q \parallel \prod_{q \in Q^* \setminus \{q^*\}} \ldots
\]

**Example 3.5.** Consider example \(EX6\), shown below.

\[
EX6 = \begin{Cases}
  for (q : Q) { \\
    send(q, ping); \\
    v := recv(Q) \\
  } \\
\end{Cases}_p \parallel \prod_{q \in Q} \begin{Cases}
  send(p, pong); \\
  w := recv(p) \\
\end{Cases}_q
\]

\(EX6\) 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.

\[
I = \begin{Cases}
  send(q^*, ping); \\
  v := recv(Q) \\
\end{Cases}_p \parallel \prod_{q \in Q^*} \begin{Cases}
  send(p, pong); \\
  w := recv(p) \\
\end{Cases}_q
\]

Applying rule R-SEND-UNFOLD yields the program \(I'\):

\[
I' = \begin{Cases}
  send(q^*, ping); \\
  id := recv(Q) \\
\end{Cases}_p \parallel \begin{Cases}
  send(p, pong); \\
  w := recv(p) \\
\end{Cases}_q \parallel \ldots
\]
However, our method fails to rewrite this program as this would require unfolding a second process to handle the receive from $Q$. 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 3.6 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.** Consider again Figure 3.9. 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 3.6.** Consider example ex7 shown below, in which process $p$ interacts with a
set of processes $Q$.

$$\text{EX7} \triangleq \left[ \begin{array}{l}
\text{for } (i : I) \{ \\
\quad \text{id} := \text{recv}(Q) ; \\
\quad \text{send(id, 0)} \\
\} \\
\} \\
\right]_p \parallel \prod_{q \subseteq Q} \left[ \begin{array}{l}
\text{while true} \{ \\
\quad \text{send(p,q)} ; \\
\quad \text{stop} := \text{recv}(p) ; \\
\quad \text{if } \text{stop} = 1 \{ \\
\quad \quad \text{break} \\
\quad \} \text{else } \{ \\
\quad \quad \text{skip} \\
\quad \} \\
\} \\
\right]_q$$

$p$ 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 $\text{stop}$ and breaks from the loop, if $\text{stop}$ is one. Our goal is to use rule R-LOOP-REPEAT to rewrite $p$ to $\text{skip}$ 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.

$$\left[ \begin{array}{l}
\text{id} := \text{recv}(q^*) ; \\
\text{send(id, 0)} \\
\} \\
\} \\
\right]_p \parallel \left[ \begin{array}{l}
\text{while true} \{ \\
\quad \text{send(p,q^*)} ; \\
\quad \text{stop} := \text{recv}(p) ; \\
\quad \text{if } \text{stop} = 1 \{ \\
\quad \quad \text{break} \\
\quad \} \text{else } \{ \\
\quad \quad \text{skip} \\
\quad \} \\
\} \\
\right]_{q^*}$$
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:

$$\text{for } (i : I) \{ \\
\sum q : Q. ([id \leftarrow q]_p ; [stop \leftarrow 0]_q) \\
\}$$

### 3.3.3 Residual Processes

Finally, Figure 3.8 shows our rules for residual processes. Rule R-SEND-RESID moves a send of value $n$ to process $x$ into the residual process $\Psi$, 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 $B$ (Procs($B$)) are added to the residual process $\Psi$. The rule then composes the residual process with $B$ and rewrites the composition. The rule requires the composition of $\Psi$ and $B$ be symmetrically non-deterministic, indicated by $rf(B \parallel \Psi)$.

**Example 3.7.** Consider again example EX4 (Equation (3.1)). 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 Equation (3.3), where we have removed the non-deterministic choice as the set only contains one element. Next, we compose the residual process with process $m$ as shown in Equation (3.4). 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 shown in Equation (3.5).
R-LOOP-REPEAT

\[ \delta' = \delta; \]
\[ \Delta_0 \triangleq \text{havoc}(\Delta, A, B) \]
\[ \Psi' \triangleq \Psi \parallel \sum q : Q. [\Psi^u]_q \]

\( \Gamma, \Delta_0, (\lfloor A[i^*/i] \rfloor_p \parallel \prod_{q \in Q} [B]_q), \text{skip} \sim \Gamma, (\Delta_0; \Delta^u), (\text{skip} \parallel [B]_u \parallel \prod_{q \notin \{u\}} B), [\Psi^u]_u \)

\[ \Gamma, \Delta, \text{for } (i : I) \{ A \} \parallel \prod_{q \in Q} [B]_q, \Psi \sim \Gamma, \Delta', \prod_{q \in Q} [B]_q, \Psi' \]

**Figure 3.6.** Rewrite Rules (Iteration over sets of indices).

R-SEND-UNFOLD

\[ \Gamma \vdash q^* \in Q \]
\[ \Gamma, \Delta, \lfloor \text{send}(t, q^*, n) \rfloor_p \parallel \prod_{q \in Q} A \parallel [A]_{q^*}, \Psi \sim \]

R-RECV-UNFOLD

\[ q^* \text{ fresh} \]
\[ \Gamma \vdash \emptyset \subset Q' \subseteq Q \]
\[ \Gamma, \Delta, \lfloor x := \text{recv}(t, q^*) \rfloor_p \parallel \prod_{q \in Q'} [A]_{q^*}, \Psi' \sim \Gamma, \Delta', [A]_{q^*}, \Psi' \]

**Figure 3.7.** Rewrite Rules (Unfold)
R-SEND-RESID

$V$ fresh $q$ is a PID
$\Delta \vdash x = q$ $\Delta' = \left( \Delta; [V := V \cup \{ n \}] \right)_p$
$\Gamma \vdash E(q)$ $\Psi' = \left( \Psi \circ \sum v : V. [send(t, q, v)]_p \right)$
$\Gamma, \Delta, [send(t, x, n)]_p, \Psi \leadsto \Gamma, \Delta', \text{skip}, \Psi'$

R-COMPOSE-RESID

$\Gamma_0 \triangleq \Gamma \cup E(p)$ for $p \in \text{Procs}(B)$ $\text{rf}(B \parallel \Psi)$
$\Gamma_0, \Delta, A, \text{skip} \leadsto \Gamma_0, \Delta', \text{skip}, \Psi$ $\Gamma, \Delta', B \parallel \Psi, \text{skip} \leadsto \Gamma, \Delta'', \text{skip}, \text{skip}$
$\Gamma, \Delta, A \parallel B, \text{skip} \leadsto \Gamma, \Delta'', \text{skip}, \text{skip}$

Figure 3.8. Rewrite Rules (Residue)

---

R-WHILE-REPEAT

$\Gamma, \Delta, [A]_p \parallel [B]_q, \Psi \leadsto \Gamma', \Delta', \text{skip}, \Psi'$
$\Gamma, \Delta, [\text{while true} \{ A \}]_p \parallel [B; C]_q, \Psi \leadsto \Gamma', \Delta', [\text{while true} \{ A \}]_p \parallel [C]_q, \Psi'$

R-WHILE-REMOVE

$\Gamma, \Delta, [A]_p \parallel [B]_q, \Psi \leadsto \Gamma', \Delta', \text{break}, \Psi'$
$\Gamma, \Delta, [\text{while true} \{ A \}]_p \parallel [B; C]_q, \Psi \leadsto \Gamma', \Delta', [C]_q, \Psi'$

R-IF-THEN

$\Delta \vdash e$
$\Gamma, \Delta, \text{if } e \{ A \} \text{ else } \{ B \}, \Psi \leadsto \Gamma, \Delta, A, \Psi$

R-IF-ELSE

$\Delta \vdash \neg e$
$\Gamma, \Delta, \text{if } e \{ A \} \text{ else } \{ B \}, \Psi \leadsto \Gamma, \Delta, B, \Psi$

Figure 3.9. Rewrite Rules (Branch)
3.3.4 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 an interpretation on prefixes and contexts such that $(\sigma, \mu) \in \Delta, \Gamma$ when $\sigma$ and $\mu$ are a store and message buffer consistent with the states reachable by executing $\Delta$ and the assumptions in $\Gamma$. 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 $\text{hprocs}(\sigma, \mu, P)$.

**Theorem 3.1.** Let $P$ be a program and assume $P : \checkmark$. If

1. $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$
2. For some extension $E$, $\text{rf}(P \times E)$ and $(\sigma, \mu) \in \Delta, \Gamma$ such that

   $$(\sigma, \mu, \Psi; P \times E) \rightarrow (\sigma_Q, \mu_Q, Q)$$

Then there exists $(\sigma', \mu') \in \Delta', \Gamma'$ such that $(\sigma', \mu', \Psi'; P' \times E) \rightarrow (\sigma'_Q, \mu'_Q, Q')$ and $\sigma_Q|_H = \sigma'_Q|_H$ where $H = \text{hprocs}(\sigma_Q, \mu_Q, Q)$.

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 [45]. Importantly, the case for R-RECV uses the fact that receives are left movers, up to their matching send. The case for R-RECV-UNFOLD relies on introducing a prophecy variable [2] that guesses the sender of the eventually received message. □
The full proof is delegated to Appendix B. Theorem 3.1 implies that we can check reachability for any halted subset of processes, consistent with results from finite-state partial order reduction techniques (e.g., [68]). Thus, if we can rewrite a program $P$ into prefix $\Delta$, then (1) $P$ is deadlock free and (2) process-local safety properties of $\Delta$ are enjoyed by $P$.

3.4 Canonical Sequentialization of $I_S$ Programs

We synthesize canonical sequentializations by implementing our rewrite rules Section 3.3 as a Prolog predicate `rewrite`, shown below. Intuitively, `rewrite($P, \Gamma, \Delta, \Psi, P', \Gamma', \Delta', \Psi'$)` holds if $\Gamma, \Delta, P, \Psi \sim \Gamma', \Delta', P', \Psi'$. In order to rewrite a program $P$, we make the query `rewrite($P, (\text{BUFF}_{\emptyset}, \emptyset), \text{skip}, \text{skip}, \text{skip}, (\text{BUFF}_{\emptyset}, _), \Delta, \text{skip}$)` which rewrites $P$ to `skip` producing its canonical sequentialization $\Delta$.

```prolog
rewrite(P, \Gamma, \Delta, \Psi, P', \Gamma', \Delta', \Psi') :-
    ( P = P', \Gamma = \Gamma', \Delta = \Delta', \Psi = \Psi' ;
      rewrite_step(P, \Gamma, \Delta, \Psi, P'', \Gamma'', \Delta'', \Psi''), !,
      rewrite(P'', \Gamma'', \Delta'', \Psi'', P', \Gamma', \Delta', \Psi')
    ).
```

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, $\Delta \vdash \varphi$, e.g., in R-SEND and R-RECV. As $\Delta$ may contain loops, entailment is undecidable. Computing approximations of $\Delta \vdash \varphi$ is orthogonal to this paper, however, we found that for our benchmarks (Section 3.6), loop invariants that track constants were sufficient.

### 3.5 From HASKELL to $I_S$

$I_S$ is designed to be a convenient target for reasoning about distributed programs. However, is is significantly less convenient to program in than more standard, high-level programming languages. In this chapter, we thus describe a syntax-directed translation from (a subset of) HASKELL programs to $I_S$ programs. First we describe our model of HASKELL programs by giving the syntax and semantics of a core language Section 3.5.1. Next, we formalize our translation to $I_S$ programs in Section 3.5.2. Finally, we argue that our translation is correct in Section 3.5.3 by showing that it is sound, in the sense that translated programs only take steps allowed by the original program, and complete, in the sense that the translated program can take steps corresponding to steps taken by the original program.

#### 3.5.1 Syntax and Semantics of $\lambda_m$

**Syntax.** To formalize our approach of translating HASKELL programs, we define a core calculus called $\lambda_m$ that is intended to model the relevant pieces of HASKELL. $\lambda_m$ is
### Constants

\[ c ::= \text{true} | \text{false} | 0 | 1 | \ldots \]  
**Pure literals**

\[ \text{return} | >>= \]  
**Monadic literals**

\[ \text{self} | \text{send} | \text{recv} | \text{spawn} \]

### Expressions

\[ e ::= c \]  
**Literal**

\[ x^t \]  
**Typed Variable**

\[ \lambda x^t. e \]  
**Abstraction**

\[ \Lambda \alpha e \]  
**Type Abstraction**

\[ e e \]  
**Application**

\[ \text{if} e \text{ then } e \text{ else } e \]  
**Control Flow**

\[ e @t \]  
**Type Application**

\[ \text{let } x^t = e \text{ in } e \]  
**Let Binding**

\[ \text{rec } x. e \]  
**Recursive Binding**

### Types

\[ t ::= \text{int} | \text{bool} | \text{pid} \]  
**Integers, Booleans, & PIDs**

\[ t \rightarrow t \]  
**Functions**

\[ P t \]  
**Effectful Computations**

---

**Figure 3.10.** $\lambda_m$ syntax

A lambda calculus extended with message passing primitives: the message passing (also referred to as the impure or effectful) subset is monadic.

**Definition 3.2 (Pure and impure types).** We say that a type $t$ is impure if $t = P t'$ for some $t'$ or $t = t_1 \rightarrow t_2$ and $t_2$ is impure. If $t$ is not impure, then it is pure.

The syntax of $\lambda_m$ is shown in Figure 3.10. We assume that $\lambda_m$ is statically typed, but the type system itself is unimportant and we do not address its particulars here, save to say that $\lambda_m$ is explicitly typed as in HASKELL’s core calculus [35]: in particular binders $x$ are annotated with their types $t$. When clear from context, or irrelevant, we will often omit $t$.

**Message Passing.** The main primitives for message passing are $\text{send } @t p m$ and $\text{recv } @t$. These functions are parametric in the messages sent or received. Thus, $\lambda_m$ requires an explicit application of the type of message: The expression $\text{send } @\text{int } p m$ is a computation that sends a message $m$ of type int to the process with PID $p$, and $\text{recv } @\text{bool}$ is a process that blocks until it receives a message of type bool.

**Effectful Combinators.** The most primitive effectful computation is $\text{return } @t e$,
which has no observable side effect and, by itself, is irreducible (assuming $e$ cannot be reduced further). Message passing computations are sequenced with $\gg=$. If $m$ is a computation returning a value of type $t$ (such as $\text{recv @} t$), and $\lambda x^t. n$ is a function from values of type $t$ to effectful computations, then their composition is written $m \gg= \lambda x^t. n$, which first executes $m$ and then binds the returned value to $n$ as the value of $x$.

**Semantics.** The semantics of $\lambda_m$ is formalized in Figure 3.11, and is based on [1, 30] in style (and content). We do not give the formalization of the underlying pure calculus, as the details are irrelevant, save to say that there is some reduction relation, $e \rightarrow_\beta e'$, which says that $e$ reduces to $e'$ without side effects, e.g., without any message passing.

$\lambda_m$ **States.** The operational semantics of $\lambda_m$ is given by the relation $\rightarrow_{\lambda_m}$, which
is a binary relation on states. In $\lambda_m$, a state is a pair $(\mu, P)$ of a message buffer $\mu$ (as as in Chapter 2) and a program $P$. Programs $P$ are a series of expressions joined by $\parallel$ and subscripted by the (distinct) PID of the executing process. Therefore, the program $e_0^p \parallel e_1^p \parallel \ldots$ denotes process $p_0$ running expression $e_0^p$ in parallel with $p_1$ running $e_1^p$, etc. We consider states equivalent up to reordering of processes.

**Program Contexts.** The relation $\rightarrow_{\lambda_m}$ is defined using program contexts, which pick a particular subexpression in a program to execute by placing “holes,” $[]$, in place of the expression to reduce. First, evaluation contexts specify the order of evaluation, which is to reduce the first argument of $>>=$ before using the BIND rule to eliminate the $>>=$ application. Next, program contexts pick out a particular process. For example,

$$\mathcal{H}[\text{return } \@t n \gg= m]_p$$

matches the program

$$(\text{return } \@t n \gg= m) \gg= l)_p \parallel (\text{send } \@\text{int } p 0 \gg= f)_q$$

by letting $\mathcal{H} = ([] \gg= l)_p \parallel (\text{send } \@\text{int } p 0 \gg= f)_q$

### 3.5.2 Translation to $I_S$

Next we describe how to compile $\lambda_m$ terms to $I_S$ programs.

**MapReduce in $\lambda_m$.** Figure 3.12 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. A mapper process (lines 4-12) consists of a loop implemented in the function mapperLoop. In the body of the loop, the mapper first sends its PID to the work queue (line 7) and then waits for a response (line 8). If the response is a unit of work, the process sends (line 10) a result (in this case, just the work unit $i$ itself) to the PID of a reducer process and calls mapperLoop to begin the next iteration of
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 = queueLoop
  where
    queueLoop = do me ← getSelfPid
                   workers ← spawn n (mapper me r)
                   foldM distribute () work
                   foldM terminate () workers
    distribute _ i = do x ← receive
                        send x (Work i)
    terminate _ _ = do x ← receive
                      send x Term

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

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

Figure 3.12. MapReduce implemented in HASKELL
the loop. If the response is Term, the loop terminates and the process exits. The work queue (lines 15-24) first spawns n mapper processes (18), and then uses foldM\(^1\) (line 19) to wait for k requests (line 21). For each request, the queue answers with a unit of work Work \(_i\) (line 22). Having distributed k units of work, the queue waits (using foldM) for each mapper to send a request (line 23), and responds with a Term message, causing the mapper to exit (line 24). The reducer process (lines 27-33) 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 on line 39.

**Preliminary Restrictions.** In this dissertation we are considering a subset of \(\lambda_m\) programs. First, we consider defunctionalized (e.g., [51]) terms: functions are allowed as arguments to built in combinators, but otherwise we consider first-order terms. Second, we make the simplifying assumption that all of the spawn expressions in a \(\lambda_m\) program occur before any of the other effectful computations: This means that we can consider the translation to be a flat composition of processes: the translation itself is not defined when spawns may occur in the body of a loop. Second, to ease the presentation of the translation, we assume that spawned processes do not close over their environment: this behavior is equivalent to sending an initialization message to each newly spawned process.

**Translation.** We traverse the abstract syntax tree (AST) of a \(\lambda_m\) term \(e\) to compute the corresponding I\(_S\) program, using the types of the child expressions to guide the translation. If the type of an expression is of the form Process \(t\), then the term corresponds to a tree of expressions where each node is an application of \(\gg\gg\gg\), which corresponds to sequencing and assignment.

In Figure 3.13, we define the function \(T(e, x, p)\) which maps a \(\lambda_m\) term \(e\), variable \(x\) and process identifier \(p\) to a I\(_S\) program. If \((A, B) = T(e, x, p)\), then \(A\) is a program corresponding to the “root” process, \(p\), of \(e\). \(A\) assigns to \(x\) the final value computed by \(p\). \(B\) is a flat composition of processes spawned during the execution of \(E\).

\(^1\)The foldM combinator is a standard left-fold where the “fold function” can be effectful.
\(T(self, x, p)\) \[= T\) (return \@pid \(p, x, p\)\)

\(T\) (send \@t \(p, m, x, q)\) \[= \{[send(t, p, \(m\)]_q, \varnothing\}\)

\(T\) (recv \@t, \(x, q)\) \[= \{[x := \text{recv}(t, \*)]_q, \varnothing\}\)

\(T\) (return \(e, x, q)\) \[= \{[x := \(e\)]_q, \varnothing\}\)

\(T\) (spawn \(e, x, q)\) \[= \{[x := p]_q, A\) \]

where \(A = T_1(e, \text{ret}_p, p) \\quad p \text{ fresh}\)

\(T\) (\(m >>= \lambda y^t. n, x, q)\) \[= (A_1; A_2, q_1 \parallel q_2) \]

where \((A_1, q_1) = T(m, y, q) \\quad (A_2, q_2) = T(n, x, q)\)

\(T\) (let \(y^t = m\) in \(n, x, q)\) \[= \{T\) (return \(m >>= \lambda y^t. n, x, q)\) Pure(t)\]

\(T\) (if \(e\) then \(m\) else \(n, x, q)\) \[= (if e \{ A_1 \} else \{ A_2 \}, q_1 \parallel q_2) \]

where \((A_1, q_1) = T(m, x, q) \\quad (A_2, q_2) = T(n, x, q)\)

\(T\) (foldM(\(\lambda a^t. \lambda y^v. e\), \(b\), \(xs, q, x)\)) \[= ([a := b]_q; \text{for } (i : I) \{ A \}, \varnothing) \]

where \(A = T_1(e, a, q)\)

\(T\) (\(\text{LOOP } f \ c \ m \ n\) (rec \(f. \lambda x^t. b), y, p)\) \[= (L(f, x, A); \text{while true } \{ L(f, x, B) \}, \varnothing) \]

where \(A = T_1((\text{LOOP } f \ c \ m \ n), y, p) \\quad B = T_1(e, x, p)\)

\(T\) (\(\text{rec } f. (\lambda x^t. e)\) \(m, y, q)\) \[= ([x := \(m\)]_q; \text{while true } \{ L(f, x, A) \}, \varnothing) \]

where \(A = T_1(e, y, q)\)

\(L(f, x, \text{skip})\) \[= \text{break}\)

\(L(f, x, [f e]_p)\) \[= [x := \(e\)]_p\)

\(L(f, x, [s]_p, A)\) \[= [s]_p; L(f, x, A)\)

\(\text{K}(f, x, A_1 \sqcap A_2)\) \[= L(f, x, A_1) \sqcap L(f, x, A_2)\)

\(L(f, x, A)\) \[= A\)

\(T_1(e, x, p)\) \[= A \quad \text{where } (A, \varnothing) = T(e, x, p)\)

(LOOP \(f c m n)\) \[= \lambda f. \text{if } c \text{ then } m \text{ else return } \@t n\)

Figure 3.13. Transformation of \(\lambda_m\) to \(I_s\)
**Basic Expressions.** The unit of effectful $\lambda_m$ programs is the $\text{return }@t e$ expression. By itself, $\text{return }@t e$ denotes a program that evaluates $e$ and terminates. Thus, the $I_5$ program corresponding to $\text{return }@t e$ is a singleton process that assigns the value of $e$ to its “return variable.”

Basic effectful primitives for sending and receiving messages and spawning processes are transformed into their $I_5$ counterparts. The explicit type applications of $t$ in $\text{send }@t p m$ and $\text{recv }@t$ are used to generate the corresponding $I_5$ primitives, which require a type (channel) identifier. Note that translated receives are always “wildcard” receives: the wildcards can be eliminated by techniques discussed previously in this dissertation. It is possible to extend the language with a primitive that receives from a particular process, which would require extending the translation.

**Spawning Processes.** Although our goal is not the analysis of programs that use dynamic process creation heavily, we nevertheless include a case for handling some process creation (assumed to occur before the main logic of the program under translation). Thus, the case for $T(\text{spawn } e, x, q)$ computes the translation of $e$ with a freshly chosen PID, $q$. The translated program corresponding to the spawn itself simply returns the new PID by assigning it to $x$.

**Sequencing Computations.** Effectful computations $m$ and $\lambda x^t. n$ are sequenced in $\lambda_m$ using $\gg=\gg$. Thus, the translation is built by translating $m$ and $n$ separately, using $x$ as the “return variable” of $m$.

With an eye towards verification, we optimize the translation by translating certain $\text{let}$ expressions into sequenced $I_5$ programs. In particular, while $\text{let } x = m \text{ in } n$ is often treated as syntactic sugar for $\lambda x^t. n \ m$, we consider the case when $m$ is a pure expression (and hence reduces to a pure value via $\beta$-reduction). We translate this case by binding the value of $m$ to $x$ rather than substituting $m$ for $x$, which allows sharing of the value via $x$ in the final program.

**Recursive Functions.** In general, the most significant mismatch between $\lambda_m$ and $I_5$ is that $\lambda_m$ employs recursive functions while $I_5$ uses while-loops to perform
general iteration. While it would be possible to reify the function stack by transforming \( \lambda_m \) terms into their continuation-passing style form, this would require higher-order reasoning about IS terms. Instead, we require that *effectful* recursive \( \lambda_m \) functions be essentially (though not strictly speaking) tail-recursive: intuitively, the recursive call should be performed as the *last action*. This requirement is expressed as a well-formedness criterion for all recursive computations. We write \( \vdash \text{rec} \ f. \ e \) to say that \( \text{rec} \ f. \ e \) is well formed. The definition of \( \vdash \text{rec} \ f. \ e \) is given in Figure 3.14, and closely follows the definition of tail recursion for a function \( f \).

Observe that if \( \vdash \text{rec} \ f. \lambda x. \ e \), then then \( e \) is *equivalent* (assuming it is an effectful computation) to a term of the form \( \text{m} \gg= \lambda y. \text{if c then f a else b} \) where \( f \) does not appear freely in \( m \) or \( b \). In other words, the decision to recurse can be deferred to the last step of the computation.

**Example 3.8** (MapReduce in IS). The Haskell code for `mapper` from Figure 3.12 is thus translated to a while loop in the extracted IS program shown in Figure 3.15, which also shows the send and receive tags, derived from the types as described in Section 3.4. 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. Note that since the receive at (1) is tagged with \( b \), which appears in the worker process, the
statement is equivalent to $x := \text{recv}(\text{Int}, P)$. Likewise, the receive at (2) is equivalent to $w_p := \text{recv}(\text{Message}, q)$ and wildcards in (3) and (4) may be instantiated to $P$.

### 3.5.3 Correctness of the Translation

The correctness criteria we are concerned with are (1) soundness of the translation from $\lambda_m$ to $I_s$, in the sense that if translating $e$ produces the program $A$, then the transitions of $A$ should correspond to only valid transitions in $e$; and (2) completeness, in that if $e$ can transition to $e'$, then the translation of $e$ can transition to the translation of $e'$. Intuitively, the monadic structure of $\lambda_m$ makes this correspondence easy to see: in general, there is a one-to-one correspondence between the (impure) semantics $\lambda_m$ and the semantics of $I_s$.

**Lifting translations to $\lambda_m$ states.** In order to talk about the correspondence between executions in the two languages, we overload the function $T$ to translate $\lambda_m$
program contexts. Let $\gamma : \text{PID} \rightarrow \text{Identifiers}$ map PIDs to the “return variable” of each process. We lift the definition of $T$ to contexts by applying $T$ to each parallel-composed process.

**Definition 3.3 (Translation of programs).** We lift the function $T$ to programs by defining $T(\mathcal{P}, \gamma)$ as follows

$$T(\mathcal{P}, \gamma) \triangleq \begin{cases} T(e, \gamma(p), p) \parallel T(\mathcal{P}', \gamma) & \mathcal{P} = e_p \parallel \mathcal{P}' \\ \text{skip} & \mathcal{P} = \epsilon \end{cases}$$

**Soundness.** In order to prove soundness, we want to map each execution in $I_S$ to an execution in $\lambda_m$. In turn, this requires decoding configurations $C$ from $I_S$ to states in $\lambda_m$. Given $C = (\sigma, \mu, A)$, we need to find all the $e_p$ such that the composition of all $T(e, x, p)$ produce $A$. As this may be an open term in general, we then produce a closed term by unloading the store $\sigma$ (i.e., producing $e[\sigma(p, x)/x]$ for each $x$ in the domain of the store).

**Decoding $I_S$ configurations.** First we need the following definitions, which handle closing open $\lambda_m$ terms and producing terminated processes.

**Definition 3.4 (Store Unloading).** We define the function $\text{Apply}(\sigma, p, e)$ which unloads the store $\sigma$ on the term $e$ as

$$\text{Apply}(\sigma, p, e) \triangleq e[v_0/x_0][v_1/x_1] \ldots \text{ for } \sigma(p, x_i) = v_i$$

**Definition 3.5 (Closing a Term).** We define the function $\text{Close}(\sigma, \gamma, \mathcal{P})$, which closes the program context $\mathcal{P}$ as

$$\text{Close}(\sigma, \gamma, \mathcal{P}) \triangleq \begin{cases} \text{Apply}(\sigma, p, e_p) \parallel \text{Close}(\sigma, \gamma \setminus p, \mathcal{P}') & \mathcal{P} = e_p \parallel \mathcal{P}' \\ (\text{return } @ t \sigma(\gamma(p))))_p & \text{for } p \in \gamma \quad \mathcal{P} = \epsilon \end{cases}$$

**Terminated Processes.** There is a slight impedance mismatch between processes
that have terminated in $I_s$ versus those in $\lambda_m$. In particular, a halted process $p$ in the semantics of $\lambda_m$ is of the form $(\text{return }@t \ v)_p$ for some value $v$. A corresponding configuration in Lang is some $(\sigma, \mu, \text{skip})$, where $\sigma(p, \text{ret}_p) = v$. Thus, when decoding such a configuration, we must produce the corresponding $(\text{return }@t \ v)_p$ term.

**Loops.** The other tricky part of the proof is the difference between the execution of recursive functions in $\lambda_m$ versus loops in $I_s$. That is, for some $s$,

$$T((\text{rec } f. (\lambda x. e)) \ m, y, p) = [x := m]_p \text{; while true } \{ s \}.$$  

$[x := m]_p \text{; while true } \{ s \}$ can step to while true $\{ s \}$, which does not correspond to the translation of a particular $\lambda_m$ expression to the translation of a particular $\lambda_m$ expression. Thus, it is necessary to account for these intermediate states reachable during the course of execution. Thus, the decoding function must first check to see if the current configuration is the result of unrolling a loop.

As $T$ is not invertible in general, we must decode $I_s$ programs as sets of $\lambda_m$ programs. For this, we define $D$ (and some helpers). $D$ walks the parallel composed $A_0 \parallel A_1 \parallel \ldots$ searching for expressions that translate to the corresponding $A_i$.

**Definition 3.6 (Decoding $I_s$ Programs).** We decode $I_s$ programs by defining the function $D$ as follows.

$$D_0(A, \gamma) \triangleq \{(e_p \parallel \mathcal{P}) \mid A = (T(e, \gamma(p), p) \parallel B) \land \mathcal{P} \in D(B, \gamma)\}$$

$$D_1(A, \gamma) \triangleq \begin{cases} ((\text{rec } f. (\lambda x. e)) \ x)_p \parallel \mathcal{P} & A = (\text{while true } \{ \mathcal{L}(f, x, s) \parallel B \} \\ \land e \in D(s, \gamma) \land \mathcal{P} \in D(B, \gamma) \end{cases}$$

$$D(A, \gamma) \triangleq \begin{cases} \{ \epsilon \} & A = \text{skip} \\ D_0(A, \gamma) \cup D_1(A, \gamma) & \text{otherwise} \end{cases}$$

The helper function $D_1(A, \gamma)$ accounts for the aforementioned difference between recursive functions in $\lambda_m$ and while loops in $I_s$. 


We can now define how to decode $I_s$ configurations:

**Definition 3.7.** Given a configuration $C = (\sigma, \mu, A)$ and map $\gamma$, the **decoding** of $C$, $[C]_\gamma$, is defined:

$$[(\sigma, \mu, A)]_\gamma \triangleq \{(\mu, \text{Close}(\sigma, \gamma, \mathcal{P})) \mid \mathcal{P} \in D(A, \gamma)\}$$

**Theorem 3.2 (Soundness).** For all well-formed $e$, $(\emptyset, \emptyset, \mathcal{T}(e, \text{ret}_p, p)) \rightarrow_{1_k} (\sigma, \mu, \text{skip}) \Rightarrow (\emptyset, e) \rightarrow_{\lambda_m} (\mu, \text{Close}(\sigma, \mathcal{E}))$.

**Lemma 3.1.** Assume $C \rightarrow_{1_k} C'$ Then, for all $(\mu, \mathcal{P}) \in [C]_\gamma$, there exists $(\mu', \mathcal{P}') \in [C']_\gamma$ such that $(\mu, \mathcal{P}) \rightarrow_{\lambda_m} (\mu', \mathcal{P}')$.

**Completeness.** The completeness of the translation execution again follows from a correspondence between states. However, the inductive lemma depends on knowing the initial $\sigma$ corresponding to the starting $\lambda_m$ state, as there is no function from $\lambda_m$ states to $I_s$ configurations. Thus, given a $\sigma$, we consider $I_s$ terms that correspond to $\lambda_m$ terms closed-over by $\sigma$.

**Theorem 3.3 (Completeness).** For all well-formed $e$, when $\mathcal{T}(e, \text{ret}_p, p)$ is defined, $(\emptyset, e) \rightarrow_{\lambda_m} (\mu, \text{Close}(\sigma, \lambda p.\text{ret}_p, \mathcal{E})) \Rightarrow (\emptyset, \emptyset, \mathcal{T}(e, \text{ret}_p, p)) \rightarrow_{1_k} (\sigma, \mu, \text{skip})$.

**Proof.** The theorem follows from Lemma 3.2.

**Lemma 3.2.** For all well-formed $E$ and $\sigma$, if $\mathcal{T}(E, \gamma)$ is defined, then there exists $E''$ such that $\mathcal{T}(E'', \gamma)$ is defined, $\text{Close}(\sigma', \gamma, E'') = E'$ and

$$\langle \mu, \text{Close}(\sigma, \gamma, E) \rangle \rightarrow_{\lambda_m} \langle \mu', E' \rangle \Rightarrow (\sigma, \mu, \mathcal{T}(E, \gamma)) \rightarrow_{1_k} (\sigma', \mu', \mathcal{T}(E'', \gamma))$$

### 3.6 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,
Table 3.1. **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.

<table>
<thead>
<tr>
<th>Name</th>
<th>#Param</th>
<th>#LOC</th>
<th>SPIN</th>
<th>Iₜ</th>
<th>#Term</th>
<th>BRISK time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX2</td>
<td>1</td>
<td>14</td>
<td>-</td>
<td>69</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>EX3</td>
<td>1</td>
<td>13</td>
<td>11</td>
<td>57</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>PINGDET</td>
<td>1</td>
<td>17</td>
<td>13</td>
<td>83</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>PINGITER</td>
<td>1</td>
<td>19</td>
<td>11</td>
<td>63</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>PINGSYM</td>
<td>1</td>
<td>13</td>
<td>10</td>
<td>44</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>PINGSYM2</td>
<td>1</td>
<td>43</td>
<td>7</td>
<td>140</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>CONCDB</td>
<td>1</td>
<td>54</td>
<td>6</td>
<td>265</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>DISTDB</td>
<td>2</td>
<td>42</td>
<td>2</td>
<td>218</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>FIREWALL</td>
<td>1</td>
<td>45</td>
<td>9</td>
<td>201</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>LOCKSERVER</td>
<td>1</td>
<td>28</td>
<td>12</td>
<td>109</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>MAPREDUCE</td>
<td>2</td>
<td>64</td>
<td>4</td>
<td>205</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>PARIKH</td>
<td>0</td>
<td>35</td>
<td>-</td>
<td>173</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>REGISTRY</td>
<td>1</td>
<td>40</td>
<td>10</td>
<td>171</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>TWOBUYERS</td>
<td>0</td>
<td>59</td>
<td>-</td>
<td>332</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>2PCOMMIT</td>
<td>1</td>
<td>47</td>
<td>6</td>
<td>281</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>WORKSTEAL</td>
<td>2</td>
<td>39</td>
<td>5</td>
<td>141</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>THEQUE</td>
<td>3</td>
<td>576</td>
<td>3</td>
<td>1443</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

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 Figure 3.1 by treating fail as
Figure 3.16. 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.

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 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 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 Section 3.1.

- **CONCDB, PARIKH [21]** and **DISTDB [47]** are implementations of key-value stores.

- **FIREWALL [21]** is a firewall mediating communication between a worker and a server.

- **REGISTRY [73]** 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 Figure 3.15, where \( n \) workers compete for \( k \) jobs.

• **LOCKSERVER** [81] implements a lock service.

• 2PCOMMIT [43] is the classic two-phase-commit protocol for atomic commitment.

• **TWOBUYERS** [33] is a protocol where two “buyers” negotiate a purchase from a “seller.”

• **THEQUE** is a prototype distributed file store that we developed, inspired by the cluster file system used in DISCO [20].

**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 3.16 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 Figure 3.16). 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 3.1, 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. IS **#Term** indicates the size of
the intermediate IS 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.

3.7 **Summary**

In this chapter 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.
Acknowledgments

Chapter 3, in full, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017. The dissertation author was the primary investigator and author of this paper.
Chapter 4

Pretend Synchrony

In Chapter 3 we saw that it is possible to reduce certain kinds of distributed system to single threaded programs. In principle, this should mean that verifying the correctness of implementations should be made simpler by producing canonical sequentializations, for an appropriate notion of correctness. Indeed, for some generic properties such as deadlock-freedom, the (derivation of a) sequentialization itself serves as a certificate to the correctness of the program. However, the question of to what extent sequentialization may be employed to simplify proofs of functional correctness remains murky.

In this chapter we consider a rewriting system in the vein of Chapter 3 to study this question. We formalize an approach that lets the user annotate the program with invariants that assume a synchronous execution model, and hence need not consider the status of undelivered messages. The rewriting logic described in this chapter propagates the user annotations to the rewritten program, at which point a standard algorithm can be used to generate and check the verification condition for the program.

The work is motivated by considering the problem of achieving consensus in a distributed system. Noting that canonical sequentializations may not exist for well known consensus protocols such as Raft [55] or Paxos [41], the rewriting logic presented in this chapter can be viewed as a generalization of the approach in Chapter 3. Rather than constraining the logic to only produce sequential programs, we describe a logic that (like canonical sequentializations) synchronizes the communication between processes.
but (unlike canonical sequentializations) may produce a coarse-grained parallel program, rather than a sequentialization.

In this chapter we describe the following contributions:

1. **Synchronization.** Our first contribution is an algorithm to automatically compute synchronizations (Section 4.3). Our algorithm proceeds by iteratively applying a number of local rewrite rules each of which transforms the input program into a new program comprising a synchronized prefix and a suffix that still needs to be rewritten. The algorithm is sound in the sense that the rewritten program preserves the halting states of the program (Theorem 4.1).

2. **Verification.** Our second contribution is a rely-guarantee based method for generating verification conditions for the synchronized program, that traverses the program in a syntax-directed fashion, and composes the code with the synchronization invariants to emit a logical formula whose validity implies the correctness of the original source (Section 4.4).

3. **Evaluation.** Our third contribution is an implementation and empirical evaluation of pretend synchrony in a tool called CHAI. We apply CHAI to four challenging case studies: the classic two phase commit protocol, a distributed key-value store, the Raft leader election protocol and single decree Paxos (Section 4.5). We show that all these protocols have simple and intuitive synchronous invariants that CHAI checks automatically. Our evaluation demonstrates that pretend synchrony significantly reduces proof effort. While other approaches relying directly on asynchronous invariants have annotations burdens between 9x [31] and 90x [82], pretending synchrony shrinks the proof burden to just around 0.3x, thereby drastically easing verification by more than an order of magnitude.

The chapter is organized as follows. We give an overview of our approach in Section 4.1. In Section 4.2 we briefly discuss extensions to $I_s$. In Section 4.3 we formalize our approach to pretending synchrony as a set of rewrite rules. In Section 4.4 we discuss
verification condition generation for synchronized programs. In Section 4.5 we evaluate our approach on a set of case studies. Finally, we make concluding remarks in Section 4.6.

4.1 Overview

We begin with an overview of our approach. First, we motivate pretend synchrony at a high-level. Next, we describe the individual ingredients that make our approach feasible.

Two-Phase Commit. Figure 4.1 shows the classic two phase commit protocol [43]. In this protocol, a coordinator node \( c \) tries to commit a value to a number of database nodes \( \text{dbs} \). The protocol is made up of two rounds. In the first round, the coordinator loops over all nodes to send them its proposal value. Each database node then nondeterministically chooses to either commit or abort the transaction and sends its choice to the coordinator. If at least one of the nodes chose to abort, the coordinator aborts the entire transaction by setting the appropriate flag. In the protocol’s second round, the coordinator broadcasts its decision to either commit or abort to the database nodes which reply with an acknowledgement. Finally, if the coordinator decided to commit the transaction, each database node sets its value to the previously received proposal.

Correctness. In order to prove correctness of our implementation of two phase commit, we want to show that, if the protocol finished and the coordinator decided to commit the transaction, all database nodes have indeed chosen the proposed value. More formally, we require the following assertion to hold after the protocol execution.

\[
\forall p \in \text{dbs}: \text{c.committed} = \text{true} \Rightarrow p\text{.value} = \text{c.proposal}
\] (4.1)

Asynchronous Invariants: Complicated. Let us first consider how to prove this property in an asynchronous setting. We follow the proof\(^1\) from [80]. Consider the

\footnote{For the full invariant, see https://github.com/DistributedComponents/disel/blob/master/Examples/TwoPhaseCommit/TwoPhaseInductiveInv.vlines 56-198.}
data Proposal = Prop Value
data Decision = Commit | Abort
data Ack = Ack

committed := false;
abort := false;
prop := init;

for (p : dbs) { (1a)
    send(p, (c, Prop prop))
}

for (p : dbs) { (2b)
    msg := recv;
    if msg = Abort {
        abort := true
    }
};

if abort = false {
    reply := Commit;
    committed := true
} else {
    reply := Abort
};

for (p : dbs) {
    send(p, reply)
};

for (p : dbs) { (1b)
    (id, Prop val) := recv;
}

if *{
    msg := Commit
} else {
    msg := Abort
};

send(id, msg); (2a)

decision := recv;

if decision = Commit {
    value := val
};

send(id, Ack)

Figure 4.1. Two phase commit. Arrows denote the flow of messages during each logical phase of the protocol. The boxes of code at either end of an arrow will be synchronized by our method. Each for loop is annotated with an invariant [I].
coordinator’s first loop in Figure 4.1 and let done denote the set of all database nodes for which the send at location (1a) has been executed, so far. In order to rule out messages appearing “out of thin air”, we need to assert that whenever \( p \notin done \), then there are no messages from \( c \) to \( p \), and \( p \) has not yet executed the corresponding receive at location (1b). If, \( p \in done \), we need to case split over the location of \( p \) due to the asynchronous nature of communication. Either

1. There is an in-flight message from \( c \) to \( p \) that contains \( c \)'s process id and proposal value and \( p \) is waiting to receive the message, or

2. The process \( p \) received \( c \)'s message, set its \( val \) variable to \( proposal \) and decided to either commit or abort the transaction, but did not respond yet, or

3. The process \( p \) responded, relaying its decision to \( c \).

We need a similar case split for \( c \)'s second loop: if \( p \notin done \) then either

1. There is a pending message from \( c \) to \( p \) containing \( c \)'s process id and proposal, or

2. The process \( p \) has chosen to commit or abort but has not yet sent a response, or

3. The process \( p \) has sent its response consisting of either a commit or abort message.

Finally, if \( p \in done \), then \( p \) must have finished the first part of the protocol and \( val \) must be set to \( c \)'s proposal. The invariant for the second part of the protocol consists of a similar case split.

**Asynchrony Makes Verification Undecidable.** While avoiding such case splits — over the joint state of the coordinator, the database nodes and the message buffer — would be desirable in itself, there is a more fundamental issue with proving correctness in an asynchronous setting: directly including the message buffer into the system state by modeling it as an array requires nested array reads \(^2\) which makes even checking a candidate invariant undecidable [8]

\(^2\)A message buffer is a function \((\text{processID} \rightarrow \text{processID} \rightarrow \text{Type} \rightarrow \text{Int}) \rightarrow \text{Value}\) that maps a typed channel \((i.e., \text{messages of a particular type between two process})\) and a message counter for that channel to a value.

Pretend Synchrony. Even though the program behaves asynchronously when executed, we can soundly pretend that communication is synchronous when reasoning about the program. Consider, for example, the send marked with (1a) in Figure 4.1 and assume that in the current iteration coordinator \( c \) sends to some process \( p \). Recalling the reasoning from Chapter 3, the message can only be received at a single receive, namely the receive of \( p \) at the location marked with (1b) in Figure 4.1 (the following receive is expecting a value of type Decision). Moreover, the send is independent of all sends between \( c \) and other processes in \( \text{dbs} \).

Synchronous Invariants: Simple. Our method exploits this insight in the following way: instead of writing invariants for the asynchronous program, the user writes invariants as if the program were synchronous, effectively treating matching send and receive pairs as assignments. The synchronous invariants are given as annotations to for loops. Consider again Figure 4.1. For the first loop, we need the following invariant \( I_1 \) stating that, if the loop was executed for some process \( p \) (indicated by \( p \in \text{done} \)), then \( p \) must have been assigned \( c \)'s proposal value.

\[
I_1 \triangleq \lambda \text{done}. \forall p. p \in \text{done} \Rightarrow p.\text{val} = c.\text{proposal}
\]

Invariants \( I_2 \) and \( I_3 \) are trivial, as the respective loops do not modify any relevant values, and need not be supplied by the user. Finally, invariant \( I_4 \) states that whenever the loop has been executed for a process \( p \), process \( p \)'s value variable must be set to the proposal it received in the first round.

\[
I_4 \triangleq \lambda \text{done}. \forall p. p \in \text{done} \wedge c.\text{committed} = \text{true} \Rightarrow p.\text{value} = p.\text{val}
\]

Together, these simple synchronous invariants – free of the case-splits needed to account for asynchrony – are sufficient to prove the correctness property for the two phase commit implementation.

Checking the Synchronous Invariant. We verify the correctness of the syn-
for (p \in \text{dbs}) 
[(id, val) \Rightarrow (c.c.prop)]_p

for (p \in \text{dbs}) 
if *{
    
    msg \Leftarrow \text{Commit}
} else {
    
    msg \Leftarrow \text{Abort}
}

[msg \Leftarrow p.msg;]
    if msg = \text{Abort} {
    abort \Leftarrow \text{true}
}

if abort = false {
    
    reply \Leftarrow \text{Commit};
    committed \Leftarrow \text{true}
} else {
    
    reply \Leftarrow \text{Abort}
}

for (p \in \text{dbs})
[decision \Leftarrow c.reply]_p

for (p \in \text{dbs})
if decision = \text{Commit} {
    
    value \Leftarrow val
} else {
    
    _ \Leftarrow \text{Ack}
}

Figure 4.2. Synchronization of two phase commit.
chronous invariant in two steps. First, we compute the synchronization; a semantically equivalent synchronous program, by iteratively applying a set of rewriting rules (Section 4.3). Second, we use the synchronization to generate and check verification conditions that ensure that the supplied invariants are inductive and imply the desired correctness property (Section 4.4).

In the first step, our implementation computes this synchronization completely automatically without relying on the user supplied invariants. Each rewriting step produces a new program that comprises a synchronized prefix and a suffix that still needs to be rewritten. This process is repeated till the suffix is reduced to skip. Figure 4.2 shows the synchronization that CHAI computes for two phase commit. Intuitively, the synchronous version matches up the connected blocks from Figure 4.1. In the synchronized version, all protocol steps are executed one after the other: first, the coordinator assigns its proposal to all database nodes; then, each node decides to either commit or abort; then, the coordinator assigns its decision to each of the database nodes, and, finally, each node assigns the proposed value in case the coordinator decided to commit.

In the second step, CHAI then uses the synchronization and the user supplied invariants to compute verification conditions that ensure that the program satisfies all assertions. CHAI checks the verification conditions by discharging them to an SMT-solver. Importantly, the verification conditions for checking the invariants in the synchronous setting fall into the array property fragment [8] and hence, checking the invariant is decidable.

### 4.2 Verification with $I_s$

In order to build a verification framework based on $I_s$ programs, we extend the language as presented in Chapter 2, shown in Figure 4.3. These extensions mostly add new forms for specifying (and ultimately verifying) loop invariants.

**Invariants.** In $I_s$, each for-loop is annotated with two loop invariants $\{I_C, I_S\}$. 
### Expressions

| $e$ ::= | $c$ &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&n

### Statements

| $s$ ::= | $i.x = e$ &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&n

### Programs

| $A$ ::= | $\text{skip}$ &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&n

### Figure 4.3. $I_5$ Syntax. Changes with respect to Figure 2.1 highlighted in gray.

The first, $I_C$, is a communication invariant that is checked during the rewrite to a synchronous program (Section 4.3). We find that, in practice, these invariants are usually simple. For all our examples, CHAI computes communication invariants completely automatically. The invariant $I_S$ is a synchronous data invariant, i.e., a loop invariant over the synchronization. Synchronous data invariants are user supplied and used by our verification condition generation procedure (Section 4.4), which checks that the invariants indeed hold on the synchronization.

**Atomic Actions.** The language includes atomic actions $(P \triangleright e)$, which are not written by the user – instead, they are computed by our rewriting rules (Section 4.3). An action $(P \triangleright e)$ comprises a program $P$ that executes atomically and an annotation $e$
that is derived from the supplied synchronous data invariants, and used to generate verification conditions (Section 4.4). We sometimes omit $e$ if it equals true.

### 4.3 Rewrite Rules

In this section, we formalize our approach by describing our rewriting rules for producing synchronizations. We focus our attention on programs that iterate over sets of processes, only, and briefly discuss how our method extends to while loops later (Section 4.3.5).

**Symbolic States.** Each rewriting rule defines a relation between a pair of **symbolic states**, whose syntax is summarized in Figure 4.4. A symbolic state comprises a context ($\Gamma$); a synchronized prefix ($\Delta$); and the program ($P$) to be rewritten. Synchronized prefixes $\Delta$ are $I_s$ programs without sends or receives. A context $\Gamma$ consists of a symbolic message buffer $\text{BUFF}$ and a set of assertions $\text{ASRT}$. $\text{BUFF}$ maps each channel to the sequence of pending messages on that channel, i.e., $\text{BUFF}(p,q,t)$ returns the sequence of pending messages of type $t$ sent from $p$ to $q$. $\text{ASRT}$ contains assertions about process identifiers and is populated during the rewrite. Finally, for a context $\Gamma \triangleq (\text{BUFF}, \text{ASRT})$ we write $\Gamma(p,q,t)$ to mean $\text{BUFF}(p,q,t)$ and $\Gamma \vdash a$ to mean $a \in \text{ASRT}$.

**Rewriting Rules.** Each rewriting rule defines a judgment of the form $\Gamma, \Delta, P \rightarrow \Gamma', \Delta', P'$. The goal of each step is to move parts of program $P$ into the prefix $\Delta$ such that eventually we can rewrite $P$ to skip. We now describe the main rules of our method, starting with rules for loop-free programs, programs that contain communication between a single process and a group of processes, and finally communication between
sets of processes.

4.3.1 Rewriting loop-free programs

Figure 4.5 shows our main rules for rewriting loop-free programs. These rules are essentially the same as their counterparts from Chapter 3.

Message Passing. Rule R-SEND handles message sends. The rule rewrites a send from process \( p \) to some \( x \) into \( \text{skip} \) if \( x \) evaluates to some PID \( q \). The rule checks that \( x \) corresponds to some PID \( q \) through the condition \( \Delta = x = q \). The rule updates \( \Gamma \) by enqueuing the expression \( n \) in the buffer for the respective channel between \( p \) and \( q \), where we use \( \Gamma[a \leftarrow b] \) to denote the function that returns the same values as \( \Gamma \) on all inputs except for \( a \) where it returns \( b \). Rule R-RECV rewrites a receive from some \( x \) into \( \text{skip} \), if \( x \) corresponds to a PID \( p \). R-RECV dequeues the first message \( m \) from the respective channel and sequences the corresponding assignment after \( \Delta \).

Branching and Context. Branching is handled through nondeterministic choice, where \( A \sqcup B \) is a choice between \( A \) and \( B \). The rule R-CHOICE rewrites a program with branching by rewriting each branch independently, where both \( A \) and \( B \) must be rewritten to \( \text{skip} \). The resulting synchronous prefix consists of a choice between the prefixes of the individual branches. The rule R-FALSE rewrites unreachable program points to \( \text{skip} \). Rule R-CONTEXT allows rewriting a program \( A \) independently of statements that are executed after, or in parallel. Our method contains an additional rule, R-Congruence (omitted for brevity, but analogous to Chapter 3), which allows rewriting program \( A \) to \( B \) if \( A \equiv B \). For instance, the rule allows rewriting \( \text{skip}; P \) to \( P \), for any program \( P \).

Example 4.1. Consider example EX1, where we have replaced wildcard receives with receives from the respective processes.

\[
\text{EX1}_{\text{async}} \triangleq \left[ \begin{array}{l}
\text{send}(q, \text{ping}); \\
\wedge := \text{recv}(q)
\end{array} \right]_p \parallel \left[ \begin{array}{l}
\nu := \text{recv}(p); \\
\text{send}(p, \text{Ack})
\end{array} \right]_q
\]
The goal is to rewrite $\text{Ex1}_{\text{ASYNC}}$ to skip producing the synchronization $\text{Ex1}_{\text{SYNC}}$:

\[
\text{Ex1}_{\text{SYNC}} = \left[ v := \text{ping} \right]_q ;
\]

\[
\left[ _- := \text{Ack} \right]_p
\]

For this, we define the initial context as $\Gamma_0 \equiv (\text{BUFF}_\varnothing, \varnothing)$ where the empty buffer $\text{BUFF}_\varnothing$ maps every channel to an empty sequence, and we assume a single messages type, $\tau$.

We first apply R-CONTEXT to select the send statement in $p$’s program, and then apply rule R-SEND to reduce the send to skip, producing the program below and updating the buffer to $\text{BUFF}_\varnothing[(p, q, \tau) \leftarrow \text{ping}]$.

\[
\begin{align*}
\begin{bmatrix}
\text{skip;} \\
_- := \text{recv}(q)
\end{bmatrix}_p & \parallel \\
\begin{bmatrix}
_\vdash := \text{recv}(p); \\
\text{send}(p, \text{Ack})
\end{bmatrix}_q
\end{align*}
\]

Next, we apply R-CONTEXT with R-RECV to yield the following with buffer $\text{BUFF}_\varnothing$ and prefix $[v := \text{ping}]_q$.

\[
\begin{align*}
\begin{bmatrix}
\text{skip;} \\
_- := \text{recv}(q)
\end{bmatrix}_p & \parallel \\
\begin{bmatrix}
\text{skip;} \\
\text{send}(p, \text{Ack})
\end{bmatrix}_q
\end{align*}
\]

Applying the congruence $\text{skip;} s \equiv s$ and repeating the same sequence of rules yields the target $\text{Ex1}_{\text{SYNC}}$.

### 4.3.2 Communication between a Single Process and a Group

Next we present our method for rewriting loops over symmetric sets of process identifiers. At a high-level, in order to reason about an unbounded collection of processes, we (1) focus on a single (arbitrarily chosen) process, (2) synchronize the interactions with that process, and (3) generalize to the entire set. This strategy is inspired by techniques such as temporal case splitting from model checking [50], materialization & blurring from shape analysis [65], or equivalently, unfold & fold or focus & adoption for linear type systems [26]. Next, we formalize this strategy in our rewrite rules.
Iterating over symmetric processes. Figure 4.6 shows rule R-LOOP-UPD, which rewrites the interaction between a set of symmetric processes $P$ and a single process $q$ that iterates over $P$. In order to rewrite the entire loop, R-LOOP-UPD requires showing that a single interaction between $q$ and an arbitrary process $u \in P$ can be rewritten. For this, the rule picks a fresh process identifier $u$ and a fresh loop variable $x$. It then adds a permission to communicate with process $u$ (i.e., to “unfold” $u$ from set $P$) to the context $\Gamma$, and strengthens the prefix $\Delta$ by assuming a communication invariant $I_C$. Intuitively, the communication invariant contains information about who to send to and who to receive from, however, we find that this invariant is often very simple. Finally, the rule requires invariant $I_C$ to be inductive. If the rewrite succeeds, the rule extends the synchronous prefix by repeating the synchronization that was computed for a single iteration, once for every process in $P$. Note that the context $\Gamma$ after the rewrite is required to be the same as the initial context. This ensures that all messages sent in a given iteration must also be received in that iteration.

Unfolded processes. Figure 4.6 contains the rules for unfolding process $u$ from a set $P$. Rule R-VAR-UNFOLD binds $u$ to $x$, allowing the process that iterates over $P$ to send or receive from $u$ via its loop variable. Rule R-SYM-UNFOLD unfolds $u$ through a symmetric receive, i.e., it transforms a receive from an arbitrary process in $P$ to a receive from $u$. Note that both rules consume the unfold permission, i.e., only a single process can be unfolded in a given iteration.

Example 4.2. We now rewrite $\text{Ex2}$ into its synchronization $\text{Ex2}_{\text{SYNC}}$ starting from the initial context $\Gamma_0 \triangleq (\text{BUFF}_q, \emptyset)$. For brevity, we omit annotation.

\[
\text{Ex2} \triangleq \begin{cases}
\text{for } (q : \text{qs}) \{ \\
\text{send}(q, \text{ping}); \\
\_ := \text{recv}(q); \\
\} \\
\end{cases}
\]

\[
\text{Ex2}_{\text{SYNC}} \triangleq \begin{cases}
\text{for } (q : \text{qs}) \{ \\
\text{v} := \text{recv}(p); \\
\text{send}(p, \text{Ack}); \\
\_ := \text{recv}(q); \\
\} \\
\end{cases}
\]

The rewrite begins with an application of R-LOOP-UPD which picks an arbitrary iteration.
### Figure 4.5. Rewrite Rules (Basic Statements)

<table>
<thead>
<tr>
<th>R-CONTEXT</th>
<th>R-CHOICE</th>
<th>R-FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, \Delta, A \leadsto \Gamma', \Delta', A'$</td>
<td>$\Gamma, \Delta, A \leadsto \Gamma, \Delta; \Delta_A, \text{skip}$</td>
<td>$\Delta \equiv \text{false}$</td>
</tr>
<tr>
<td>$\Gamma, \Delta, A \circ B \leadsto \Gamma', \Delta', A' \circ B$</td>
<td>$\Gamma, \Delta, B \leadsto \Gamma, \Delta; \Delta_B, \text{skip}$</td>
<td>$\Gamma, \Delta, A \leadsto \Gamma, \Delta, \text{skip}$</td>
</tr>
<tr>
<td>$\Gamma, \Delta, A \parallel B \leadsto \Gamma', \Delta', A' \parallel B$</td>
<td>$\Gamma, \Delta, A \parallel \Delta_B, \text{skip}$</td>
<td></td>
</tr>
</tbody>
</table>

**R-SEND**

$\Delta := x = q$

- $q$ is a PID
- $\Gamma' = \Gamma[(p, q, t) \leftarrow m \cdot n]$  
  $p$ is a PID
- $\Gamma(p, q, t) = m \cdot n$
  $\Delta' = \Delta; \{y := p, m\}_q$

- $\Gamma, \Delta, [\text{send}(t, x, n)]_p \leadsto \Gamma', \Delta, \text{skip}$

**R-RECV**

$\Delta := x = p$

- $\Gamma' = \Gamma[(p, q, t) \leftarrow m \cdot n]$  
  $p$ is a PID
- $\Gamma(p, q, t) = m \cdot n$
  $\Delta' = \Delta; \{y := p, m\}_q$

- $\Gamma, \Delta, [\text{recv}(x, t)]_q \leadsto \Gamma', \Delta', \text{skip}$

**R-LOOP-UPD**

$x, u$ fresh

- $\Gamma_0 \equiv \Gamma \cup \{\text{unfold}(u, x, P)\}$ and $\Delta_0 \equiv \text{assume}(I_C)$
- $\Delta' \equiv \Delta$; $\Delta_u[p/u]$  
  $\{\text{for } (p : P) \{I_C}\}$
- $\Gamma_0, \Delta_0, [A]_u \parallel B[x/p] \leadsto \Gamma, (\Delta_0; \Delta_u), \text{skip}$

**R-VAR-UNFOLD**

- $\Gamma \vdash \text{unfold}(u, x, P)$
- $\Gamma' \equiv \Gamma \setminus \{\text{unfold}(u, x, P)\}$

- $\Gamma, \Delta, A \leadsto \Gamma', \Delta \cup \{\text{assume}(x = u)\}, A$

**R-SYM-UNFOLD**

- $\Gamma \vdash \text{unfold}(u, x, P)$
- $\Gamma' \equiv \Gamma \setminus \{\text{unfold}(u, x, P)\}$

- $\Gamma, \Delta, y := \text{recv}(P, t) \leadsto \Gamma', \Delta, y := \text{recv}(u, t)$

### Figure 4.6. Rewrite Rules (Iteration over sets of processes and unfolding)
of the loop, generating the rewrite obligation \( E_{X^2U} \) shown below.

\[
E_{X^2U} = \begin{cases}
\text{send}(x, \text{ping}); & v := \text{recv}(p); \\
\_ := \text{recv}(x) & p \text{ send}(p, \text{Ack})
\end{cases}
\]

As the new obligation occurs in the context \( \Gamma \cup \{\text{unfold}(u, x, Q)\} \), the rewrite consumes \( \{\text{unfold}(u, x, Q)\} \) to produce \( E_{X^2U}' \) through an application of R-VAR-UNFOLD.

\[
E_{X^2U}' = \begin{cases}
\text{assume}(x = u); & v := \text{recv}(p); \\
\text{send}(x, \text{ping}); & p \text{ send}(p, \text{Ack})
\end{cases}
\]

The rewrite of the body does not depend on any loop-carried state, so the invariant \( I_C \triangleq \text{true} \) allows to rewrite the program to skip with prefix \( ([v := \text{ping}]_u; [\_ := \text{Ack}]_p) \).

This lets us rewrite \( E_2 \) to skip with the prefix \( E_{SYNC} \).

**Example 4.3.** Next we consider a program \( E_{BAD} \) which our rules fail to rewrite. In \( E_{BAD} \), each iteration of \( p \)'s for-loop can receive from an arbitrary process in \( qs \). However, \( p \) also sends to the process bound by its loop variable causing it to potentially communicate with two different processes in the same iteration. As this breaks the symmetry between the processes in \( qs \), it is impossible to rewrite \( E_{BAD}^1 \) into a synchronous for-loop. Applying R-LOOP-UPD generates the obligation to rewrite \( E_{BAD}^1 \) to skip. Applying R-SYM-UNFOLD to \( E_{BAD}^1 \) yields \( E_{BAD}^2 \) and consumes the permission \( \text{unfold}(u, x, qs) \). At this point, rewriting is doomed: applying R-SEND and R-RECV yields the program

\[
[send(x, x);]_p \parallel [\_ := \text{recv};]_u
\]
which cannot be further rewritten, as the value of $x$ is unknown.

\[
\text{EX}_{BAD} \triangleq \begin{cases}
\text{for } (q : qs) \{ \\
\quad \_ := \text{recv}(qs) ; \\
\quad \text{send}(q, q)
\} \\
\end{cases}
\phi

\prod_{q : qs} \left[ \begin{array}{c}
\text{send}(p, \text{ping}) ; \\
\quad \_ := \text{recv}(p)
\end{array} \right]_q
\]

\[
\text{EX}^1_{BAD} \triangleq \left[ \begin{array}{c}
\_ := \text{recv}(qs) ; \\
\text{send}(x, x)
\end{array} \right]_p \parallel \left[ \begin{array}{c}
\text{send}(p, \text{ping}) ; \\
\_ := \text{recv}(p)
\end{array} \right]_u
\]

\[
\text{EX}^2_{BAD} \triangleq \left[ \begin{array}{c}
\_ := \text{recv}(u) ; \\
\text{send}(x, x)
\end{array} \right]_p \parallel \left[ \begin{array}{c}
\_ := \text{recv}
\end{array} \right]_u
\]

### 4.3.3 Communication between Groups of Processes

Finally, we turn to the rules for rewriting pairs of symmetric groups of processes. The addition of these rules marks a crucial increase to the expressiveness of the system described in Chapter 3.

**Set to set.** The rule R-SYM-RMV handles rewriting the parallel composition of two sets of PIDs, $P$ and $Q$. Analogous to R-LOOP-UPD, the rule works by (a) focusing on an element $u \in P$, (b) rewriting the interaction of $u$ with the members of $Q$, and (c) generalizing the interactions to all of the members of $P$. As before, the rule picks a fresh pid $u$ and strengthens the prefix $\Delta$ through a communication invariant $I_C$ which is required to be inductive. Note that the rule substitutes the singleton set $\{u\}$ for $P$ causing all loops in process $q$ to iterate over $u$, only. If the rewrite succeeds, the rule extends the synchronous prefix by repeating the synchronization $\Delta^u$ in parallel, once for each $u \in P$.

**Atomic actions.** The rule R-FOR-ATOMIC complements R-SYM-RMV: given a loop over a set that has been focused on a single element, R-FOR-ATOMIC generates the permission to use element by unfolding it. For this, the rule rewrites the loop body to produce the synchronization $\Delta^u$. As unfolding ensures that the rewrite is restricted to the interaction between the process pair $u$ and $q$, we know that the resulting interaction can be executed atomically and satisfies loop invariant $I_s$. 
Note that this is only sound if there are no control-flow dependencies on the loop iteration variable, $p$. The judgment $\vdash A :: cf(p)$ ensures that this is the case. The rules for determining $\vdash A :: cf(p)$ are given in Figure 4.7, and implement a simple taint analysis.

The combination of R-SYM-RMV and R-FOR-ATOMIC thus produces concurrent programs where message passing has been replaced with coarse-grained atomic actions.

**Example 4.4.** We now examine Ex4.

First we apply R-SYM-RMV, whose precondition obliges us to rewrite the following to skip:

$$\begin{align*}
\prod_{p:ps} & \left[\begin{array}{l}
\text{for (q:qs) { }
  \text{send(q, (p, ping)); }
  \_ := recv(q)
\end{array}\right] \\
\| & \prod_{q:qs} \left[\begin{array}{l}
\text{for (p:ps) { }
  (id, v) := recv(ps); }
  \text{send(id, Ack)}
\end{array}\right]
\end{align*}$$

$$\begin{align*}
\prod_{p:ps} & \left[\begin{array}{l}
\text{for (q:qs) { }
  \text{send(q, (p', ping)); }
  \_ := recv(q)
\end{array}\right] \\
\| & \prod_{q:qs} \left[\begin{array}{l}
\text{for (p: {p'}) { }
  (id, v) := recv(ps); }
  \text{send(p, Ack)}
\end{array}\right]
\end{align*}$$
Applying R-LOOP-UPD and R-VAR-UNFOLD transforms the obligation to

\[
\begin{align*}
\text{for (p: \{p^\mu\}) { } } \\
&\quad (id, v) := \text{recv}(\{p^\mu\}); \\
&\quad \text{send}(id, \text{Ack})
\end{align*}
\]

The obligation opened by R-SYM-RMV is closed by R-FOR-ATOMIC and R-SYM-UNFOLD, yielding

\[
\begin{align*}
\text{for (p: \{p^\mu\}) { } } \\
&\quad (id, v) := \text{recv}(p^\mu); \\
&\quad \text{send}(id, \text{Ack})
\end{align*}
\]

which is rewritten to skip. The prefix computed by R-FOR-ATOMIC is thus the atomic action

\[
\Delta_u(p^\mu, q^\mu) \equiv ([v := \text{ping}]_{q^\mu}; [\_ := \text{Ack}]_{p^\mu}).
\]

Propagating \(\Delta_u\) back through the applications of R-LOOP-UPD and R-SYM-RMV finally yields the rewritten program

\[
\prod_{p,p_s} \text{for (q: qs) { } } ([v := \text{ping}]_{q}; [\_ := \text{Ack}]_{p}).
\]

4.3.4 Rewrite Rule Correctness

**Interpretations.** We interpret prefixes and contexts such that \((\sigma, \mu) \in [\Delta, \Gamma]\) when \(\sigma\) and \(\mu\) are a store and buffer consistent with the states reachable by executing \(\Delta\) and the assumptions in \(\Gamma\).

The soundness theorem says that each rewriting step preserves the reachability of halted states. Let \(C \approx C'\) when \(C = C' = \text{crash}\) or \(C\) and \(C'\) have equal stores.

**Theorem 4.1** (Soundness). *Let \(A\) contain only almost-symmetric races, and assume*
Figure 4.8. Rewrite Rules (case splitting on a single process)

1. $\varnothing, \text{skip}, A \rightarrow \Gamma_B, \Delta_B, B$

2. $C_A \in \text{skip,} \varnothing, A$ and $C_A \rightarrow_{I_S} C_H$ where $C_H \in \text{Halted}(C_A)$ (is halted).

Then there exists $C_B \in \text{[\Delta_B,} \Gamma_B, B]$ and $C'_H$ such that $C_B \rightarrow_{I_S} C'_H$ and $C_H \approx C'_H$.

Proof Sketch. We provide the intuition for why the rewrite rules are sound, but do not present a detailed proof. Many of the rules reduce to a special case of the rules for computing canonical sequentializations, so we direct our attention to the new features, namely the rules R-SYM-RMV and R-FOR-ATOMIC. The proof is by induction on the rewriting derivation.

• Case R-SYM-RMV:

To prove R-SYM-RMV, we strengthen the inductive hypothesis to include an environment that can execute behaviors on behalf of processes in $P$ - the constraint
is that these behaviors must preserve $I_C$. The conclusion follows by induction on the size of $P$. The rewritten $\Delta$ is collected for each $p \in P$ and composed in parallel with the input environment of the inductive hypothesis. Thus, the output rewritten program is the parallel composition over $\Delta$.

- **Case R-FOR-ATOMIC:**
  As R-SYM-RMV case splits on a processes $u$, the key property to show is that the communication invariant $I_C$ summarizes the behavior of statements that interact with processes $p \in P$ when $p \neq u$. In this case, we consider the entire for loop rather than just the iteration corresponding to $u$. From the rewriting assumption, we can prove that, if the environment maintains $I_C$, then the halting states of $\left[ \text{for } (p) \{ I \} \parallel [B] \right]_q$ are contained in $\left[ \text{for } (p' \setminus \{ u \}) \{ I \} \right]_q \parallel \langle \Delta^u \rangle$. This is justified by the fact that the rewriting hypothesis guarantees that each iteration of the loop terminates, and that the environment can only interrupt (i.e., havoc the state) between iterations.

## 4.3.5 Extensions

We now discuss three extensions to the rules presented above that are implemented in CHAI.

**Grouping Statements.** Sometimes there is a choice of which for-loop to merge a local action with. For instance, $p'$'s conditional statement in Figure 4.1 can be merged either with $c'$’s first or second for-loop. To make verification more predictable, our system contains a statement $\text{group}(A)$ that ensures that all statements in $A$ are rewritten together.

**Nonblocking receives.** Our implementation of nonblocking receives follows the description in Ex5 of Section 4.1. The rule for $t \leftarrow \text{newType}$ binds $t$ to a fresh identifier. The rule for $\text{recvT0}$ rewrites $\text{recvT0}$ as if it were a blocking receive and then wraps the resulting prefix into a conditional, non-deterministically either assigning the received
value or Nothing.

**General While Loops.** Our system also handles while loops and iteration over sets of indices. We consider while loops that implement a reactive pattern, where processes serve requests in a non-terminating loop. In this setting, our rules rewrite the finite interaction between a set of clients and the reactive servers. For while loops, we can relax our condition on races in the following way: if a receive occurs in a while loop, we can allow races between sends in the same symmetric set (for an example, consider the acceptor process in Section 4.5.2). Intuitively, this is possible since while loops in our setting do not terminate: we again employ rely-guarantee style reasoning to ensure that, after one client interacts with the server, the same interaction can be repeated by another client. Finally, we note that our system handles purely local computations. In particular, this includes general while loops that perform only local computations, i.e., do not send or receive messages.

### 4.4 Generating Verification Conditions

Given a program $A$ and a property $\varphi$, we can verify that $A \models_H \varphi$ by first rewriting $A$ to produce the synchronized prefix $\Delta$, and then verifying that $\Delta; \text{assert}(\varphi) \Rightarrow_{\text{crash}}$. Let $\varphi$ be a formula over a subset of the processes of $A$ and assume that $A$ can be rewritten to $\Delta$, i.e., $\text{assert}(\varphi) \Rightarrow_{\text{crash}}$. Then, by Theorem 4.1, the original program is safe (i.e.,
Fig. 4.10. Weakest precondition computation. \( V(A) \) denotes the variables that may be assigned in \( A \).

\[
\begin{align*}
wp(x_p := e, Q) & \triangleq Q[x[p \leftarrow e]/x] \\
wp(\text{assume}(P), Q) & \triangleq P \Rightarrow Q \\
wp(\text{assert}(P), Q) & \triangleq P \land Q \\
wp(\langle a \triangleright P \rangle, Q) & \triangleq P \land wp(a, Q) \\
wp(s_1; s_2, Q) & \triangleq wp(s_1, wp(s_2, Q)) \\
wp(s_1 \boxdot s_2, Q) & \triangleq wp(s_1, Q) \land wp(s_2, Q) \\
wp(\text{for } (x : X)_{\{A, I\}} \{ A \}, Q) & \triangleq I[\emptyset/d] \\
& \land (\forall y. I[X'/d] \land x \in X' - d \Rightarrow wp(A, I[(d \cup \{x\})/d])) \\
& \land (\forall y. (I[X/d] \Rightarrow Q) \\
\text{where } y & \triangleq V(A) \\
wp(\bigcap_{p \in P} A, Q) & \triangleq ((\forall p \in P. \text{pc}_p[p] = l_0) \Rightarrow I) \\
& \land \forall p_0 \in P. \forall y. \\
& \quad \land (I \land CF(I) \Rightarrow wp(a[p_0/p], I[\text{pc}'_p/\text{pc}_p])) \quad \forall_{a \in A} \\
& \land \forall y. ((\forall p \in P. \text{pc}_p[p] = l_{\text{exit}}) \land I \Rightarrow Q) \\
\text{where } I & \triangleq \forall p \in P. \\
& \quad \land \text{pc}_p[p] = I \Rightarrow \bigvee_{(a \triangleright l') \in \text{pred}(A,l)} B_{l'} \\
CF(I) & \triangleq \text{pc}_p[p] = I \\
& \quad \land \bigvee_{l' \in \text{out}(A,l)} \text{pc}'_p = \text{pc}_p[p \leftarrow l'] \\
y & \triangleq \{ \text{pc}_p, \text{pc}'_p \} \cup V(A)
\end{align*}
\]
\[ x[p \leftarrow e] \text{ in } Q. \]

**Loops.** Determining the weakest precondition of a loop requires a user-supplied loop invariant. The for (\( x : X \)) \( \{ \ldots \} \) case uses a user-supplied invariant parameterized by \( d \), an auxiliary variable that contains the elements of \( X \) that have already been processed by the loop (i.e., are “done”).

**Parallel Composition.** We define the case for parallel processes \( \bigcap_{p \in P} A \) as an analog of the standard case for loops. First, let the function \( \text{actions}(A) \) collect the labeled atomic actions of \( A \). Next, we define an auxiliary variable \( pc_p \) (the program counter) that maps each process \( p \) in \( P \) to its current program location (i.e., the current atomic block). The final component is an inductive invariant. The invariant is constructed from the annotations on the actions in \( \text{actions}(A) \): if \( \langle a \triangleright B \rangle l \in \text{actions}(A) \), then the invariant includes the conjunct \( pc_p[p] = l \Rightarrow B \) where \( B \) is the disjunction of all annotations for control locations that can transition to \( l \), i.e., all \( B_m \) such that \( \langle a \triangleright B \rangle m \in \text{actions}(A) \) and \( l \) is one of the immediate successors of \( m \) in the control flow of \( A \).

We define \( \text{wp}(\bigcap_{p \in P} A, Q) \) as three main assertions, requiring that (1) the invariant \( I \) holds in the initial state; (2) each transition preserves the invariant (i.e., the invariant is inductive); and (3) when every process in \( P \) has terminated (i.e., has its program counter set to \( l_{\text{exit}} \)), \( I \) implies \( Q \).

**Example 4.5.** Consider the program \( A \).

\[
A \triangleq \bigcap_{p \in Ps} \left\{ \begin{array}{l}
\text{for } (q : qs)_{\text{true, true}} \{ \\
\quad \text{send}(q, (q, \text{ping})) \\
\} \\
\text{for } (q : qs)_{\text{true, true}} \{ \\
\quad x := \text{recv}(qs); \\
\quad \text{assert}(x = \text{ping})
\} \\
\end{array} \right\}_p \\
\bigcap_{q \in Qs} \left\{ \begin{array}{l}
\text{for } (p : ps)_{\text{true, true}} \{ \\
\quad (id, y) := \text{recv}(ps); \\
\quad \text{send}(id, y);
\} \\
\end{array} \right\}_q
\]

\[ S \triangleq \lambda d. q \in d \Rightarrow q.y = \text{ping} \]
We want to prove that $A$ is safe, i.e., that $A \Rightarrow_1 \text{crash}$. To do so, we rewrite $A$ to $A_{\text{SYNC}}$, and then check the validity of $\text{wp}(A_{\text{SYNC}}; \text{assert(true)}, \text{true})$.

\[
A_{\text{SYNC}} \triangleq \bigcap_{q \in q_s} \begin{cases} 
\text{for } (p : ps) \{ 
\left[\{(q.id, q.y) \leftarrow (q, \text{ping})\}_q \triangleright S \{q\}_l_0; 
\begin{array}{l}
\text{p.x} := q.y; \\
\text{assert(p.x = ping)} 
\end{array}
\right]_l_1
\} \\
\}
\end{cases}
\]

The invariant for $A_{\text{SYNC}}$ is derived from its two atomic actions. Because $l_1$ is reachable from $l_0$, the annotation from $l_0$ is used as the assertion when some $q$ is at $l_1$, reflected in the conjunct $\text{pc}_{q_s}[q] = l_1 \Rightarrow S$. Since $l_0$ and $l_{\text{exit}}$ are reachable from $l_1$, whose annotation is true, the assertion at $l_0$ is simplified to true. The invariant $I$ is thus defined $I \triangleq \forall q \in q_s. \text{pc}_{q_s}[q] = l_1 \Rightarrow y[q] = \text{ping}$. The weakest precondition comprises three assertions: (1) the initial state satisfies $I$, (2) the two inductive steps corresponding to whether $q$ runs $l_0$ or $l_1$, and (3) the final state satisfies true (which is trivial). Below, $T_0$ and $T_1$ check that the actions at $l_0$ and $l_1$ each preserve $I$. $\text{CF}_{l_0}$ and $\text{CF}_{l_1}$ encode the control flow by constraining the “current” and “next” program counter. The verification condition is defined below. We omit the assignment to $id$, and variables are implicitly universally
quantified.

\[ \wp(\text{SYNC}, \text{true}) = \left( \forall q \in qs. \text{pc}_{qs}[q] = l_0 \Rightarrow I \right) \land \\
\left( \forall q \in qs. \forall p \in ps. (T_0 \land T_1) \right) \land \\
\left( \forall q \in qs. \land I \land \text{pc}_{qs}[q] = l_{\text{exit}} \Rightarrow \text{true} \right) \]

where

\[
T_0 = (I \land CF_l_0 \land y' = y[q \leftarrow \text{ping}] \Rightarrow I[pc'_{qs} / pc_{qs}][y' / y])
\]

\[
T_1 = (I \land CF_l_1 \land x' = x[p \leftarrow y[q]] \Rightarrow (x'[p] = \text{ping}) \land I[pc'_{qs} / pc_{qs}][x' / x])
\]

\[
CF_l_0 = \text{pc}_{qs}[q] = l_0 \land \text{pc}'_{qs} = \text{pc}_{qs}[q \leftarrow l_1]
\]

\[
CF_l_1 = \text{pc}_{qs}[q] = l_1 \land (\text{pc}'_{qs} = \text{pc}_{qs}[q \leftarrow l_0] \lor \text{pc}'_{qs} = \text{pc}_{qs}[q \leftarrow l_{\text{exit}}])
\]

As this formula is valid, \text{SYNC} \rightarrow_{l_1} \text{crash} and hence \text{SYNC} \rightarrow_{l_1} \text{crash}.

Proposition 4.1 states the correctness of \(\wp(A, \text{true})\) and says that if \(A\) executes in a state satisfying \(\wp(A, \text{true})\), then \(A\) does not reach the crash configuration. The definition of \(\wp(s, Q)\) is standard and we do not prove it sound here.

**Proposition 4.1 (VC generation).** If \(\text{Valid}(\wp(\Delta, \text{true}))\) then \(\Delta \rightarrow_{l_1} \text{crash} \).

In general, even if the formulas asserted and assumed in a program are decidable, the naive model of message buffers as arrays requires nested array reads, yielding undecidable verification conditions. However, when \(\sim, \text{skip}, A \sim \sim, \Delta, B, \Delta\) does not contain message buffers. Hence, if the formulas asserted and assumed in \(A\) are in the array property fragment [8], then the verification resulting from \(\wp(\Delta, \text{true})\) is as well.

**Proposition 4.2 (VC Decidability).** Assume \(\sim, \text{skip}, P \sim \sim, \Delta, Q, \psi = \wp(\Delta, \text{true}), \) and the local states of \(P\) do not contain nested reads. Then, checking the validity of \(\psi\) is decidable.
4.5 Case Studies: Synchronous Invariants for Distributed Protocols

In this section, we describe three case studies where we use CHAI to verify functional correctness properties of complex asynchronous programs.

Implementation. We have implemented our approach of pretending synchrony in a tool, CHAI. CHAI takes as input a IS program \( A \), a property \( \varphi \) and (1) computes a synchronization \( \Delta \), and, should one exist, (2) verifies that \( \Delta \) satisfies \( \varphi \), thereby proving the safety of \( A \) (Section 4.4), i.e., that \( A \models H \varphi \). We implemented the rewriting step by interpreting our rewriting rules (Section 4.3) as a PROLOG predicate. The verification step is implemented by computing the weakest precondition of the rewritten program (Section 4.4) and then using Z3 to prove the validity of the verification condition.

Experiments. We used CHAI to verify the following case studies implemented in IS:

- **Two-Phase Commit** (as discussed in 4.1). We verify that if the transaction is committed then each participant has accepted the coordinator’s proposal.

- **Key-Value Store**, a database implementing a write-once key/value store. We verify that if a client sets the value of a key, a subsequent read of that key returns the same value.

- **Raft Leader Election** (from [55]). We verify that no two candidates both think they are leader in the same term.

- **Single-Decree Paxos** (from [41]). We verify agreement, that is, if two proposers decide on a value, they must have decided on the same value.

Results. The results of our experiments are summarized in Table 4.1. The table indicates (1) the number of lines of IS required for each benchmark; (2) the number of lines of annotations in the form of loop invariants; (3) the amount of time it took for
Table 4.1. CHAI benchmark results. #LOC is the number of lines of code of Is, #Annot is the number of lines of annotations (loop invariants), Rewrite (s) is the time CHAI took to compute the benchmark’s synchronization, and VC (s) is the time it took to prove the validity of the verification condition output by CHAI. All experiments were run on a 2.3 GHz Intel Core i7 CPU.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#LOC</th>
<th>#Annot</th>
<th>Rewrite (s)</th>
<th>VC (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Value Store</td>
<td>36</td>
<td>4</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Two-Phase Commit</td>
<td>33</td>
<td>4</td>
<td>0.09</td>
<td>0.02</td>
</tr>
<tr>
<td>Raft Leader Election</td>
<td>54</td>
<td>12</td>
<td>0.05</td>
<td>0.03</td>
</tr>
<tr>
<td>Single-Decree Paxos</td>
<td>87</td>
<td>44</td>
<td>1.66</td>
<td>0.39</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>210</strong></td>
<td><strong>64</strong></td>
<td><strong>1.82</strong></td>
<td><strong>0.46</strong></td>
</tr>
</tbody>
</table>

CHAI to compute the synchronization of the benchmark, and (4) the amount of time it took Z3 to prove the verification condition valid.

**Comparison.** Our benchmarks required about 0.25x proof overhead i.e., one line of annotation per four lines of code, with Paxos requiring about one line of annotation per two lines of source. This is in contrast to recent systems like Ironfleet [31] (9x overhead), Verdi [82] (90x overhead on Raft), and DISEL [80] (3x overhead on two-phase commit). A direct comparison to these tools is difficult, as, for one, they implement significantly more featureful systems. Nevertheless, the reduction in annotations is significant, and we conjecture that it is largely due to the automatic reduction computation, i.e., the rewriting step. The rewritten program is synchronous composed of coarse-grained atomic blocks which further reduce the number of program interleavings, simplifying the invariants needed to verify correctness.

4.5.1 Raft Leader Election

Figure 4.11 shows an implementation of the leader election phase of the Raft consensus algorithm [55]. We have removed components like logs and timeouts that are not relevant to leader election, and have omitted the initialization phase for brevity.

**Asynchronous Program.** The protocol contains two sets of symmetric processes:
for \((c \in C)\) \{ 
\begin{align*}
\text{ReqVote} & (t, \text{pid}) := \text{recv}(); \\
\text{if } t > \text{term} & \{ \\
\text{term} & := t; \\
\text{voted} & := \text{false} \\
\} \\
\text{s} & := (t \geq \text{term} \land \text{(voted} \Rightarrow \text{vote} = \text{pid})); \\
\text{if } s & \{ \\
\text{voted} & := \text{true}; \\
\text{vote} & := \text{pid}; \\
\text{votes}[t] & := \text{vote} \\
\} \\
\text{send}(\text{pid}, \text{Vote}(s, \text{term})) & (A) \\
\}
\}
for \((f \in F)\) \{ 
\begin{align*}
\text{send} & (f, \text{ReqVote}(\text{term}, \text{c})); \\
\text{Vote}(s, t) & := \text{recvTO}(f); \\
\text{if } s & \{ \\
\text{count} & := \text{count} + 1 \\
\} & (B) \\
\}
\}
\text{if } 2 \times \text{count} > |F| \{ 
\text{leader} & := \text{true} \\
\}
\}
\text{RAFT} \triangleq \bigcap_{f \in \text{Followers}} F(f) \parallel \bigcap_{c \in \text{Candidates}} C(c)
\]

**Figure 4.11.** Simplified Raft leader election algorithm (asynchronous version)
the set of *candidate* processes \( C \) and the set of *follower* processes \( F \). Each follower is initialized with \( \text{voted} := \text{false} \), and each candidate initialized with \( \text{leader} := \text{false} \) and \( \text{counter} := 0 \). The algorithm starts with candidates sending a message of type \( \text{ReqVote} \) containing the candidate’s current term and its process id to every follower. If the candidate term is larger than the follower’s current term, the follower updates its term and sets its \( \text{voted} \) flag to false. The follower then determines whether or not to accept the vote by setting a flag \( s \). \( s \) is set to true if the candidate term is at least as big as the follower’s term and either the follower did not vote yet, or it voted for the candidate. The follower then updates its bookkeeping and responds back to the candidate with a message of type \( \text{Vote} \) with a flag set to true if the vote is granted, and the current term of the follower, otherwise. After this message exchange, the candidate then claims to be the leader if it receives messages with granted votes from a majority of the followers. In addition to the original protocol, we have added a ghost variable \( \text{votes} \) to keep track of the history of votes that a follower grants.

**Property: Election Safety.** We want to prove the key election safety property [55]: for any given term, *at most one leader is elected* by the followers, i.e., the assertion:

\[
\forall c_1, c_2 \in C. \ c_1.\text{leader} \land c_2.\text{leader} \land c_1.\text{term} = c_2.\text{term} \Rightarrow c_1 = c_2
\]

Upon closely inspecting the follower’s code, we can see that each follower votes for *at most* one candidate for a given term due to the condition \( \text{voted} \Rightarrow \text{vote} = \text{pid} \) in variable \( s \). Unfortunately, this reasoning is not inductive and hence cannot be used to prove election safety. To derive an inductive invariant, we must relate the value of the counter in candidate \( c \) with the number of followers that have voted for \( c \) at some point during the execution. Thus, an asynchronous invariant for the follower’s loop must relate the follower’s local state to that of the candidates, requiring an extensive case split. However, the invariant becomes easy, once we pretend synchrony.

**Invariants.** The property is proven by the following invariant \( I_1 \). \( I_1 \) refers to the
cardinality of two sets: $k[c]$, the number of followers that may vote for $c$ in the future, and $l[c]$, the number of followers that have voted for $c$ in the past. Then, the invariant states that the number of followers who voted for a candidate is as least as big as its counter, and the sum of the cardinalities is bounded by the total number of followers.

$$I_1 \triangleq \forall c \in C. |F| \geq k[c] + l[c] \land c.count \leq l[c]$$

$k[c] \triangleq \# \{ f \mid f.term < c.term \}$  \quad (# may vote for $c$ in future)

$l[c] \triangleq \# \{ f \mid f.term \geq c.term \land f.votes[c.term] = c \}$  \quad (# have voted for $c$ in past)

The synchronized version computed by our method contains a single atomic action built from the bodies of the two loops. In particular, synchronization schedules the block $(A)$ before $(B)$, clarifying the relationship between $f.s$ and $c.count$: $c.count$ is incremented only when $f.s$ is set to true.

**Verification.** The above invariants let us prove election safety as follows. Suppose that there are two different leaders $c_1$ and $c_2$ within the same term. Then, both $c_1.count$ and $c_2.count$ should be greater than $|F|/2$. But, due to $I_1$, it must be that $l[c_1] > |F|/2$ and $l[c_2] > |F|/2$. By cardinality reasoning, there should exist a follower $f^*$ in both $l[c_1]$ and $l[c_2]$, yielding the contradiction:

$$c_1 = f^*.votes[c_1.term] = f^*.votes[c_2.term] = c_2$$

While the proof involves cardinality invariants, the key bit is that once we have eliminated asynchrony — specifically, message buffers — via pretend synchrony, we can completely automate the discovery of such invariants using known techniques like #Π [76].

### 4.5.2 Single-decree Paxos

Figure 4.12 contains the single-decree version of the Paxos consensus protocol [41], where the proposers do not update their proposal number after a failed attempt and try again. For brevity, we omitted the initialization of the local variables. It also does
while true {
    (pid, msg) := recv();
    case msg of
        Proposal(n) →
            if max < n {
                max := n
            }
        Accept(n,val) →
            if max ≤ n {
                vt := n;
                ts := ts ∪ {n}
            }
    end;
    send(pid, (vt, v))
}

\[ P(p) \triangleq \prod_{p: \text{Proposers}} P(p) \]
\[ A(a) \triangleq \prod_{a: \text{Acceptors}} A(a) \]

Figure 4.12. Single-decree Paxos (asynchronous version)
not contain the learner processes that do not participate in the value selection.

**Asynchronous Program.** The protocol consists of a set of proposers $P$ and a set of acceptors $A$. Some of the local variables are initialized as follows: $\max := -\infty$ contains the smallest proposal number to accept in the future;

$v_t := -\infty$ contains the highest proposal number accepted; $v := \bot$ that contains the accepted value; $ts := \{\}$ contains the numbers of all accepted proposals; $t := \ast$ contains the arbitrary and unique proposal number; $x := \ast$ is the value to be proposed; and $x_t := -\infty$ keeps track of the highest numbered proposal accepted by the acceptors.

The protocol starts with proposers sending a Propose message that contains their proposal number. Acceptors update their $\max$ to reject every proposal with smaller number, and respond back with their current values of $v_t$ and $v$. After receiving a response, the proposers update their value with the value of the highest numbered proposal.

The first phase above deviates from the original Paxos protocol where acceptors respond only if the received proposal number is bigger than they have ever seen. However, this can be considered as a performance optimization. In addition, since we model message drops with the $\text{recvT0}$ primitive, we actually support the same behavior.

If a proposer hears back a response from a majority of acceptors, it starts its second phase by sending an Accept message with its proposal number and value to every acceptor. Upon receiving this message, acceptors update their accepted value and proposal number if necessary, and respond back with the same message as before. If the majority of received replies contain the proposer’s own number, the proposer declares that number to be the decided value.

**Property: Agreement.** We want to prove that the agreement property is satisfied in this protocol, i.e., that every proposer agrees on the decided value: $\forall p_1, p_2 \in P. p_1.\text{decided} \land p_2.\text{decided} \Rightarrow p_1.x = p_2.x$.

**Invariants.** The following invariant can be used to satisfy the agreement property: Here, $k$ counts the number of acceptors that have accepted $p$’s proposal before, $l$
counts the ones that *may accept* it in the future, and *m* counts the ones that *will never* accept it. $I_1^m$, the main lemma, states that if $p$ is ready and not outdated (*i.e.*, a majority of acceptors have either accepted its proposal or might accept it later), then, whenever an acceptor accepts a proposal that is at least as high as $p.t$, it must in fact accept $p.x$.

$$I_1 \equiv I_1^4 \land I_1^m \land ...$$

$$I_1^4 \equiv \forall \ p. \ k[p] + l[p] + m[p] = |A|$$

$$I_1^m \equiv \forall a, p. \ p.ready \land k[p] + l[p] \geq \frac{|A|}{2} \land a.v \geq p.t \Rightarrow a.v = p.x$$

$$k[p] \equiv \# \{a \in A | p.t \in a.ts\} \quad (# \text{have accepted})$$

$$l[p] \equiv \# \{a \in A | p.t \notin a.ts \land a.max \leq p.t\} \quad (# \text{may accept})$$

$$m[p] \equiv \# \{a \in A | p.t \notin a.ts \land a.max > p.t\} \quad (# \text{will never accept})$$

**Synchronized Program.** The synchronized program is the parallel composition of, for each proposer, a single *for*-loop that, for each acceptor $a$, executes two atomic actions: one derived from rewriting the block $(A)$ with the body of $a$’s *while* loop, and one similarly derived from block $(B)$.

As with Raft, once the program is synchronized, the sets and cardinalities we used could be found automatically. In addition, note how simple and intuitive the invariants above are with respect to invariants that one might need to formally verify the original asynchronous protocol.

### 4.6 Summary

In this chapter we have described *pretend synchrony*, a new approach to verifying asynchronous distributed systems by soundly treating them as if they were synchronous when verifying their correctness. We have implemented pretend synchrony in a tool called CHAI and shown that by automatically computing an asynchronous program’s synchronization, it is possible to prove challenging protocols such as the Raft leader election algorithm or single decree Paxos correct while reducing annotation burden by an order of magnitude, even compared to previous SMT based methods.
Acknowledgments

Chapter 4, in full, is adapted from the material as it appears in Klaus v. Gleissenthall, Alexander Bakst, Rami Gökhan Kıcı, and Ranjit Jhala. Pretend Synchrony. In submission to the ACM SIGPLAN International Conference on Principles of Programming Languages, POPL 2018. The dissertation author was the primary investigator and author of this paper.
Chapter 5

Conclusion

Distributed systems are large, complicated pieces of software, and maintaining an accurate mental model of their behavior is taxing if not impossible. In the article “How Amazon Web Services Uses Formal Methods,” [53] the authors note that engineers naturally focus on designing the "happy case" for a system, or the processing path in which no errors occur. This is understandable, as the happy case is by far the most common case. That code path must solve the customer’s problem, perform well, make efficient use of resources, and scale with the business – all significant challenges in their own right. When the design for the happy case is done, the engineer then tries to think of "what could go wrong" based on personal experience and that of colleagues and reviewers. The engineer then adds mitigations for these scenarios, prioritized by intuition and perhaps statistics on the probability of occurrence. Almost always, the engineer stops well short of handling "extremely rare" combinations of events, as there are too many such scenarios to imagine.

Biased by motivated reasoning, it is hard for human engineers to exhaustively enumerate the problematic corner cases.

In this dissertation, we have proposed a method of simplifying proofs of correctness that appeal to the “happy case.” To do so, we have made three distinct contributions to ease the task of message-passing and distributed system verification.

**Symmetric non-determinism.** Symmetric non-determinism is intended to capture a “goodness of design” property that allows developers to reason about their systems as if they were synchronous. The essence of the property is that well-designed systems are agnostic of the underlying asynchrony of the programming model. One way
to understand what “well-designed” means is that it is easy to match program locations across processes that are (implicitly) related by the implemented protocol while lacking any formal annotation. This structure becomes even more apparent in programming languages with expressive type systems: each step (and, hence, message) in the protocol is associated with a distinct type. Matching complementary parts of a message exchange, when faced with many choices, amounts to matching the types of the send and received messages. While not all systems are symmetrically non-deterministic, our work suggests that well-known algorithms are or can be refactored as such.

**Canonical sequentialization.** Symmetric non-determinism allows us to find proofs of some shallow properties, like deadlock freedom, completely automatically. Moreover, the algorithm we have presented for computing sequentializations is fast enough to incorporate into development tools. We conjecture that relating a distributed program to its sequentialization further aids the developer in understanding her code during development.

**Pretend synchrony.** Finally, we have applied symmetric non-determinism to reason about program that do not have sequentializations. Nevertheless, we argue that most verification approaches conflate reasoning about the particulars of the messaging subsystem and the data exchanged by the messages. By pretending synchrony, we completely eliminate asynchronous message buffers, which connects a particular implementation to an idealized, synchronized algorithm with coarse grained concurrency. Our case studies proving the correctness of well-known distributed systems suggests this amounts to a significant savings in proof effort.

### 5.1 Future Directions

Designing distributed systems is likely to remain a challenging problem. However, we believe the techniques described in this dissertation open the door to empowering developers with better programming environments. Necessarily, the automated techniques in this dissertation are incomplete, in the sense that there are correct programs
that can not be verified as such by the presented proof systems. We conjecture that some of these shortcomings may be addressed by somewhat straightforward extensions. For example, we do not discuss programs with dynamic process creation, but it is possible that these can be modeled as programs with a set of *static* (but unbounded) groups of processes that receive a distinguished “initialization” message to model creation.

**Parameterized topologies.** A class of problems with a murkier set of potential solutions are systems *lacking* symmetric processes in the sense used throughout the dissertation. Notable examples include protocols that overlay a *ring* structure on the participants. In this setting, the topology of the protocol, not merely the set of participants, is parameterized. Reasoning about these protocols might require reasoning about *ordsets* of processes [], for example in the recursive lookup procedure of Chord. A major limitation of the work presented in this dissertation is the minimal treatment of network anomalies.

**Failures.** While the work applies in a limited capacity to models in which messages can be dropped, we do not consider Byzantine failures. A fruitful research direction would be to use the ideas developed in this dissertation as a foundation for layering additional reasoning targeted at more interesting failure modes.

**Effectiveness.** We argued that the approach of synchronization as presented Chapter 4 decreases the proof burden. As with any verification framework, a useful question to ask is if this manifests when actual engineers (rather than motivated PhD students) are given the tool in question. However, if it is indeed the case that pretending synchrony eases reasoning about distributed systems, then it should be possible to leverage tools for reasoning about shared memory systems to further automated proofs. It would be interesting to see if, for example, automated verification tools such as #Π [76] could be used to make certain kinds of correctness proofs completely automatic.

**IDE integration & synthesis.** Finally, sequentializations and synchronizations have the honor of being easy and fast to compute (when they exist). Because they can be used to provide rapid feedback, it would be valuable to study to what extent they
aid developers in designing distributed applications. One could imagine an IDE that presents the developer with the current sequentialization, if it exists, or a set of send and receive statements that are unpaired. Moreover, we showed an implementation of canonical sequentialization that treats the sequential prefix as the unification variable in a Prolog predicate and “runs” the rewrite relation to calculate the sequentialization. An alternative idea is to ask the user to supply the sequentialization and leave the input program unknown, thus yielding a synthesis algorithm for distributed systems.
Appendix A

Translation from $\lambda_m$ to $I_S$

Lemma 3.1. Assume $C \rightarrow_{I_S} C'$ Then, for all $\langle \mu, \mathcal{P} \rangle \in [C]_\gamma$, there exists $\langle \mu', \mathcal{P}' \rangle \in [C']_\gamma$ such that $\langle \mu, \mathcal{P} \rangle \rightarrow_{\lambda_m} \langle \mu', \mathcal{P}' \rangle$.

Proof. The proof is by induction on the length of the derivation of $\rightarrow_{I_S}$, splitting cases on the first step.

- Case Receive:
  
  By assumption,
  
  $$\mu[(p,q,t)] = v \cdot v' \quad \text{(A.1)}$$
  
  $$\sigma, \mu[\text{recv}(t,w)] \xrightarrow{\text{recv}(t,p)\gamma} \sigma, \mu[(p,q,t) \leftarrow v'], [\gamma(q) \equiv v]_q \quad \text{(A.2)}$$

  By the definition of $[\cdot]_\gamma$,
  
  $$[\sigma, \mu[\text{recv}(t,w)]_q]_\gamma = (\mu, \text{recv}_q) \quad \text{(A.3)}$$
  
  $$[\sigma, \mu[(p,q,t) \leftarrow v'], [\gamma(q) \equiv v]_q]_\gamma = (\mu[(p,q,t) \leftarrow v'], (\text{return @} t v)_q) \quad \text{(A.4)}$$

  By the definition of $\rightarrow_{\lambda_m}$ and Equations (A.3) and (A.4),
  
  $$[\sigma, \mu[\text{recv}(t,w)]_q]_\gamma \rightarrow_{\lambda_m} [\sigma, \mu[(p,q,t) \leftarrow v'], [\gamma(q) \equiv v]_q]_\gamma \quad \text{(A.5)}$$

  and the inductive hypothesis concludes the proof.
• Case Sequence:
  
  By assumption,
  
  \[ (\sigma, \mu, A) \xrightarrow{T} I_\text{is} (\sigma', \mu', \text{skip}) \]  
  \[ (A.6) \]

  There are three cases for each singleton process:

  1. Case \( A = \mathcal{L}(f, x, m) \) and \( B = \text{while true} \{ \mathcal{L}(f, x, e) \} \):

     Then, by definition

     \[ \langle \mu, m[\text{rec } f \cdot (\lambda x. e)/f]\rangle \rightarrow_{\lambda m} \langle \mu', (\text{rec } f \cdot (\lambda x. e)) \sigma(x) \rangle \]  
     \[ (A.7) \]

     and unfolding \( f \) yields the conclusion, since \( \rightarrow_f m \).

  2. Case Otherwise:

     There are again two cases to consider for \( \langle \mu, \text{Close}(\sigma, \gamma, \mathcal{P}) \rangle \in [(\sigma, \mu, A; B)]_{\gamma} \)

     (a) Case \( \mathcal{P} = m \gg= \lambda x. n \):

     By the inductive hypothesis,

     \[ \langle \mu, \text{Close}(\sigma, \gamma[p \leftarrow x], m) \rangle \rightarrow_{\lambda m} \langle \mu', \text{Close}(\sigma', \gamma[p \leftarrow x], \sigma'(p, x)) \rangle \]  
     \[ (A.8) \]

     And hence by contextual evaluation,

     \[ \langle \mu, \text{Close}(\sigma, \gamma, (m \gg= \lambda x. n)_p) \rangle \rightarrow_{\lambda m} \langle \mu, (\text{Close}(\sigma', \gamma, n)[\sigma'(x)/x])_p \rangle \]  
     \[ (A.9) \]

     (b) Case \( \mathcal{P} = \text{let } y' = m \text{ in } n \):

     Then (since we assume no variable shadowing)

     \[ \langle \mu, \text{Close}(\sigma, \gamma, \text{let } y' = m \text{ in } n) \rangle \]

     \[ = \langle \mu, \text{let } y' = \text{Close}(\sigma, \gamma, m) \text{ in } \text{Close}(\sigma, \gamma, n) \rangle \]  
     \[ (A.10) \]
and

\[ \langle \mu, \text{let } y^t = \text{Close}(\sigma, \gamma, m) \text{ in } \text{Close}(\sigma, \gamma, n) \rangle \]

\[ \rightarrow_{\lambda_m} \langle \mu, \text{Close}(\sigma, \gamma, n)[\text{Close}(\sigma, \gamma, m)/y] \rangle \quad (A.11) \]

\[ \langle \mu, \text{Close}(\sigma, \gamma, \text{let } y^t = m \text{ in } n) \rangle \rightarrow_{\lambda_m} \langle \mu, \text{Close}(\sigma, \gamma, \text{let } y^t = m \text{ in } n) \rangle \]

(A.12)

i. Case \( t \) is pure:

Therefore,

\[ A = [y := \text{Apply}(\sigma, p, m)]_p \quad (A.13) \]

\[ B = T(n, \gamma(p), p) \quad (A.14) \]

And thus

\[ (\sigma, \mu, A; B) \rightarrow_{1_\sigma} (\sigma[(p, y) \leftarrow \text{Apply}(\sigma, p, m)], \mu, B) \quad (A.15) \]

Finally, by definition,

\[ \langle \mu, \text{Close}(\sigma, \gamma, \text{let } y^t = m \text{ in } n) \rangle \in [(\sigma[(p, y) \leftarrow \text{Apply}(\sigma, p, m)], \mu, B)]_\gamma \]

(A.16)

ii. Otherwise:

The case is identical to the case for \( m >>= n \) after the substitution.

- Case While-Expand:
By assumption,

\[(\sigma, \mu, \text{while true } \{ A \}) \xrightarrow{\epsilon} I_s (\sigma', \mu', A; \text{while true } \{ A \})\] (A.17)

By definition,

\[L(f, x, f \, x) = \text{skip}\] (A.18)

By definition, \[\text{let } \gamma = \lambda x. \text{Close}(\sigma \setminus x, \gamma, e).\]

By the definition of \[\text{let } \gamma\text{ and Equation (A.18), for some } e, f, x, \text{ and } t,

\[\lfloor (\sigma, \mu, \text{while true } \{ A \}) \rfloor_\gamma = \langle \mu, ((\text{rec } f. (\lambda x. \text{Close}(\sigma \setminus x, \gamma, e))) \sigma(x))_p \rangle\] (A.20)

By the definition of \[\lfloor \cdot \rfloor_\gamma,

\[\lfloor (\sigma, \mu, A; \text{while true } \{ A \}) \rfloor_\gamma = \langle \mu, \text{Close}(\sigma, \gamma, e[\text{rec } f. (\lambda x. e)/f]) \rangle\] (A.21)

By the definition of \[\rightarrow_\beta\text{ (to unfold the recursive application) and hence } \rightarrow_{\lambda_m}, \text{ and Equations (A.20) and (A.21),

\[\lfloor (\sigma, \mu, \text{while true } \{ A \}) \rfloor_\gamma \rightarrow_{\lambda_m} \lfloor (\sigma, \mu, A; \text{while true } \{ A \}) \rfloor_\gamma\] (A.22)

The remaining cases are proved by similar expansions of the definition. □

**Lemma 3.2.** For all well-formed \(E\) and \(\sigma\), if \(T(E, \gamma)\) is defined, then there exists \(E''\) such that \(T(E'', \gamma)\) is defined, \(\text{Close}(\sigma', \gamma, E'') = E'\) and

\[\langle \mu, \text{Close}(\sigma, \gamma, E) \rangle \rightarrow_{\lambda_m} \langle \mu', E' \rangle \Rightarrow (\sigma, \mu, T(E, \gamma)) \rightarrow I_s (\sigma', \mu', T(E'', \gamma))\]

**Proof.** By induction on the length of the transition sequence, splitting cases on the first
step.

- Case \( E = H[\text{recv } @ t]_p \):

  By definition,

  \[
  \langle \mu[(q, p, t) \leftarrow v \cdot vs], \text{Close}(\sigma, \gamma, H[\text{recv } @ t]_p) \rangle \rightarrow_{\lambda_m} \langle \mu, H[\text{return } @ t \ v]_p \rangle \quad (A.23)
  \]

  There are two cases based on the shape of \( H \):

  1. Case \( []_p \parallel P \):

     Then by definition,

     \[
     T(E, \gamma) = [\gamma(p) := \text{recv}(t, *)]_p \parallel A \quad (A.24)
     \]

     and thus

     \[
     (\sigma, \mu[(q, p, t) \leftarrow v \cdot vs], [\gamma(p) := \text{recv}(t, *)]_p \parallel A) \rightarrow_{1_u} (\sigma, \mu, [\gamma(p) := v]_p \parallel A)
     \]

     Let \( E'' = (\text{return } @ t \ v)_p \parallel P \) concludes the proof, as by definition,

     \[
     [\gamma(p) := v]_p = T(\text{return } @ t \ v, \gamma(p), p) = \text{Close}(\sigma, p, \text{return } @ t \ v) = \text{return } @ t \ v
     \]

     which proves the case.

  2. Case \( []_p \gg= \lambda x. m \):

     Then by definition,

     \[
     T(E, \gamma) = [x := \text{recv}(t, *)]_p ; T(m, \gamma(p), p) \parallel A \quad (A.26)
     \]
and thus

\[
(\sigma, \mu[(q, p, t) \leftarrow v \cdot vs], [\gamma(p) \coloneqq \text{recv}(t, *)]_p; T(m, \gamma(p), p) \parallel A) \\
\rightarrow_{I_\text{a}} (\sigma, \mu[\gamma(p) \coloneqq v]_p; T(m, \gamma(p), p) \parallel A)
\]

Letting \( E'' = \text{return }@t v \gg= \lambda x. m \parallel P \) concludes the proof as above.

- **Case** \( E = \mathcal{H}[\text{return }@t n \gg= \lambda x. m]_p \):

  Then we have by assumption and the form of \( \rightarrow_{\lambda m} \),

\[
\langle \mu, \text{Close}(\sigma, \mu, \mathcal{H}[\text{return }@t n \gg= \lambda x. m]_p) \rangle \\
\rightarrow_{\lambda m} \langle \mu, \text{Close}(\sigma, \mu, \mathcal{H}[m[n/x]]_p) \rangle
\]

And by definition,

\[
T(\mathcal{H}[\text{return }@t n \gg= \lambda x. m]_p, \gamma) = [x \coloneqq n]_p; T(m, \gamma(p), p); A \parallel B
\]

By the rule for sequencing,

\[
(\sigma, \mu[x \coloneqq n]_p; T(m, \gamma(p), p); A \parallel B) \rightarrow_{I_\text{a}} (\sigma[x \leftarrow \text{Apply}(\sigma, p, n)], \mu, A \parallel B)
\]

It is easy to see that \( m[n/x] = \text{Close}(\sigma[x \leftarrow \text{Apply}(\sigma, p, n)], \gamma, m) \), so letting \( E'' = \mathcal{H}[m]_p \) finishes the case.

- **Case** \( E = \mathcal{H}[(\text{rec } f. \lambda x. m) n]_p \):

  Then let \( e' = m[n/x][((\text{rec } f. \lambda x. m)/f] \) and so by definition,

\[
\langle \mu, \text{Close}(\sigma, \gamma, \mathcal{H}[(\text{rec } f. \lambda x. m)]_p) \rangle \rightarrow_{\lambda m} \langle \mu, \text{Close}(\sigma, \gamma, \mathcal{H}[e']_p) \rangle
\]
Next, by the definition of $\mathcal{T}$,

$$\mathcal{T}(\mathcal{H}[(\text{rec } f. \lambda x. m) \ n], \gamma) = [x := n]_p; \text{while true } \{ \mathcal{L}(f, x, \mathcal{T}(e, x, p)) \}; A \parallel B$$

(A.32)

Since $E$ is well formed, we know that $m$ must have the form, for some $c, a, b, m_0$ where $f$ does not appear in $m_0$.

$$m_0 >>= \lambda r. (\lambda f. (\text{LOOP } f c a b)) \ f$$

and hence $e'$ has form

$$m_0[n/x] >>= \lambda r. (\lambda f. (\text{LOOP } f c a b)) (\text{rec } f. \lambda x. m)$$

so by definition

$$\mathcal{T}(\mathcal{H}[e']_p, \gamma(p), p) = \mathcal{L}(f, x, (\text{LOOP } f c a b)); \text{while true } \{ \mathcal{L}(f, x, \mathcal{T}(m, x, p)) \}$$

(A.33)

and hence by the definition of $(\text{LOOP } f c a b)$ and WHILE-EXPAND,

$$\mathcal{T}(\mathcal{H}[(\text{rec } f. \lambda x. m) \ n], \gamma(p), p) \rightarrow_{\text{I}} \mathcal{T}(\mathcal{H}[e']_p, \gamma(p), p)$$

The remaining cases are similar. 
Appendix B
Sequentialization

**Definition B.1** (Composition of buffers). As an abuse of notation, define $\mu \circ \mu'$ as follows:

$$(\mu \cdot \mu')(p, q, t) = \mu(p, q, t) \cdot \mu'(p, q, t)$$

**Definition B.2** (Interpretation of stores). $\sigma \in [\Delta]_{c_0}$ if and only if $(\sigma_0, \emptyset, \Delta) \rightarrow^* (\sigma, \emptyset, \text{skip})$

**Definition B.3** (Interpretation of contexts). $\mu \in [\Gamma]_{\sigma}$ if and only if $\forall (p,q,t) \in \text{dom}(\Gamma), \mu(p,q,t) = [\Gamma(p,q,t)]_{\sigma}$

**Definition B.4** (Interpretation of stores and contexts). Let $(\sigma, \mu) \in [\Delta, \Gamma]_{c_0}$ if and only if

1. $\sigma \in [\Delta]_{c_0}$
2. $\mu \in [\Gamma]_{\sigma}$
3. $\forall \{x \in X\} \in \Gamma. \sigma(x) \in \sigma(X)$
4. $\forall \{\emptyset \subseteq X \subseteq Y\} \in \Gamma. \emptyset \subseteq \sigma(X) \subseteq \sigma(Y)$

**Definition B.5** (Preorder on stores and buffers).

$$\sigma \leq \sigma' \iff \text{dom}(\sigma) \subseteq \text{dom}(\sigma') \land \forall x \in \text{dom}(\sigma). \sigma'(x) = \sigma(x)$$

$$\mu \leq \mu' \iff \text{dom}(\mu) \subseteq \text{dom}(\mu') \land \forall x \in \text{dom}(\mu). \exists m. \mu'(x) = \mu(x) \cdot m$$
**Definition B.6** (Halted processes). \( p \in \text{hprocs}(\sigma, \mu, P) \) if and only if \( P \equiv [s_p]_p \parallel P_0 \) and any of the following hold:

1. \( s_p = \text{skip} \) or \( s_p = \text{error} \)

2. The next action in \( s_p \) is \( x = \text{recv}(t, y), \sigma(y) = q \), and for all \((\sigma', \mu', P')\) such that \((\sigma, \mu, P) \rightarrow^*(\sigma', \mu', P'), \mu'(q, p, t) = \emptyset \).

3. The next action in \( s_p \) is \( x = \text{recv}(t, S) \) and for all \((\sigma', \mu', P')\) such that \((\sigma, \mu, P) \rightarrow^*(\sigma', \mu', P'), \mu'(q, p, t) = \emptyset \) for all \( q \in S \).

**Definition B.7** (Restriction of program stores and buffers).

\[
\sigma|_p(x_p) = \begin{cases} 
\sigma(x_p) & p \in P \\
\bot & \text{otherwise}
\end{cases}
\]

\[
\mu|_Q(p, q, t) = \begin{cases} 
\mu(p, q, t) & q \in Q \\
\bot & \text{otherwise}
\end{cases}
\]

**Definition B.8** (Preorder on halted states).

\((\sigma, \mu, P) \leq (\sigma', \mu', P') \iff \sigma|_H \leq \sigma'|_H \land \mu|_H \leq \mu'|_H \) where \( H = \text{hprocs}(\sigma, \mu, P) \)

**Definition B.9** (Simulation on states). Define the relation \( \in \) on program states such that \((\sigma, \mu, P) \in (\sigma', \mu', P')\) if and only if whenever

\[(\sigma, \mu, P) \rightarrow^* (\sigma_f, \mu_f, P_f)\]

then there exists \((\sigma'_f, \mu'_f, P'_f)\) such that

\[(\sigma', \mu', P') \rightarrow^* (\sigma_f, \mu_f, P_f) \quad \text{and} \quad (\sigma_f, \mu_f, P_f) \leq (\sigma_f, \mu_f, P_f)\]
Definition B.10. We overload $\in$ to rewriting states as follows.

Let $\Gamma, \Delta, P, \Psi \in \Gamma', \Delta'; P', \Psi'$ if and only if, for all $P_x$ such that $rf(P \otimes P_x)$,

$$\forall (\sigma, \mu) \in [[\Delta, \Gamma]_0]. \exists (\sigma', \mu') \in [[\Delta', \Gamma']_0]. (\sigma, \mu, \Psi_x \otimes \Psi; P \otimes P_x) \in (\sigma', \mu', \Psi_x \otimes \Psi'; P' \otimes P_x)$$

Definition B.11 (Left movers). Let $\lambda$ be a left-mover in $(\sigma, \mu, P \parallel [\lambda; s]_p)$ if and only if

1. If $(\sigma, \mu, P \parallel [\lambda; s]_p) \rightarrow^* (\sigma', \mu', P' \parallel [\lambda; s]_p)$ then $\lambda$ is enabled in $(\sigma', \mu', P' \parallel [\lambda; s]_p)$.
   ($\lambda$ does not become disabled.)

2. If $(\sigma, \mu, P \parallel [\lambda; s]_p) \xrightarrow{[\Gamma]} (\sigma_0, \mu_0, P_0 \parallel [\lambda; s]_p) \xrightarrow{[\lambda]} (\sigma', \mu', P' \parallel [s]_p)$ then there exists $(\sigma_1, \mu_1, P_1)$ such that $(\sigma, \mu, P \parallel [\lambda; s]_p) \xrightarrow{[\Gamma]} (\sigma_1, \mu_1 \parallel [s]_p) \xrightarrow{[\lambda]} (\sigma', \mu', P' \parallel [s]_p)$.
   ($\lambda$ commutes to the left.)

Lemma B.1. If $\sigma \in [\Delta]_\sigma$ and $\sigma' \in [\Delta_0]_{\sigma_0}$ then $\sigma \in [\Delta_0; \Delta]_{\sigma_0}$.

Proof. By the definition of $[\cdot]$.

Lemma B.2 (Send is a left mover). For all $\sigma, \mu, P, p, s$, send$(t, x, m)$ is a left-mover in $(\sigma, \mu, P \parallel [send(t, x, m); s]_p)$.

Proof. The proof is by the definition of left movers.

1. By the definition of $\rightarrow^*$ it is clear that send$(t, x, m)$ in $[send(t, x, m); s]_p$ is always enabled.

2. Let $(\sigma, \mu, P \parallel [send(t, x, m); s]_p) \xrightarrow{[\Gamma]} (\sigma_0, \mu_0, P_0 \parallel [send(t, x, m); s]_p)$. Then

$$(\sigma_0, \mu_0, P_0 \parallel [send(t, x, m); s]_p) \xrightarrow{[send(t, x, m)]} (\sigma', \mu', P_0 \parallel [s]_p)$$

with $\sigma' = \sigma_0$ and $\mu'(p, \sigma(x), t) = \mu_0((p, \sigma(x), t) \leftarrow \mu_0(p, \sigma(x), t) \cdot \sigma(m))$. 

\qed
Lemma B.3. If $\lambda$ is a left mover in $(\sigma, \mu, P \parallel [\lambda; s])$ then

$$(\sigma, \mu, P \parallel [\lambda; s])_p \in (\sigma, \mu, \lambda; (P \parallel [s])_p)$$

Proof. Suppose $(\sigma, \mu, P \parallel [\lambda; s])_p \rightarrow^* (\sigma', \mu', P')$. Then there is a sequence of transitions $T \in \text{Traces}(\sigma, \mu, P \parallel [\lambda; s])_p$ that reaches $(\sigma', \mu', P')$. We need to show that there exists a configuration $(\sigma'', \mu'', P'')$ reachable from $(\sigma, \mu, [\lambda]_p; (P \parallel [s])_p)$ such that $(\sigma', \mu', P') \preceq (\sigma'', \mu'', P'')$.

- Case $T_i = \lambda$ for some $i$.

Then by the definition of left movers, $T_i$ can be moved to the left in $T$ until it appears first. The resulting trace is also a trace of $(\sigma, \mu, [\lambda]_p; (P \parallel [s])_p)$, so $\sigma'' = \sigma'$, $\mu'' = \mu'$, and $P'' = P'$.

- Case $T_i \neq \lambda$ for all $i$.

Then, by definition, $P' \equiv P'_\lambda \parallel [\lambda; s]_p$ and $(\sigma', \mu', P'_\lambda \parallel [\lambda; s]_p) \rightarrow^* (\sigma'_\lambda, \mu'_\lambda, P'_\lambda \parallel [s]_p)$ by executing $[\lambda]_p$.

It must be the case that $p \not\in \hprocs(\sigma', \mu', P')$ by definition, since $\lambda$ is enabled. The proof that $(\sigma', \mu', P') \preceq (\sigma'_\lambda, \mu'_\lambda, P'_\lambda \parallel [s]_p)$ is by contradiction, so we assume $(\sigma', \mu', P') \neq (\sigma'_\lambda, \mu'_\lambda, P'_\lambda \parallel [s]_p)$. Letting $H = \hprocs(\sigma', \mu', P')$ there are three cases for some $q \in H$:

- Case $\sigma'(x_q) \neq \sigma'_\lambda(x_q)$.

Then $x_q$ must be assigned in $\lambda$ and hence $q = p$ since processes only assign to local variables. But $p \in H$ is a contradiction, as $[\lambda]_p$ is enabled in $P'$.

- Case $\forall m. \mu'_\lambda(q', q, t) \neq \mu'(q', q, t) \cdot m$. Since $p$ executes $\lambda$, $p = q$ or $p = q'$. In the former case, then $p \in H$ which is a contradiction. In the latter $\lambda$ could have only appended messages to $\mu'(p, r', t)$, which contradicts the assumption that $\mu'_\lambda(p, q, t)$ does not contain $\mu'(p, q, t)$ as a prefix.

\[\Box\]
Lemma B.4. If $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta'; P', \Psi'$ then $\Gamma, \Delta, P, \Psi \in \Gamma', \Delta'; P', \Psi'$

Proof. The proof is by induction on the derivation of $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta'; P', \Psi'$, and by case analysis on the last step.

- Case R-SEND.

Let $(\sigma, \mu) \in [\Gamma, \Delta]$ and assume

$$(\sigma, \mu, \Psi_x \times \Psi; [\text{send}(t, x, n)]_p \times P_x) \rightarrow^{*} (\sigma', \mu', H)$$

and thus by definition of sequencing,

$$(\sigma, \mu \cdot \mu \Psi, [\text{send}(t, x, n)]_p \times P_x) \rightarrow^{*} (\sigma', \mu', H)$$

then by lemma B.3:

$$(\sigma, \mu \cdot \mu \Psi, [\text{send}(t, x, n)]_p ; P_x) \rightarrow^{*} (\sigma', \mu', H)$$

and hence

$$(\sigma, \mu \cdot \mu \Psi \cdot \{\text{send}(t, x, n)\}_{p \times P_x} ) \rightarrow^{*} (\sigma', \mu', H).$$

By assumption, $\sigma(x) = q$ and $\Gamma \notin E(q)$, so $(p, q, t) \notin \text{dom}(\mu \Psi)$ and thus

$$\mu \cdot \mu \Psi \cdot \{(p, q, t) \leftarrow \mu \cdot \mu \Psi \cdot (p, q, t) \cdot n\} = \mu \cdot \{(p, q, t) \leftarrow \mu \cdot (p, q, t) \cdot n\} \cdot \mu \Psi,$$

which gives us

$$(\sigma, \mu \{\text{send}(t, x, n)\}_{p \times P_x} ) \rightarrow^{*} (\sigma', \mu', H).$$

and hence

$$(\sigma, \mu \{\text{send}(t, x, n)\}_{p \times P_x} , \Psi_x \times \Psi; P_x) \rightarrow^{*} (\sigma', \mu', H).$$
• Case R-RECV.

Let \((\sigma, \mu) \in [\Delta, \Gamma]\) and assume

\[
(\sigma, \mu, \Psi_x \cdot \Psi; [x := \text{recv}(t, p)]_q \cdot P_x) \rightarrow^* (\sigma', \mu', H).
\]

Then,

\[
(\sigma, \mu \circ \mu, [x := \text{recv}(t, p)]_q \cdot P_x) \rightarrow^* (\sigma', \mu', H),
\]

and by Lemma B.3,

\[
(\sigma, \mu \circ \mu, [x := \text{recv}(t, p)]_q : P_x) \rightarrow^* (\sigma', \mu', H),
\]

and hence

\[
(\sigma, \mu \cdot \mu \cdot [((p, q, t) \leftarrow \text{pop}(\mu(p, q, t)))]_q : P_x) \rightarrow^* (\sigma', \mu', H).
\]

As in the case of R-SEND, we exchange buffers to produce

\[
(\sigma, \mu \cdot [(p, q, t) \leftarrow \text{pop}(\mu(p, q, t))]; \mu, P_x) \rightarrow^* (\sigma', \mu', H).
\]

• Case R-CONTEXT.

By straightforward application of the inductive hypothesis.

• Case R-Congruence.

By definition, as \(A \equiv B\) if and only if the set of program traces in \(A\) is equivalent to the program traces in \(B\).

• Case R-RECV-UNFOLD.

Let \((\sigma, \mu) \in [\Delta, \Gamma]\) and assume

\[
(\sigma, \mu, \Psi_x \cdot \Psi; [x := \text{recv}(Q, t)]_p \prod_{q \in Q'} [A]_q \cdot P_x) \rightarrow^* (\sigma', \mu', H).
\]
via some sequence of actions $T$. If $[x := \text{recv}(Q, t)]_q$ does not appear in $T$ then $T$
must also be a trace of $(\sigma, \mu, \Psi \times \Psi; [x := \text{recv}(q^* , t)]_p \parallel \prod_{q \in Q'} [A]_q)$ and hence of
$(\sigma, \mu, \Psi \times \Psi; [x := \text{recv}(q^* , t)]_p \parallel \prod_{q \in Q' \setminus q^*} [A]_{q^*})$.
Otherwise, there are configurations such that

$$(\sigma, \mu, \Psi \times \Psi; [x := \text{recv}(Q, t)]_p \parallel \prod_{q \in Q'} [A]_q \times P_x) \xrightarrow{T_0} (\sigma_0, \mu_0, P_0) \xrightarrow{[x := \text{recv}(Q, t)]_p} (\sigma_1, \mu_1, P_1) \xrightarrow{T_1} (\sigma', \mu', H)$$

and there is some $q \in Q$ such that $\mu_0(q, p, t) \neq \mu_1(q, p, t)$ and hence was the matching sender for $p$’s receive. First, we show by contradiction we that $q \in Q'$, for if $q \in Q \setminus Q'$ and the unique syntactic matching send must appear in $P_x$. But by the IH the syntactic matching send must be in $A$. Since $\text{rf}(\prod_{q \in Q'} [A]_q \times P_x)$ by assumption, this is a contradiction. Finally, since $q \in Q'$, the trace $T_0 \cdot [x := \text{recv}(Q, t)]_p \cdot T_1$ is a trace of the output program and we are done.

- **Case R-SEND-UNFOLD.**

The case follows by definition since the output is congruent with the input program.

- **Case R-SEND-RESID.**

Let $(\sigma, \mu) \in [\Delta, \Gamma]$ and assume

$$(\sigma, \mu, \Psi \times \Psi; [\text{send}(t, x, n)]_p \times P_x) \rightarrow^* (\sigma', \mu', H).$$

Then because $\Psi \times \Psi$ are sequenced, we can conclude for some $\mu_\Psi$,

$$(\sigma, \mu \cdot \mu_\Psi; [\text{send}(t, x, n)]_p \times P_x) \rightarrow (\sigma, \mu \cdot \mu_\Psi \cdot [(p, q, t) \leftarrow \sigma(n)]; P_x) \rightarrow^* (\sigma', \mu', H).$$

Now, there exists $(\sigma_V, \mu) \in [\Delta; V := V \cup n, \Gamma]$ such that $\sigma(n) \in \sigma_V(V)$. And further-
more,

\[(σ_ν, μ, \Psi_x \times \Psi; [\sum v:V.\text{send}(t, x, v)]_p \times P_x) \rightarrow^* (σ', μ', H)\].

And hence, as before,

\[(σ_ν, μ, \Psi_x \times \Psi; [\sum v:V.\text{send}(t, x, v)]_p \times P_x) \rightarrow^* (σ' \cdot μ; \Psi; [\sum v:V.\text{send}(t, x, v)]_p \times P_x) \rightarrow^* (σ', μ', H)\].

Since \(σ(n) \in σ_ν(V)\), there is a choice of \(v\) such that \(σ(n) = v\) resulting in

\[(σ_ν, μ, \Psi_x \times \Psi; [\sum v:V.\text{send}(t, x, v)]_p \times P_x) \rightarrow^* (σ' \cdot μ; \Psi; [\sum v:V.\text{send}(t, x, v)]_p \times P_x) \rightarrow^* (σ', μ', H)\].

which concludes the case.

- **Case R-COMPOSE-RESID.**

  This case follows by straightforward application of the inductive hypothesis, and from the fact that the rule explicitly checks that the residual term satisfies symmetric non-determinism.

- **Case R-WHILE-REPEAT** and **R-WHILE-REMOVE.**

  These cases follow by definition, as they simply “unfold” a single iteration of the while loop.

- **Case R-IF-THEN** and **R-IF-ELSE.**

  These cases follow by definition, as \((\text{if true \{ s_1 \} else \{ s_2 \}}) \equiv s_1\) (resp. false and \(s_2\)).

- **Case R-LOOP-UPD.**
The proof of this case is by induction on the size of $Q$.

- $|Q| = 1$.
  Then for some $q_0$, $Q = \{q_0\}$ and by definition,
  
  \[
  \text{for } (q : Q) \{ A \} = A[q_0/q] \\
  \prod_{q \in Q} B; C \equiv B; C[q_0/q].
  \]

By hypothesis,

\[
(\sigma, \mu) \in [\Gamma, \Delta] \\
(\sigma, \mu, \Psi_x \times \Psi; (A[q_0/q] \parallel B; C[q_0/q]) \times P_x) \rightarrow^* (\sigma', \mu', H).
\]

By the definition of $\Gamma_0$ and $\Delta_0$,

\[
(\sigma, \mu) \in [\Gamma_0, \Delta_0].
\]

By the rewriting hypothesis,

\[
(\sigma, \mu, \Delta^{q_0}; \Psi_x \times \Psi \times \Psi^{q_0}; X) \rightarrow^* (\sigma'', \mu'', H'')
\]

such that $(\sigma', \mu', H) \preceq (\sigma'', \mu'', H')$. By definition again,

\[
\Delta' = \Delta; \Delta^{q_0} = \Delta; \text{for } (q : \{q_0\}) \{ \Delta^q \} \\
\Psi' = \Psi \parallel \Psi^{q_0}
\]

so

\[
(\sigma, \mu, \text{for } (q : Q) \{ \Delta^{q_0}[q/q_0] \}; \Psi_x \times \Psi \parallel \Psi^{q_0}; P_x) \rightarrow^* (\sigma'', \mu'', H'')
\]

- $|Q| = n + 1$ and $IH(n)$.  

Then \( Q = Q' \cup \{ q_n \} \) for some \( q_n \notin Q' \), so

\[
\text{for } (q : Q) \{ A \} \equiv A(q_n); \text{for } (q : Q') \{ A[q_n/q] \}.
\]

and hence the obligation is solved by the inductive hypothesis, since

\[
\text{for } (q : Q) \{ A \} \equiv A(q_n); \text{for } (q : Q') \{ A[q_n/q] \} \parallel \prod_{q \in Q} B; C \parallel P_x
\]

is symmetrically nondeterministic as \( p \) is a concrete process. By the rewriting assumption, if for \((\sigma, \mu) \in [\Gamma, \Delta]\),

\[
(\sigma, \mu, \Psi_x \parallel \Psi; A[q_n/q] \parallel \prod_{q \in Q} B \parallel (\text{for } (q : Q) \{ A \} \parallel \prod_{q \in Q} C)) \rightarrow (\sigma', \mu', H),
\]

then

\[
(\sigma, \mu, \Delta'; \Psi_x \parallel \Psi'; \prod_{q \in Q \setminus u} B \parallel (\text{for } (q : Q) \{ A \} \parallel \prod_{q \in Q} C)) \rightarrow (\sigma', \mu', H),
\]

By “running” \( \Delta' \) we obtain \( \sigma_N' \), and hence

\[
(\sigma_N', \mu, \Psi_x \parallel \Psi'; \prod_{q \in Q \setminus u} B \parallel (\text{for } (q : Q) \{ A \} \parallel \prod_{q \in Q} C)) \rightarrow (\sigma', \mu', H),
\]

and apply the inductive hypothesis to finish the obligation.

\( \square \)
Bibliography


[40] Igor V. Konnov, Helmut Veith, and Josef Widder. What you always wanted to know about model checking of fault-tolerant distributed algorithms. In Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers, pages 6–21, 2015.


