CHARACTERIZING IMPLEMENTATIONS THAT PRESERVE PROPERTIES OF CONCURRENT RANDOMIZED ALGORITHMS

AMGAD RADY

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
DECEMBER 2017

Abstract

We show that correctness criteria of concurrent algorithms are mathematically equivalent to the existence of so-called simulations between implementations of the algorithms in a well-known framework (that of input/output automata) and simple canonical automata. This equivalence allows us to frame our proofs of correctness in a language much more amenable to machine-checking than conventional proofs.

We give the first demonstration that when strongly linearizable implementations of randomized concurrent algorithms are utilized, then the distributions of a well-defined class of random variables are preserved under object substitution by non-concurrent implementations of the same algorithms. We also consider weaker conditions than strong linearizability under which implementations are still correct in the presence of randomization.

This thesis is dedicated to my mother, Heba A. Hassan, without whose selfless love and support this would be impossible.

Acknowledgements

I would like to first thank my supervisor, Professor Eric Ruppert of the Department of Electrical Engineering and Computer Science at York University. His generosity with his time, wise, patient, and insightful guidance, and insistence on clarity and rigour have all made me a better researcher. His cheerful demeanour made this journey a pleasure.

I would next like to thank Professor Franck van Breugel, Graduate Program Director of the Department of Electrical Engineering and Computer Science at York University. His door was always open when I needed his help, and his insights and encouragement were nothing short of invaluable.

Last but not least, I would like to thank Professor Thomas Salisbury of the Department of Mathematics and Statistics at York University for sitting on my examination committee, for his input on the areas of my work that touched on the theory of probabilities, and for the fresh perspective that only an outsider could bring.

Table of Contents

1 Linearizability

Linearizability, defined formally in Definition 4.3.2, is a crucial safety property of asynchronous shared-memory systems introduced by Herlihy and Wing in (HW90). An execution of an implementation of an abstract data type by several processes (abstractions of processors, control threads, etc.) accessing and modifying a set of shared objects is said to be *linearizable* if the operations, which are in actuality interleaved by the executing system's scheduler, can be thought of as taking place *atomically* at some point during the operation. By this we mean there is some sequential ordering of the operations that is indistinguishable from the actual execution to any observer that cannot examine the implementation of the system. It is possible to show that an execution is linearizable by specifying "linearization points" – specific steps where the operations 'take effect.'

The concept of linearizability was a crucial advance in the study of distributed computation, since sequential executions are far more analytically tractable than concurrent ones. Also, executions that are not linearizable (and the algorithms that

produce them) are almost always undesirable from a system design perspective, as the following example will demonstrate.

1.1 A Rudimentary Bank Account

Suppose we wish to implement a bank account with two operations – DEPOSIT(n) and WITHDRAW(n) – with the following properties:

- DEPOSIT(n) must increment the account balance by n at termination and return TRUE.
- WITHDRAW(n) must, if the account contains at least n, disburse n to the invoker and decrement the account balance by n. If n exceeds the balance, WITHDRAW must return **insufficient funds** and leave the balance unaltered.

Consider two individuals, A and B, who simultaneously perform some sequence of the above operations from separate terminals on a joint bank account. We say that we can linearize the bank's processing of these transactions if we can propose a sequential order of the transactions that is indistinguishable from the actual execution of the transactions by the bank's system, if we cannot inspect the implementation details of that system. Let us consider the following implementations for the operations DEPOSIT(n) and WITHDRAW(n), respectively, using a shared

memory location – implemented as a multi-reader/multi-writer register – called "account":

Algorithm 1: DEPOSIT(n) - Sequential

```
1 read the amount in the account and store in local variable v;
2 v ⟵ v + n;
3 write v to account;
4 return TRUE;
```

Algorithm 2: WITHDRAW(n) - Sequential

```
1 read the amount in the account and store in local variable v;
2 v ⟵ v − n;
3 if v ≥ 0 then
4 |   disburse n;
5 |   write v to account;
6 else
7 |   fail due to insufficient funds
8 end
```

These algorithms are valid implementations of the operations in a sequential context. To see where the problem lies when steps are interleaved in an asynchronous system, suppose that both A and B perform DEPOSIT(100). Clearly, all valid executions of these two operations should leave the account with \$200 at termination. However, suppose that a concurrent system implementing Algorithm

1 is scheduled as follows:

- A and B read 0 and assign 0 to their local variables.
- A and B each stores $0 + 100 = 100$ into its local variable v.
- A writes 100.
- B writes 100.

The account has \$100 at termination of both operations. Therefore, this execution is not linearizable (and we now have two irate customers). Why has this difficulty arisen? Let us examine Algorithm 1 more closely.

We observe that the **read** and **write** operations are separated in time. This is not a problem in a sequential setting, as no process can cause any mischief between the invocations of the **read** and **write** operations. However, as we have seen, A interferes with B's write since it invalidates B's cached copy of the account balance. How is this to be repaired?

One solution is to use a locking mechanism like a *mutual exclusion object*. A mutual exclusion object (or mutex), is an object that allows several processes to share a resource (here the location in memory where the account balance is stored) ensuring that each process can employ the resource without interference by other processes.

While this is a simple fix for our issue – and indeed is used in practice – it has shortcomings of its own. By design, locks enable a process, for a shorter or longer period, to monopolize the protected resource to the exclusion of other processes. If the process holding the mutex is slow or *in extremis*, the entire system is hampered in the same proportion. This is a pertinent worry, as we often cannot anticipate how an operating system will choose to schedule processes, which it may privilege or neglect.

Having seen that using a locking mechanism could cause all of the system's processes to crawl or crash, we desire a stronger guarantee of progress for our system.

Definition 1.1.1. An algorithm using a set of processes $\mathcal{P}$ is *lock-free* if, for every infinite execution, including those in which some processes experience halting failures, infinitely often some process in $\mathcal{P}$ completes its operation.

It might be that a cleverer implementation of the operations that is linearizable and uses only **read** and **write** instructions on shared registers is possible. In fact, this is true for the DEPOSIT operation; however, this is not true for the WITHDRAW operation if we impose the condition of lock-freedom.

Lemma 1.1.2. *There is no lock-free implementation of the WITHDRAW operation using only shared read/write registers.*

We postpone the proof of this lemma until we introduce the *consensus problem* (Definition 1.2.1).

Fortunately, hardware designers have introduced an operation known as **compare & swap** (or CAS) which, instead of simply writing a value to a shared register, does the following sequence of operations *atomically* – i.e., the constituent steps of the algorithm are performed without interruption.

Algorithm 3: COMPARE & SWAP

Input: $loc : \&T, test : T, v : T$

```
1 if *loc ≠ test then
2 |   return FALSE
3 else
4 |   *loc ⟵ v;
5 |   return TRUE
6 end
```

Why have designers chosen to implement CAS in hardware? To understand the answer we must take a brief jaunt into the theory of distributed computation and one of its foundational problems: consensus.

1.2 The Binary Consensus Problem

The *Binary Consensus Problem*, to be defined shortly, plays a role in distributed computing analogous to that played by the Halting Problem in the theory of com-

putability: It is a rich source of lower bound and impossibility results. We consider the Binary Consensus Problem with crash failures, where processes either execute their instructions faithfully or irrecoverably crash.

Definition 1.2.1. A protocol involving a set of processes $\mathcal{P}$ with input set $\mathcal{I} = \{0, 1\}$ solves the Binary Consensus Problem if every execution of the protocol satisfies the following conditions:

- Termination: Every correct process outputs some value, where a process is correct if it faithfully executes its given protocol.
- Validity: If all processes propose the same value v, every correct process outputs v.
- Agreement: All correct processes output the same value.

An object's *consensus number* is defined to be the maximum number of processes for which binary consensus can be solved in a lock-free manner using only objects of that type and read/write registers, or $\aleph_0$ if no such maximum exists. Read/write registers themselves have a consensus number of 1 (Her91, LAA87), and stacks a consensus number of 2. CAS objects have consensus number $\aleph_0$, meaning they can solve binary consensus for any number of processes (Her91).

We now give the proof for Lemma 1.1.2:

Proof. Let us suppose that there is some implementation of WITHDRAW (which we shall also term WITHDRAW) that is lock-free and uses only read/write registers. We use this to derive a contradiction to the claim that read/write registers have consensus number 1 by giving an algorithm using WITHDRAW and the **read** operation that can solve the binary consensus problem for any number of processes – implying that read/write registers have consensus number $\aleph_0$.

Let $n \in \mathbb{N}$ and $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ be a set of n processes, each given an initial value of either 0 or 1, and consider the following algorithm that each process implements on a shared register initialized to the value 100.

Algorithm 4: Binary consensus for n processes using WITHDRAW

```
1 if initial value = 1 then
2 |   WITHDRAW(99)
3 else
4 |   WITHDRAW(100)
5 end
6 read the value in account and return it;
```

Let us now consider the conditions of termination, validity, and agreement for the above algorithm.

- Termination: Since the implementation of WITHDRAW is lock-free, and the **read** operation on registers is atomic, for each correct process p_i, its

WITHDRAW operation eventually terminates. Since there are finitely many processes, the algorithm itself eventually terminates.

- Validity: If all processes p_i have the value 0, then the first process to successfully perform WITHDRAW will remove 100 from **account**. All subsequent **read** operations will then return 0. The argument for value 1 is similar.

- Agreement: Suppose that the first process to successfully perform WITHDRAW has initial value 0. All subsequent **read** operations return 0. The argument for value 1 is similar.

□

1.3 Attempted Repair of Banking Example

Let us make a first attempt to solve our problem by directly substituting a CAS operation for **write** in our example:

Algorithm 5: DEPOSIT(n) - Attempt I

```
1 read the amount in the account and store in local variable v;
2 v' ⟵ v + n;
3 CAS(account, v, v');
```

Algorithm 6: WITHDRAW(n) - Attempt I

```
1 read the amount in the account and store in local variable v;
2 v' ⟵ v − n;
3 if v ≥ 0 then
4 |   disburse n;
5 |   CAS(account, v, v');
6 else
7 |   return insufficient funds
8 end
```

A couple of problems are evident with this substitution: First, if the CAS fails in Algorithm 5, the operation halts without having deposited money into the account. While it might be feasible, if annoying, for the user to repeat the operation until it is successful, this rapidly becomes burdensome in a congested system. As we shall

soon see, this is easily fixed by enclosing the CAS in a **while** loop.

The second and much more dangerous problem is the order of the 'disburse' and CAS operations in Algorithm 6. The funds are disbursed before the account is altered; while this might not have been a problem in the sequential system, it is a glaring vulnerability here. Suppose that a pair of observant and unscrupulous customers A and B keep performing DEPOSIT(MIN) and WITHDRAW(MAX), respectively, where MIN and MAX are constants denoting the lowest and highest denominations of currency that the system accepts, respectively. At least some of the time, a DEPOSIT (Algorithm 5) changes the balance between WITHDRAW's (Algorithm 6) **read** and CAS. The system will disburse the maximum amount of funds but the account will not be decremented since the CAS fails.

Both failures, the inconvenient first and disastrous second, arise because we mechanically substituted a CAS for a **write** operation. We hope that this brief example illustrates to the reader that a measure of caution and subtlety is required when implementing even simple operations in a distributed setting using lock-free primitives like CAS. Let us now present the correct code.

Algorithm 7: DEPOSIT(n) - Attempt II

```
1 while true do
2 |   read the amount in the account and store in local variable v;
3 |   v' ⟵ v + n;
4 |   if CAS(account, v, v') then
5 |   |   break;
6 |   end
7 end
```

Algorithm 8: WITHDRAW(n) - Attempt II

```
1  while true do
2      read the amount in the account and store in local variable v;
3      v' ⟵ v − n;
4      if v' ≥ 0 then
5          if CAS(account, v, v') then
6              disburse n;
7              break;
8          end
9      else
10         fail due to insufficient funds
11     end
12 end
```

1.4 A Safe Rudimentary Bank Account

We have repaired the errors that arose when we mechanically transplanted the sequential implementations (Algorithms 1 and 2) into a distributed setting (Algorithms 5 and 6). However, while Algorithms 7 and 8 are safe in the sense that certain classes of vulnerability are absent, there remains the possibility of resource deprivation in our system. Let us consider the following execution of Algorithms 7

and 8 by two users A and B:

- A and B wish to perform WITHDRAW(10) once and DEPOSIT(1) *ad infinitum*, respectively, on an account with \$100 initially.
- A reads, B performs CAS and increments the account, A performs CAS and fails, A reads...

We can arbitrarily delay the completion of a procedure – in this example A's WITHDRAW(10) – by strategically interleaving other processes. A **wait-free** system is one in which we cannot do this, i.e., one where every process can complete its operation within a finite number of steps. We can modify our implementation to make it wait-free; however, doing so is beyond the scope of our discussion as we are principally interested in safety, not progress.

We now argue for the linearizability of Algorithm 7 and Algorithm 8:

- DEPOSIT(n) has two operations on shared data: **read** and CAS. The read value is used only as an argument to CAS, hence DEPOSIT is linearized at a successful CAS. An invariant maintained by the DEPOSIT operation is that immediately preceding a successful CAS on line 4, the amount present in the account is v, and immediately after the CAS, the amount present is $v + n$.
- WITHDRAW(n) is more interesting. How the value WITHDRAW reads from

the shared register is used depends on whether the account has sufficient funds at the point of WITHDRAW's **read**:

- If so, and the CAS is successfully performed, we linearize at the CAS. If the CAS is unsuccessful, we loop anew. If sufficient funds are present, the operation is correct since immediately before the CAS on line 5, the amount present in the account is v, and immediately after the CAS the amount is $v - n \geq 0$.
- If not, we linearize at the previous **read**. This is correct since immediately after the **read** operation on line 2, the account had an amount $v < n$.

As the previous example has shown, even simple algorithms like DEPOSIT and WITHDRAW can fail critically in distributed systems if care is not taken to ensure their correctness. This is not merely an academic curiosity: multi-core processors (an example of a distributed system) have become more widespread as physical limits on processor clock speeds are reached (ABD+09). Linearizability is a simple and natural criterion of correctness that has stood the test of time. Despite its simplicity, proofs of linearizability for non-trivial algorithms oftentimes require subtle techniques (DD15).

2 Strong Linearizability

In the bank account example, we can substitute our linearizable implementations for the atomic operations and rest assured that the system will retain its correctness, in the sense that for any execution of the implementation there is an equivalent sequential execution.

Unfortunately – when randomization is introduced into algorithms that use shared objects – there are certain examples where substituting linearizable implementations for atomic operations will manifest very subtle errors in the implemented object. Randomized algorithms are of great importance in shared-memory systems, and can sometimes render uncomputable problems tractable. For example, while binary consensus is not solvable in the presence of crash failures by deterministic algorithms that use only read/write registers (LAA87), (Her91), there are correct randomized binary consensus algorithms that terminate with probability 1 (CIL94).

Let us examine an example due to Golab et al. (GHW11) using a linearizable implementation by Vidyasankar (Vid88).

2.1 A SRSW Implemented Using Bit Array

Consider a single-reader/single-writer register with domain $\{0, 1, \ldots, n\}$ implemented using an array $A[0 \ldots n]$ of atomic single-reader/single-writer bit registers (i.e., registers with domain $\{0, 1\}$), where the value $0 \leq k \leq n$ is represented by 1 in the k^{th} position of the array and 0 in every position preceding the k^{th} (the values in positions after the k^{th} are irrelevant to the representation). The WRITE(k) and

READ() operations are implemented as follows:

Algorithm 9: WRITE(k)

```
1  A[k] ⟵ 1
2  for i ← k − 1, k − 2, . . . , 0 do
3  |   A[i] ⟵ 0
4  end
```

Algorithm 10: READ()

```
1  i ← 0
2  while A[i] = 0 do
3  |   i ← i + 1
4  end
5  k ← i
6  for j ← i − 1, i − 2, . . . , 0 do
7  |   if A[j] = 1 then
8  |   |   k ← j
9  |   end
10 end
11 return k
```

Consider two processes, r and w, with the following operations on a register initialized to 1:

- r: READ().
- w: WRITE(2), $c \leftarrow \textbf{Uniform}(\{0, 2\})$, WRITE($c$).

By $c \leftarrow \textbf{Uniform}(\{0, 2\})$, we mean that c is set to the value 0 or 2 with equal probability.

We make a familiar argument commonly used in computer science to establish lower bounds and impossibility results: the adversarial argument. Suppose that an adversary that controls the scheduling of steps wishes to minimize the expected value of r's READ() operation. In the case of an atomic implementation, the adversary should schedule r's READ operation before the first or after the second of w's WRITE operations. In both cases, the expected value of the READ() operation is 1.

Suppose that we replace the atomic implementation of READ and WRITE with Vidyasankar's linearizable implementations, Algorithms 9 and 10, respectively. Initially, the array A contains $[01000\ldots0]$, and the adversary defines the scheduler $\mathcal{I}$ that performs the following scheduling:

- Perform r's READ() up to the termination of the while loop (which terminates when $i = 1$).

- Perform w's operations to completion. This leaves A in one of the following states: $[00100\ldots0]$ or $[10100\ldots0]$, with equal probability.
- Perform the remainder of r's operation, which returns either 1 or 0 with equal probability.

Let us note that the scheduler $\mathcal{I}$ described above is completely oblivious to the coin flip's outcome. We can attempt to define a "scheduler" $\mathcal{A}$ for the atomic operations that would have the same expected value for r's READ() outcome under scheduler $\mathcal{I}$ for Vidyasankar's implementation. The only possible way to achieve an expected READ() value of $\frac{1}{2}$ using atomic operations is shown in the following table:

$c = 0$	$c = 2$
w: WRITE(2)	r: READ()
w: $c \leftarrow 0$	w: WRITE(2)
w: WRITE(0)	w: $c \leftarrow 2$
r: READ()	w: WRITE(2)

We see here that if the coin flip outcome is 0, $\mathcal{A}$ schedules r's READ() last, and if the the coin flip outcome is 2, $\mathcal{A}$ schedules r's READ() first. In order to match the expected value for READ() produced by an oblivious scheduler $\mathcal{I}$, $\mathcal{A}$ must be clairvoyant!

2.2 Ameliorating Linearizability's Insufficiency

The reader can verify that no other scheduler will exhibit an expected value of $\frac{1}{2}$ for r's READ(). Unhappily, linearizability is not a sufficient condition for maintaining the distribution of operations' returned values in a program. What can be done to remedy this?

Golab et al. (GHW11) strengthened the definition of linearizability to disqualify implementations such as Vidyasankar's (Vid88) above. We formally introduce *strong linearizability* in Definition 4.3.3.

We observe that for both coin flip outcomes, $\mathcal{A}$'s executions are valid linearizations of the corresponding executions given by $\mathcal{I}$. Hence, $\mathcal{A}$ defines a linearization function for the set of executions given by $\mathcal{I}$ in a natural way. This function is not prefix-preserving, hence it is not a strong linearization function since it fails to satisfy the second condition of Definition 4.3.3.

3 Main Results and Related Work

Having introduced the core concepts of linearizability and strong linearizability to the reader, we can now give an outline of the results that appear in this thesis. The thesis is divided into two parts:

- In Chapter 4 and 5 we introduce input/output automata and simulations. We give the first formal proof in the literature that the existence of a forward simulation between an implementation of an abstract data type and a canonical linearization automaton of the abstract data type is equivalent to the implementation being strongly linearizable (Theorem 5.2.3). For completeness we give a proof of an analogous result for forward-backward simulations and linearizability (Theorem 5.1.1). The former result is especially important: By linking strong linearizability with forward simulations – a commonly used technique in computer verification (SAGG$^+$93) – it forms the theoretical basis for an automated strong linearizability checker.

- In Chapter 6 we apply our result to a singly-linked list based set algorithm to

demonstrate the existence of a forward simulation from the implementation to the canonical automaton for the set abstract data type by showing that the algorithm is strongly linearizable.

- In Chapter 7 we introduce and consider systems of linearizable/strongly linearizable objects, whereas in the previous chapters we considered individual objects in isolation. In Chapter 8 we introduce randomization to our systems and define the probability space of executions. In Chapters 9-11 we compare different systems and examine the conditions under which the systems can be said to be indistinguishable. The main results of this part of the work are Theorem 10.1.3 and Theorem 10.1.4. As far as we can tell, this is the first direct analysis of the relationship between random variables defined on executions and strong linearizability in the literature. Previous discussions (GHW11) dealt only indirectly with this question through considering the existence of equivalent adversaries.

As previously noted, strong linearizability was introduced by Golab, Higham, and Woelfel in (GHW11). To the best of our knowledge, the links between strong linearizability and simulations have not previously been investigated – this is in contrast to the case of the links between linearizability and simulations. Furthermore, while Golab *et al.* show that strong linearizability is a necessary and sufficient

condition for the preservation of distributions of random variables defined on the outcomes of executions, this is done by showing the existence of adversaries that are "equivalent" or, in our terminology, directly linked (see Definition 9.0.1). As we will show, this is not quite identical to preserving the distributions of random variables defined on executions. There are programs and adversaries for which an equivalent adversary (in the sense of Golab) does not exist, yet an non-equivalent adversary exists that still preserves the distributions of random variables.

While we make only cursory note of progress conditions for strongly linearizable implementations of objects, this very interesting research avenue is explored by Helmi, Higham, and Woelfel in (HHW12), where the authors demonstrate that

- there is no strongly linearizable lock-free implementation of multi-writer registers, max-registers, snapshots, and counters from multi-reader/single-writer atomic registers. In contrast, these objects have wait-free linearizable implementations from multi-reader/single-writer atomic registers.

- There is a universal strongly linearizable obstruction-free (a weaker progress condition than lock-freedom, where a process eventually completes its operation if it is scheduled exclusively until termination of its current operation) implementation of any object from multi-reader/single-writer registers.

- There is a strongly linearizable wait-free implementation of bounded max-

registers from multi-reader/multi-writer registers.

Denysyuk and Woelfel build on this work in (DW15) by introducing improved proof techniques to prove the following results:

- There are no deterministic strongly linearizable wait-free implementations of snapshots, counters, or max-registers for three or more processes from multi-writer registers.

- There exist deterministic strongly linearizable lock-free implementations of counters, snapshots, and logical clock objects for any number of processes from multi-writer registers. Note that this answers an open question raised in (HHW12) on whether employing multi-writer instead of single-writer registers would improve the progress guarantee of the implementation – this result answers the question in the affirmative.

4 Abstract Data Types, Automata, and Simulations

So far, we have been quite informal in what we mean by terms like "execution" or "scheduler." We trust that the reader has sufficient grounding in the relevant topics to be able to understand these heretofore imprecise notions. In the interest of rigour and precision, we now give a model of abstract data types and input-output automata that will serve as the theoretical foundation upon which we can build our results.

4.1 Abstract Data Types

We begin by introducing a general notion of abstract data types. We use ADT's to specify the behaviour, at a high level, of the set of implemented objects on which the processes act.

Definition 4.1.1. An *abstract data type* (abbreviated to ADT) $\mathcal{D}$ is a 5-tuple

$(Q, OPS, RSP, \delta, q_0)$ with a set of states Q, a set of operations OPS, a set of responses RSP, a transition function $\delta : Q \times OPS \longrightarrow \mathcal{P}(RSP \times Q) \setminus \{\emptyset\}$ where δ has *finite non-determinism* – i.e., for each $(q, op) \in Q \times OPS$, $|\delta(q, op)| < \infty$ – and a unique start state $q_0 \in Q$.

If $(r, q') \in \delta(q, op)$, it means that if op is performed when the object is in state q, one possible outcome is that op returns r and the object changes state to q'. Note that abstract data types may be – at least without *a priori* restrictions – inherently non-deterministic.

An abstract data type is called *deterministic* if, for each $(q, op) \in Q \times OPS$, $|\delta(q, op)| = 1$. We restrict our attention to finite non-determinism since this restriction avoids a host of pathological behaviours; we lose little by making such a restriction, as most realistic and useful automata have finite non-determinism. For completeness, we shall include an example (Counterexample 4.3.6) of the sort of pathology that can arise in the absence of finite non-determinism.

As an example of an ADT, we can model a multi-writer/multi-reader register

that stores a natural number as follows:

$$Q = \mathbb{N}$$

$$OPS = \{write(j) \mid j \in \mathbb{N}\} \cup \{read\}$$

$$RSP = \{i \mid i \in \mathbb{N}\} \cup \{ack\}$$

$$\delta(i, write(j)) = \{(ack, j)\}$$

$$\delta(i, read) = \{(i, i)\}$$

$$q_0 = 0.$$

Definition 4.1.2. A *sequential history* S of an abstract data type $\mathcal{D}$ for a set of processes $\mathcal{P}$ is a finite or infinite sequence $a_1 a_2 \ldots$ where, for each i, $a_i = (op_i, rsp_i, p_i) \in OPS \times RSP \times \mathcal{P}$ and there exists a sequence of states $q_0 q_1 \ldots$ where q_0 is the start state of $\mathcal{D}$ and, for each i, $(rsp_i, q_i) \in \delta(q_{i-1}, op_i)$.

The set of sequential histories of an ADT $\mathcal{D}$ is called the *sequential specification* of $\mathcal{D}$. We note that by Definitions 4.1.1 and 4.1.2, the sequential specification of an ADT is *prefix-closed*, i.e., if a sequential history h is in the sequential specification of $\mathcal{D}$, then every prefix of h is in $\mathcal{D}$'s sequential specification.

4.2 Input-Output Automata

We will use Lynch and Vaandrager's definition of automata (LV95) to model both the safety properties (in our context, linearizability) and implementation of shared

objects.

Definition 4.2.1. (LV95) An *automaton* A consists of:

- A set $states(A)$ of states.
- A non-empty set $start(A) \subseteq states(A)$ of start states.
- A set $acts(A)$ partitioned into a set $\underline{acts}$ of internal actions and a set $\overline{acts}$ of external actions.
- A set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of steps.

We write $s' \xrightarrow{a}_A s$ if $(s', a, s) \in steps(A)$.

Since the automata of Definition 4.2.1 can have an infinite number of states, they can be used to model Turing machines in a natural way. Therefore, any program can be represented by an input-output automaton.

Definition 4.2.2. An *execution* of an automaton A is an infinite sequence

$$s_0 a_1 s_1 a_2 \ldots$$

or finite sequence

$$s_0 a_1 s_1 \ldots a_k s_k$$

of alternating states and actions of A where $s_0 \in start(A)$ and, for each i, $s_i \xrightarrow{a_{i+1}}_A s_{i+1}$.

Definition 4.2.3. The *interpretation* $\Gamma(E)$ of an execution E is the subsequence of E formed by projecting E onto the set of external actions.

Thus, in an interpretation, both the states and internal actions are removed. We also refer to the interpretations of an automaton A, by which we mean the set

$$\mathcal{H}_A = \{\Gamma(E) \mid E \text{ is an execution of } A\}.$$

The external actions of an automaton are those that can be viewed by an outside observer, while internal actions cannot be so viewed. We compare two automata A and B by comparing the sets $\mathcal{H}_A$ and $\mathcal{H}_B$ of sequences of external actions generated by A and B, respectively.

4.3 Linearizable and Strongly Linearizable Implementations

Definition 4.3.1. An automaton A *implements* or is an *implementation* of an ADT $\mathcal{D} = (Q, OPS, RSP, \delta, q_0)$ for a set $\mathcal{P}$ of processes if the set of external actions of A is

$$\{invoke(op, p), respond(rsp, p) \mid op \in OPS, rsp \in RSP, p \in \mathcal{P}\}$$

and every execution E of A is *well-formed* – i.e., for every process $p \in \mathcal{P}$, $\Gamma(E)||p$, the projection of $\Gamma(E)$ onto operations by p, is of the form

$$invoke(op_0, p)respond(rsp_0, p)invoke(op_1, p)respond(rsp_1, p)\ldots$$

Although Definition 4.3.1 is very basic, in that implementations need not satisfy any safety or progress guarantees beyond elementary typing, there are many ways to elaborate on and extend Definition 4.3.1 to add such properties, including the ones we will consider: *linearizability* (Definition 4.3.2) and *strong linearizability* (Definition 4.3.3).

We also refer to a specific *instance* of an operation $op \in OPS$ by a process $p \in \mathcal{P}$, by which we mean a specific invocation $invoke(op, p)$ and the response immediately following it in $\Gamma(E)||_p$, if such a response exists.

We adapt the definitions of *linearizability* given in (HW90) and *strong linearizability* given in (GHW11) to automata.

Definition 4.3.2. An execution E of an automaton implementing an ADT $\mathcal{D}$ for $\mathcal{P}$ is *linearizable* if there is an extension H' of $H = \Gamma(E)$ obtained by adding responses to some subset of pending operation instances –i.e., instances of operations that have been invoked but not yet returned – of H, and a sequential history S of $\mathcal{D}$ such that:

- complete(H') is equivalent to S, where complete(H') is the history formed by removing pending operation instances from H'. By equivalent, we mean that each process has the same number of operations and corresponding operation instances have the same responses.

- If the response to operation instance op_i by p_i precedes the invocation of operation instance op_j by p_j in H then the triple corresponding to op_j does not occur before the triple corresponding to op_i is S.

Such a sequential history S is called a *linearization* of E.

Definition 4.3.3. An automaton A is *strongly linearizable* with respect to $\mathcal{D}$ if there is a function $g : \mathcal{E}_A \longrightarrow \mathcal{S}_\mathcal{D}$ mapping the set of all executions of A to the sequential specification of $\mathcal{D}$ such that:

- For any $E \in \mathcal{E}_A$, $g(E)$ is a linearization of $\Gamma(E)$.

- g is prefix-preserving, i.e., if E is a prefix of E' then $g(E)$ is a prefix of $g(E')$

We say that an implementation is linearizable if each execution it produces is linearizable. We say that an implementation is strongly linearizable if the *set* of all its executions is strongly linearizable. Note that if an implementation is strongly linearizable, then it is linearizable. We will have more to say about this when we discuss systems.

We use the following automaton, first introduced by Lynch in (Lyn96) and reformulated by Doherty in (Doh04), to model a minimal linearizable implementation of an abstract data type shared by a collection of processes. It is intended to produce all possible linearizable sequences of operation invocations and responses by a finite set of processes $\mathcal{P}$.

Definition 4.3.4. The *abstract linearization automaton* (called *canonical automaton* in (Doh04)) B of an abstract data type $\mathcal{D} = (Q, OPS, RSP, \delta, q_0)$ over a set of processes $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ is defined by

- $states(B) = \{idle, invoked(op), done(rsp) \mid op \in OPS, rsp \in RSP\}^n \times Q$.
- $start(B) = \{(idle, idle, \ldots, idle, q_0)\}$.
- $acts(B) = \{invoke(op, p), do(op, p), respond(rsp, p) \mid op \in OPS, rsp \in RSP, p \in \mathcal{P}\}$, where $invoke(op, p)$ and $respond(rsp, p)$ are external actions, and $do(op, p)$ is an internal action.
- The following is the set of steps of B:
  - $(\ldots, idle, \ldots, q) \overset{invoke(op,p_i)}{\longrightarrow} (\ldots, invoked(op), \ldots, q)$ for $q \in Q, op \in OPS, p_i \in \mathcal{P}$.
  - $(\ldots, invoked(op), \ldots, q) \overset{do(op,p_i)}{\longrightarrow} (\ldots, done(rsp), \ldots, q')$ for $q, q' \in Q$, $op \in OPS, p_i \in \mathcal{P}$, and $(rsp, q') \in \delta(q, op)$.
  - $(\ldots, done(rsp), \ldots, q) \overset{respond(rsp,p_i)}{\longrightarrow} (\ldots, idle, \ldots, q)$ for $q \in Q, op \in OPS, p_i \in \mathcal{P}$.

  The ellipses indicate that all elements of the tuple except the printed one are unchanged.

The abstract linearization automaton models the states of processes: whether they are idle or performing some operation. If the latter, then $invoked(op)$ and $done(rsp)$ in the i-th position indicates that process p_i has invoked operation op or is has completed an operation and wishes to return rsp, respectively. This obviates the need for input and output buffers, as given in (Lyn96). A $do(op, p_i)$ indicates the linearization point of the operation instance op. This defines a linearization, by taking operation instances in the order of their do actions. The responses of the processes are consistent with the linearization order since both the rsp value and the state variable q of the ADT are set atomically (i.e., in a single step) by the do action.

Let us explore some properties of implementation and linearization automata.

Lemma 4.3.5. *The set of interpretations $\mathcal{H}_B$ of abstract linearization automaton B of ADT $\mathcal{D}$ is limit-closed, i.e., if every finite prefix of an infinite sequence H is in $\mathcal{H}_B$, then $H \in \mathcal{H}_B$.*

Proof. Let $Pre(H) = \{H_0, H_1, \ldots\}$ where H_i is the length-i prefix of an interpretation H and suppose $Pre(H) \subseteq \mathcal{H}_B$. We construct a directed graph G with vertices of the form (H_i, E) where E is an execution of B with $\Gamma(E) = H_i$.

Let G_0 be the graph consisting of the single vertex $v_0 = (\varepsilon, (idle, idle, \ldots, idle, q_0))$,

where ε is the empty sequence. Let G_i be the graph with vertex set

$$V(G_i) = \{(H_j, E) \mid j \leq i, \Gamma(E) = H_j\}.$$

$|V(G_i)| < \infty$, since by Definition 4.3.4 and finite non-determinism, the number of executions of B with interpretation H_j is finite.

G_i has the edge set

$$E(G_i) = \{((H_j, E'), (H_{j+1}, E)) \mid j < i, E' \text{ is a prefix of } E\}.$$

We now show that, for each $i \geq 0$, there is a path from v_0 to each vertex in G_i. The claim holds trivially for G_0. Suppose that the claim holds for some $i \geq 0$. Consider a vertex $v = (H_j, E)$ in G_{i+1}. If v is also a vertex of G_i, then the claim is true, since G_i is a subgraph of G_{i+1}.

If v is not in G_i, then $v = (H_{i+1}, E)$. Let E' be the shortest prefix of E with $\Gamma(E') = H_i$. By construction of G_{i+1}, there is an edge $((H_i, E'), (H_{i+1}, E)) \in G_{i+1}$. By the induction hypothesis, there is some path from the root to (H_i, E'). Therefore, the claim is true.

Let $G = \bigcup_{i=0}^{\infty} G_i$:

1. G is an infinite graph since it contains distinct vertices of the form $\{(H_j, \_)\}_{j \in \mathbb{N}}$.

2. Each vertex of G has finite out-degree, since in any execution of B, there are at most $|\mathcal{P}|$ internal actions between any two external actions.

3. There is a path from v_0 to every to each vertex of G.

By König's Lemma (Lemma 2.1 of (LV95)), G has an infinite path $(H_1, E_1)(H_2, E_2)\ldots$ with finite prefixes $\{H_i \mid i \geq 0\}$. Therefore, $H \in \mathcal{H}_B$. The execution $E = \lim_{m\to\infty} E_n$ is a valid execution since E_n is a prefix of E_{n+1} for all $n \geq 1$, and $\Gamma(E) = H$.

□

Without the finite non-determinism restriction placed on ADT's, Lemma 4.3.5 would not hold, as the following counter-example from (GR14) will demonstrate.

Counter Example 4.3.6. We construct an ADT representing a countdown object. This object first chooses some natural number, and then counts down from that number – outputting '1' with each decrement – until it reaches 0 – whereupon the object responds '0' to each invocation.

The object has infinite non-determinism in its initial choice of a natural number.

$$Q = \{q_i \mid i \geq 0\}$$

$$OPS = \{op\}$$

$$RSP = \{0, 1\}$$

$$\delta(q_i, op) = \begin{cases} \{(1, q_j) \mid j > 0\} & \text{if } i = 0 \\ \{(0, q_i)\} & \text{if } i = 1 \\ \{(1, q_{i-1})\} & \text{if } i > 1. \end{cases}$$

Let H be the following interpretation of an execution of an automaton A implementing the countdown ADT for a single process:

$$invoke(op)respond(1)invoke(op)respond(1)\ldots$$

Every prefix H_j of length j for $j \geq 0$ is an interpretation of an execution of A, as the following execution shows.

$$(idle, q_0) \overset{invoke(op)}{\longrightarrow} (invoked(op), q_0) \overset{do(op)}{\longrightarrow} (done(1), q_{j+1}) \overset{respond(1)}{\longrightarrow} (idle, q_{j+1}) \overset{invoke(op)}{\longrightarrow}$$
$$(invoked(op), q_{j+1}) \overset{do(op)}{\longrightarrow} \ldots \overset{do(op)}{\longrightarrow} (done(1), q_1) \overset{respond(1)}{\longrightarrow} (idle, q_1).$$

However, H is not an interpretation of any execution of A since the responses to op must eventually be 0. □

Very often, implementation automata are difficult to analyze for correctness. As we will shortly demonstrate in Lemma 4.3.7, if we can show that the set of inter-

preted histories – also called *traces* – produced by our implementation automaton is a subset of the set of traces produced by the abstract linearization automaton, then we would be certain that our program automaton meets the linearizability criterion defined by the latter automaton.

This relationship, called *trace inclusion*, can be shown to hold by means of *simulations* between two automata, A and B. If $\mathcal{H}_A \subseteq \mathcal{H}_B$, then we write $A \leq_T B$.

Lemma 4.3.7. *Let B be the abstract linearization automaton of an ADT $\mathcal{D}$. Let A be an automaton. Then A is a linearizable implementation of $\mathcal{D}$ if and only if $A \leq_T B$.*

Proof. Suppose that A is linearizable and let H be an interpretation of A. We must show H is also an interpretation of B.

Let $\mathcal{S}_D$ be the sequential specification of $\mathcal{D}$. By Definition 4.3.2, there is some sequential history $S \in \mathcal{S}_\mathcal{D}$ such that S is equivalent to complete(H') for some extension H' of H. We use S to construct an execution E of B with $\Gamma(E) = H$.

Since S is a sequential history, by Definition 4.1.2, there is some sequence $\mathcal{Q} = q_0 q_1 \ldots$ of states of $\mathcal{D}$ and

$$S = (op_1, rsp_1, p_1)(op_2, rsp_2, p_2)\ldots$$

where, for each $i \geq 0$, $(rsp_i, q_i) \in \delta(q_{i-1}, op_i)$, and q_0 is the start state of $\mathcal{D}$. Let

$$Pre(H) = \{H_i \mid i \geq 0, H_i \text{ is the prefix of } H \text{ with length } i\}.$$

We will show, by induction over m, that $H_m \in \mathcal{H}_B$.

For $m \in \mathbb{N}$, let S_m be the shortest prefix of S that contains all completed operation instances in H_m; S_m exists since S contains all completed operation instances of H. Let $I(m)$ be the following claim:

There is an execution E of B such that:

- $\Gamma(E) = H_m$.
- The state of B following E is $(s_1, s_2, \ldots, s_n, q)$ where, for each $1 \leq i \leq n$:
  - s_i is *idle* only if p_i has no pending operation instance in H_m.
  - s_i is *invoked*(op) only if p_i has operation instance op pending in H_m but there is no triple corresponding to op in S_m.
  - s_i is *done*(rsp) only if p_i has an operation instance op pending in H_m and the operation instance has a corresponding triple in S_m with response rsp.
  - q is the state of $\mathcal{D}$ after S_m.

$H_0 = \varepsilon$, the empty interpretation, is an interpretation of B, since the execution $(idle, \ldots, idle, q_0)$ is an execution of B satisfying the claim $I(0)$. Choose $m \geq 0$ and suppose that $I(m)$ is true, let E be an execution that satisfies $I(m)$, and let the state of B following E be $(s_1, s_2, \ldots, s_n, q)$. Let a be the last action of H_{m+1}. We consider the following cases:

- a is $invoke(op, p_i)$ for some operation $op \in OPS$ and some process $p_i \in \mathcal{P}$. Since H_m is the interpretation of a well-formed execution, p_i has no pending operation instance in H_m. Therefore, by the claim, s_i is $idle$ in E and

$$(\ldots, idle, \ldots, q) \overset{invoke(op,p_i)}{\longrightarrow} (\ldots, invoked(op), \ldots, q)$$

  is a valid extension of E by Definition 4.3.4, which, since $S_{m+1} = S_m$, satisfies $I(m+1)$.

- a is $respond(r, p_i)$ for some response $r \in RSP$, and process $p_i \in \mathcal{P}$. Since H_{m+1} is the interpretation of a well-formed execution, a corresponds to some previously invoked operation instance o (this is uniquely defined by Definition 4.3.1). Since H_m is an interpretation of a well-formed execution and $H_{m+1} = H_m respond(r, p_i)$, the last external action by p_i in H_m is $invoke(o, p_i)$. Let $(op_1, rsp_1, p_{i_1}), (op_2, rsp_2, p_{i_2}) \ldots (op_k, rsp_k, p_{i_k})$ be the sequence of triples such that

$$S_{m+1} = S_m(op_1, rsp_1, p_{i_1})(op_2, rsp_2, p_{i_2}) \ldots (op_k, rsp_k, p_{i_k}).$$

  If $k = 0$ then, by the induction hypothesis, p_i has o pending in H_m, the triple corresponding to o is in S_m, and $s_i = done(rsp)$. Therefore,

$$(\ldots, done(rsp), \ldots, q) \overset{respond(rsp,p_i)}{\longrightarrow} (\ldots, idle, \ldots, q)$$

is a valid extension of E by Definition 4.3.4, which, since $S_{m+1} = S_m$, satisfies $I(m+1)$.

If $k > 0$ then a is the response corresponding to the triple (op_k, rsp_k, p_{i_k}) since S_{m+1} is the shortest prefix of S that contains the completed operation instance o with response rsp.

By Definition 4.3.4, for all $1 \leq i \leq n$, s_i is either $idle$, $invoked(op)$ for some $op \in OPS$, or $respond(rsp)$ for some $rsp \in RSP$.

Let us examine the state of a process p_{i_j} from a triple (op_j, rsp_j, p_{i_j}) at the terminal state of execution E.

- If $s_{i_j} = idle$, then by the induction hypothesis, p_{i_j} has no pending operation instances in H_m. By the real-time ordering condition of Definition 4.3.2 on S, we know that op_j cannot begin after op_k ends. Thus, op_j is invoked in H_m. This means that op_j responds in H_m. By definition of S_m, $(op_j, rsp_j, p_{i_j}) \in S_m$. This contradicts the fact that $(op_j, rsp_j, p_{i_j}) \notin S_m$. Therefore, $s_{i_j} \neq idle$.
- If $s_{i_j} = done(rsp)$, then by the induction hypothesis, the operation instance referred to by the triple (op_j, rsp_j, p_{i_j}) is pending in H_m and $(op_j, rsp_j, p_{i_j}) \in S_m$. This contradicts the fact that $(op_j, rsp_j, p_{i_j}) \notin S_m$. Therefore, $s_{i_j} \neq done(rsp)$.

This shows that, for all $1 \leq j \leq k$, s_{i_j} must be $invoked(op)$ for some $op \in OPS$. By the real-time ordering condition of Definition 4.3.2, the operation instance op_j is the last operation instance invoked by p_{i_j} in H_{m+1}. Since the last action of H_{m+1} is not an invocation, op_j is the last operation instance invoked by p_{i_j} in H_m. This shows that $op = op_j$.

Consider the following extension E' of E:

$$(\ldots, invoked(op_1), \ldots, q) \overset{do(op_1, p_{i_1})}{\longrightarrow} (\ldots, done(rsp_1), \ldots, q_1)$$

$$(\ldots, invoked(op_2), \ldots, q_1) \overset{do(op_2, p_{i_2})}{\longrightarrow} (\ldots, done(rsp_2), \ldots, q_2)$$

$$\vdots$$

$$(\ldots, invoked(op_k), \ldots, q_{k-1}) \overset{do(op_k, p_{i_k})}{\longrightarrow} (\ldots, done(rsp_k), \ldots, q_k)$$

$$(\ldots, done(rsp), \ldots, q_k) \overset{respond(r, p_i)}{\longrightarrow} (\ldots, idle, \ldots, q_k).$$

The 'do' portion of this extension is legal since, by the induction hypothesis, q is a state of $\mathcal{D}$ that can be reached after S_m. Since S_{m+1} is in the sequential specification of $\mathcal{D}$ by Definition 4.3.2, we have that $(q_j, rsp_j) \in \delta(q_{j-1}, op_j)$ for $1 \leq j \leq k$, with $q_0 := q$. Therefore, q_k is a state of is a state of $\mathcal{D}$ that can be reached after S_{m+1}. The 'respond' step is legal since $p_i = p_{i_k}$ and s_i is $done(rsp)$ just before the last step of E'.

We have that:

– $\Gamma(E') = H_{m+1}$.

– Let $t' = (s'_1, s'_2, \ldots, s'_n, q')$ and $t = (s_1, s_2, \ldots, s_n, q)$ be the states of B following E' and E, respectively. Note that $s'_i = idle$ by construction of E' and that, by Definition 4.3.1, there are no pending operation instances by p_i in H_{m+1}. For $j \neq i$:

  * If s'_j is $idle$, then s_j is $idle$ since there are no respond actions for process p_j between t and t'. By $I(m)$, p_j has no pending operation instances in H_m. By Definition 4.3.1, p_j has no pending operation instance in H_{m+1} since the last step of H_{m+1} is a response.

  * If s'_j is $invoked(op)$ for some $op \in OPS$, then s_j is $invoked(op)$ since there are no invocations between t and t'. By $I(m)$, p_j has operation instance op pending in H_m but without a corresponding triple in S_m. By construction of E', the sequence of do operations exactly matches $S_{m+1} \setminus S_m$. Since that sequence does not include $do(op_j, p_j)$, there is no triple corresponding to this operation instance in S_{m+1}. Since A satisfies Definition 4.3.1 and $H_{m+1} = H_m respond(rsp, p_i)$, p_j has a pending operation instance in H_{m+1}.

  * If $s'_j = done(rsp)$ for some response $rsp \in RSP$, then, by construction of E', $s_j = done(rsp)$ or $s_j = invoked(op)$ for some $op \in OPS$

and the transition

$$(\ldots, \mathit{invoked}(op), \ldots, q) \xrightarrow{do(op,p_j)} (\ldots, \mathit{done}(rsp), \ldots, q')$$

is a step between t and t'. We consider the two cases:

- Suppose $s_j = \mathit{done}(rsp)$. By $I(m)$, p_j has operation instance op pending in H_m and the operation instance has a corresponding triple in S_m with response rsp. Then op is pending in H_{m+1} since A satisfies Definition 4.3.1 and $H_{m+1} \setminus H_m$ is a response by $p_i \neq p_j$. So, op has a corresponding triple in S_{m+1} since S_m is a prefix of S_{m+1}.
- Suppose $s_j = \mathit{invoked}(op)$. By $I(m)$, p_j has operation instance op pending in H_m but there is no corresponding triple in S_m. By construction of E', the sequence of do operations exactly matches $S_{m+1} \setminus S_m$. Since that sequence includes $do(op, p_j)$, there is a triple corresponding to this operation instance in S_{m+1}. Since the only response action from t to t' is by $p_i \neq p_j$, op is still pending in H_{m+1}.

By induction, for all $m \in \mathbb{N}$, $I(m)$ is true, therefore $H_m \in \mathcal{H}_B$. By Lemma 4.3.5, $H \in \mathcal{H}_B$.

Suppose that $A \leq_T B$. Let H be an interpretation of A. Since H is also an

interpretation of B, there is an execution E of B such that $\Gamma(E) = H$.

The projection of E onto *do* operations yields a sequential history that is equivalent to complete(H') for some extension H' of H as follows:

By Definition 4.3.4, the only allowable transitions for a process p are

$$idle \overset{invoke(op,p)}{\longrightarrow} invoked(op) \overset{do(op,p)}{\longrightarrow} done(rsp) \overset{respond(rsp,p)}{\longrightarrow} idle.$$

Hence, for every $do(op, p)$ in E, there is a uniquely defined process p and response rsp associated with $do(op)$. Hence, for every *do* transition in E, there is a uniquely defined triple (op, rsp, p). Projecting E onto *do* operations and replacing each element in the sequence with its associated triple yields a well-defined sequence

$$J = (op_1, rsp_1, p_{i_1}), (op_2, rsp_2, p_{i_2}), \ldots$$

By Definition 4.3.4, there is a sequence $q_0, q_1, \ldots$ – given by execution E – such that, for each $i \geq 1$, $(rsp_i, q_i) \in \delta(q_{i-1}, op_i)$. Therefore, J is a sequential history of $\mathcal{D}$.

By construction, every operation instance that is complete in H appears in J – this subsequence of J corresponds to H' in the first bullet of Definition 4.3.2. Pending operation instances in H that have *do* operations in E are present in J, while those that do not are absent from J – the sequence J corresponds to complete(H') of Definition 4.3.2. Therefore, J satisfies the first condition of Definition 4.3.2.

From the transition diagram above, we see that if rsp_j precedes op_i as an action

of B, then do_j precedes do_i as an action of B, since, according to Definition 4.3.4, every do operation by p_i is preceding and succeeded by an invocation and response, respectively. □

4.4 Forward and Forward-Backward Simulations

We now consider simulations as a technique for proving trace inclusion (and hence linearizability). In our discussion, A will be the implementation of the abstract data type $\mathcal{D}$, while B will be the corresponding abstract linearization automaton for $\mathcal{D}$ with the same set of processes. We use the following notation:

- $f[s] = \{x \mid x \in \mathit{states}(B) \wedge (s, x) \in f\}$
- $u' \stackrel{\hat{a}}{\Longrightarrow}_B u$ means the sequence of actions $\hat{a}$ can take the automaton B from state u' to state u.

Definition 4.4.1. (LV95) A *forward simulation* from an automaton A to B is a relation f over $\mathit{states}(A)$ and $\mathit{states}(B)$ that satisfies:

- If $s \in \mathit{start}(A)$, then $f[s] \cap \mathit{start}(B) \neq \emptyset$.
- If $s' \stackrel{a}{\longrightarrow}_A s$ and $u' \in f[s']$, then there exists a state $u \in f[s]$ and a sequence of steps $\hat{a}$ that has the same external actions as a such that $u' \stackrel{\hat{a}}{\Longrightarrow}_B u$.

We write $A \leq_F B$ if there exists a forward simulation from A to B.

A result from (LV95) demonstrates that if $A \leq_F B$, then $A \leq_T B$. However, the reverse is not necessarily true; in fact, some implementations that satisfy trace inclusion do not have a forward simulation to their corresponding abstract linearization automata – we shall see an example of this in Counterexample 5.2.1. We need the following notion of simulation:

Definition 4.4.2. (LV95) An *image-finite backward simulation* from an automaton A to B is an image finite relation f over $states(A)$ and $states(B)$ (i.e., $|f[s]| < \infty$ for all $s \in states(A)$) that satisfies:

- If $s \in start(A)$, then $f[s] \subseteq start(B)$.
- If $s' \xrightarrow{a}_A s$ and $u \in f[s]$, then there exists a state $u' \in f[s']$ and a sequence of steps $\hat{a}$ that has the same external actions as a such that $u' \stackrel{\hat{a}}{\Longrightarrow}_B u$.

We write $A \leq_{iB} B$ if there exists an image-finite backward simulation from A to B.

We say there there is an image-set finite forward-backward simulation in between A and B if there is some intermediate automaton C such that $A \leq_F C \leq_{iB} B$. We write $A \leq_{iFB} B$ if there exists an image-set finite forward-backward simulation from A to B.

This notion of simulation perfectly captures what we require, since another

result from (LV95) shows that, if B has finite non-determinism, then $A \leq_T B$ if and only if $A \leq_{iFB} B$.

5 Equivalences between Linearizability and Simulation

We now have all of the concepts we need to express our results.

5.1 Linearizability and Forward-Backward Simulations

The following result is common lore among those in the distributed computing community interested in linearizability (see (Lyn96)); for completeness, we provide a formal proof of this equivalence result.

Theorem 5.1.1. *Let B be the abstract linearization automaton of an ADT $\mathcal{D}$. Let A be an automaton. The following are equivalent:*

1. *A is a linearizable implementation of $\mathcal{D}$.*
2. *$A \leq_T B$.*
3. *$A \leq_{iFB} B$.*

Proof. (1) $\Leftrightarrow$ (2) is proved in Lemma 4.3.7. (2) $\Rightarrow$ (3) and (3) $\Rightarrow$ (2) are Theorems 5.6 and 5.5 of (LV95), respectively. □

Forward-backward simulations are generally more difficult to work with than the more straightforward forward simulations. This is because one must find a suitable intermediate automaton such that there is a forward simulation from the implementation to the intermediate automaton, and a backward simulation from the intermediate automaton to the abstract linearization automaton (CGLM06). Constructing such an automaton and its backward simulation is often more challenging than constructing a more intuitive forward simulation.

5.2 Strong Linearizability and Forward Simulations

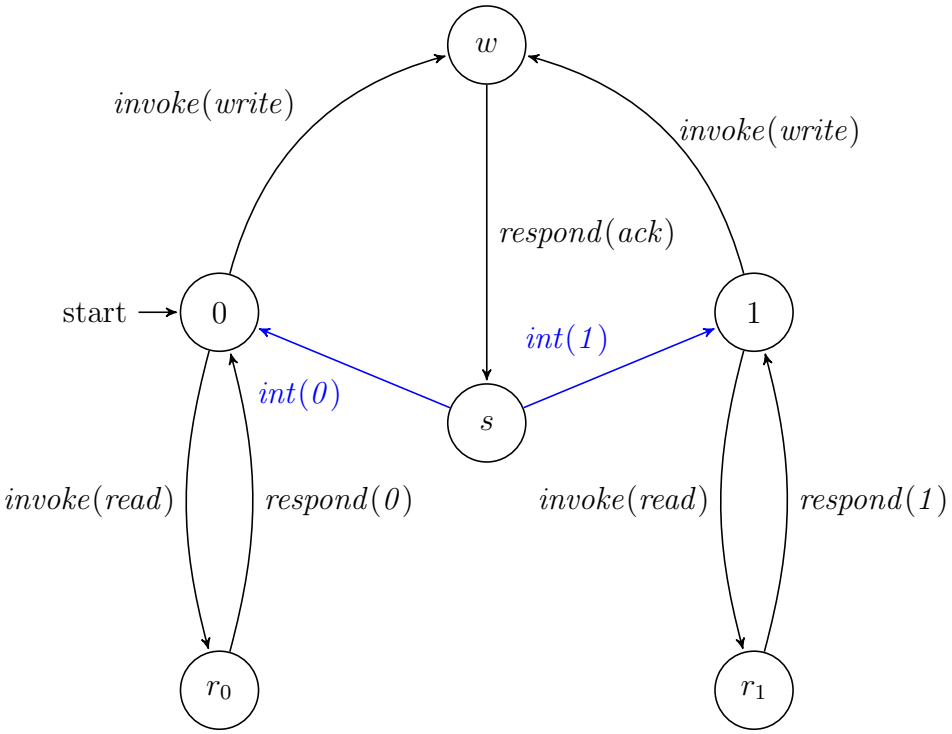
It appears plausible that a forward simulation exists between an implementation and an abstract linearization automaton of $\mathcal{D}$ if and only if that implementation is strongly linearizable, since once a linearization decision is made in some prefix of an execution, it is not changed in any continuation of the execution. However, this is not the case for some *non-deterministic* abstract data types, as the following counterexample will show.

Counter Example 5.2.1. Consider a binary register R initialized to 0 and supporting two operations:

- *write* which non-deterministically writes 0 or 1 to R, i.e., $\delta(\_, write) = \{(ack, 0), (ack, 1)\}$.

- *read* which non-destructively reads the value stored in R, i.e., $\delta(i, read) = \{(i, i)\}$, for $i \in \{0, 1\}$.

Let B be the abstract linearization automaton of R for a single process and consider the following automaton A that implements R for one process:



The automaton A illustrated is an implementation of R for one process since it conforms to Definition 4.3.1: all executions are well-formed, as the reader can verify by tracing the illustrated transitions that no operation instance can be invoked while

another is ongoing. The non-determinism of the **write** operation can be seen by the transitions (depicted as blue lines) from the state s.

Furthermore, it does not "show its hand" to an observer before the next **read** operation since the non-deterministic transition is an internal action. This peculiarity is the reason for the non-existence of a forward simulation from A to B. Note also that A is strongly linearizable, since it is an implementation for a single process, and the linearization formed by taking the instances of the operations in the order in which they appear is prefix-preserving.

Lemma 5.2.2. *There is no forward simulation from the implementation A of Counterexample 5.2.1 to R's abstract linearization automaton B for a single process.*

Proof. To derive a contradiction, suppose that f is a forward simulation from A to B. By Definition 4.3.4 and R's specification, $start(B) = \{(idle, 0)\}$. Therefore, by Definition 4.4.1, $(idle, 0) \in f[0]$. Consider the path $0ws$ in A, which has external actions $invoke(write), respond(ack)$. Since f is a forward simulation, there is some state $u \in f[s]$ such that

$$(idle, 0) \overset{invoke(write,p),do(write,p),respond(ack,p)}{\Longrightarrow_B} u.$$

By Definition 4.3.4 and R's specification, $u = (idle, 0)$ or $u = (idle, 1)$, i.e., $(idle, 0) \in f[s]$ or $(idle, 1) \in f[s]$.

Suppose $(idle, 0) \in f[s]$. Consider the path $s1r_11$, which has external actions

$invoke(read), respond(1)$; there is some state $v \in f[1]$ such that

$$(idle, 0) \overset{invoke(read,p),do(read,p),respond(1,p)}{\Longrightarrow_B} v.$$

There is no state $v \in states(B)$ satisfying this property since any call to *read* from $(idle, 0)$ will return 0. Therefore, $(idle, 0) \notin f[s]$. A similar argument using the path $s0r_00$ shows that $(idle, 1) \notin f[s]$. This is a contradiction, and hence there is no forward simulation from A to B. □

If we restrict our consideration to *deterministic* abstract data types, we can show a correspondence between strong linearizability and forward simulations. We now present our main result for automata and simulations:

Theorem 5.2.3. *Let B be the abstract linearization automaton of a deterministic ADT $\mathcal{D}$ over a set of processes $\mathcal{P}$. Let A be an automaton. The following are equivalent:*

1. *A is a strongly linearizable implementation of $\mathcal{D}$ for $\mathcal{P}$.*

2. *$A \leq_F B$.*

Proof. We first prove that (1) ⇒ (2). Let $g : \mathcal{E}_A \longrightarrow \mathcal{S}_\mathcal{D}$ be a strong linearization function from the set of executions of A to the sequential specification of $\mathcal{D}$, given by Definition 4.3.3.

For $s \in states(A)$, let

$$\mathcal{E}_s := \{E \in \mathcal{E}_A \mid E \text{ terminates in state } s\}.$$

We define the relation $f \subseteq states(A) \times states(B)$ as follows. For each $s \in states(A)$, let $f[s]$ be the set of all states $(r_1, r_2, \ldots, r_n, q)$ such that $\exists\ E \in \mathcal{E}_s$ such that:

(i) $\forall\ 1 \leq i \leq n$, r_i is *idle* only if there is no pending operation instance by p_i in $\Gamma(E)$.

(ii) $\forall\ 1 \leq i \leq n$, r_i is $invoked(op)$ only if p_i has operation instance op pending in $\Gamma(E)$ but there is no triple corresponding to op in $g(E)$.

(iii) $\forall\ 1 \leq i \leq n$, r_i is $done(rsp)$ only if p_i has an operation instance op pending in $\Gamma(E)$ and the operation instance has a corresponding triple in $g(E)$ with response rsp.

(iv) q is the state of $\mathcal{D}$ after applying $g(E)$ to q_0, the start state of $\mathcal{D}$.

Note that since $\mathcal{D}$ is a deterministic ADT, the state q is uniquely determined. We show that f is a forward simulation from A to B.

Suppose $s_0 \in start(A)$. The execution $E = s_0$ terminates in s_0. Therefore $start(B) = \{(idle, \ldots, idle, q_0)\} \subseteq f[s_0]$. This shows that $f[s_0] \cap start(B) \neq \emptyset$.

Consider the transition $s' \xrightarrow{a} s$ in A and let $u' \in f[s']$. By definition of f, $u' = (r'_1, \ldots, r'_n, q') \in states(B)$ and there is some execution E' terminating in s'

such that u' and E' satisfy (i) - (iv). Let E be the execution $E'as$. Since g is prefix-preserving, $g(E')$ is a prefix of $g(E)$. Let

$$g(E) = g(E')(op_1, rsp_1, p_{i_1})(op_2, rsp_2, p_{i_2}) \ldots (op_k, rsp_k, p_{i_k}).$$

Since u' satisfies property (iii), none of $r'_{i_1}, r'_{i_2}, \ldots, r'_{i_k}$ is $done(\_)$. To see why, suppose $r_{i_j} = done(\_)$ to derive a contradiction. Then p_{i_j} has a pending operation instance in $\Gamma(E')$ and there is a corresponding triple in $g(E')$ by (iii). Therefore,

$$\begin{aligned} \#\text{ of triples in } g(E') \text{ for process } p_{i_j} &\geq \#\text{ ops invoked by } p_{i_j} \text{ in } E' \\ &= \#\text{ ops invoked by } p_{i_j} \text{ in } E \end{aligned}$$

since $E = E' \cdot a$ and p_{i_j} has a pending operation instance at the end of E', and E' and E are well-formed (i.e., if a is done by p_{i_j} then it is a *respond* and, hence, not an *invoke*). Therefore, the number of triples in $g(E)$ for process p_{i_j} is strictly greater than the number of operation instances invoked by p_{i_j} in E, since $(op_j, rsp_j, p_{i_j}) \in g(E)$ but not in $g(E')$. This contradicts Definition 4.3.2.

This leaves only *idle* or *invoked*(op) for operation instance op pending in $\Gamma(E')$. We first observe that a process features at most once in the triples

$$(op_1, rsp_1, p_{i_1})(op_2, rsp_2, p_{i_2}) \ldots (op_k, rsp_k, p_{i_k}).$$

To see why, suppose that p_{i_j} features in $g(E)$ at least twice and that this is the first time this occurs in the execution for any process. Let T, T' be the number of triples

in $g(E), g(E')$ for process p_{i_j}, respectively. Let K, K' be the number of invocations by p_{i_j} in E, E', respectively.

We have $T \leq K$, since, by the first part of Definition 4.3.2, only operation instances that have been invoked can be linearized and $T' + 1 \geq K'$ since, also by the first part of Definition 4.3.2, for each process at most one operation instance can be excluded from a linearization – the last operation instance invoked, and only if it has not responded. Furthermore,

$$\begin{aligned} 2 + T' &\leq T && \text{by assumption} \\ &\leq K \\ &= K' \end{aligned}$$

since p_{i_j} is pending in E' and hence cannot invoke by well-formedness

$$\leq T' + 1.$$

This is a contradiction.

If $r'_{i_j} = \mathit{invoked}(op)$, then $op = op_j$. To see why, suppose that $op \neq op_j$ to derive a contradiction. By (ii), p_{i_j} is pending in $\Gamma(E')$. Since E' is well-formed, op is the last operation instance invoked by p_{i_j} in E'. Since p_{i_j} features in a triple in $g(E)$ but not in $g(E)$, it features exactly once by the observation above. By Definition 4.3.2, this triple is (op, rsp, p_{i_j}). But by construction, this triple is (op_j, rsp_j, p_{i_j}), a contradiction.

Case I: Suppose a is an internal action. Consider the path

$$u' \xrightarrow{do(op_1,p_{i_1})} \cdots \xrightarrow{do(op_k,p_{i_k})} u.$$

We show that this path P is a valid path for automaton B and that u and E satisfy (i)-(iv). By Definition 4.3.2, if $(op_j, rsp_j, p_{i_j}) \in g(E)$, then $invoke(op_j)$ is in $\Gamma(E)$. Since a is an internal action, $\Gamma(E') = \Gamma(E)$. Therefore, $invoke(op_j)$ is in $\Gamma(E')$. $respond(rsp_j)$ is not in $\Gamma(E')$ since, by Definition 4.3.2, the triple (op_j, rsp_j, p_{i_j}) would then be in $g(E')$. Since op_j is pending in $\Gamma(E')$ and $(op_j, rsp_j, p_{i_j}) \notin g(E')$, we have that $r'_{i_j} = invoked(op_j)$. So, P is valid.

1. r_i is *idle* if and only if r'_i is *idle*, since the path $u' \Longrightarrow u$ defined above has only *do* operations. If r_i is *idle* then by property (i), p_i has no pending operation instances in $\Gamma(E')$. Since $\Gamma(E) = \Gamma(E')$, p_i has no pending operation instance in $\Gamma(E)$.

   This shows that r_i is *idle* only if p_i has no pending operation instance in $\Gamma(E)$.

2. If r_i is $invoked(op)$, then since P contains no *invoke* actions it follows from Definition 4.3.4 that r'_i is $invoked(op)$. By property (ii), p_i has operation instance op pending in $\Gamma(E')$ but there is no triple corresponding to op in $g(E')$. Since $\Gamma(E) = \Gamma(E')$, p_i has op pending in $\Gamma(E)$. By construction of P, since p_i has no *do* transition in P, there is no triple

corresponding to op in $g(E)$.

This shows that p_i has operation instance op pending in $\Gamma(E)$ but there is no triple corresponding to op in $g(E)$.

3. If r_i is $done(rsp)$, then by the definition of P and Definition 4.3.4, r'_i is either $done(rsp)$ and there is no transition by p_i along P, or r'_i is $invoked(op)$ for some operation op and there is a transition $do(op, p_i)$ in P.

   - Suppose r'_i is $done(rsp)$. By property (iii), p_i has an operation instance op pending in $\Gamma(E')$ and the operation instance has a corresponding triple in $g(E')$ with response rsp. Since $\Gamma(E) = \Gamma(E')$, p_i has op pending in $\Gamma(E)$. Since $g(E')$ is a prefix of $g(E)$ and there is a triple corresponding to op in $g(E')$, there is a triple corresponding to op in $g(E)$.

     This shows that p_i has an operation instance op pending in $\Gamma(E)$ and the operation instance has a corresponding triple in $g(E)$ with response rsp.

   - Suppose r'_i is $invoked(op)$ for some operation op and there is a transition $do(op, p_i)$ in P. By property (iii), p_i has operation instance op pending in $\Gamma(E')$ but there is no triple corresponding to op in $g(E')$. By construction of P, there is a triple corresponding to op in

$g(E)$ – specifically, our previous discussion shows that the operation op is the same in $\Gamma(E)$ and P; by property (iv), B and $\mathcal{D}$ are in the same state q; and, by Definition 4.3.4 and determinism of δ, the responses to op are the same in the response to $do(op, p_i)$ and $g(E)$. This triple is in $g(E) \setminus g(E')$. Since $\Gamma(E) = \Gamma(E')$, p_i has op pending in $\Gamma(E)$.

This shows that p_i has an operation instance op pending in $\Gamma(E)$ and the operation instance has a corresponding triple in $g(E)$ with response rsp.

4. Since $u' \in f[s']$, q' is the state of $\mathcal{D}$ after applying $g(E')$ to q_0, the start state of $\mathcal{D}$. Applying the do transitions of P to q' yields q, the state variable in the tuple u. By construction of P and property (iv), q is the state of $\mathcal{D}$ after applying $g(E)$ to q_0.

Case II: Suppose a is an $invoke(op, p)$ action for some $op \in OPS$ by $p \in \mathcal{P}$, and consider the path

$$u' \xrightarrow{invoke(op,p)} u'' \xrightarrow{do(op_1,p_{i_1})} \cdots \xrightarrow{do(op_k,p_{i_k})} u.$$

We show that this path is valid and that u and E satisfy (i)-(iv). We first show that the $u' \xrightarrow{invoke(op,p)} u''$ is a valid transition. Since $E = E'$ $invoke(op, p)$ s

is an execution produced by an implementation of $\mathcal{D}$, E is well-formed by Definition 4.3.1. Therefore, p has no pending operation instance in $\Gamma(E')$. By property (i), p's state r in u' is *idle*. Therefore, the transition $u' \xrightarrow{invoke(op,p)} u''$ is valid. The validity of the remainder of the path and that $u \in f[s]$ is given by an argument similar that given for the case where a is an internal action.

Case III: Suppose a is a $respond(rsp, p)$ action for some $rsp \in RSP$ and $p \in \mathcal{P}$, consider the path

$$u' \xrightarrow{do(op_1,p_1)} \cdots \xrightarrow{do(op_k,p_k)} u'' \xrightarrow{respond(rsp)} u.$$

The validity of the of the path from u' to u'' is given by an argument similar to that given for the case where a is an internal action. Since $E = E'\ respond(rsp, p)\ s$ is a well-formed execution, p has a pending operation instance in $\Gamma(E')$. Furthermore, since g is a linearization function, by Definition 4.3.2, the triple $(op, rsp, p) \in g(E)$. By properties (ii) and (iii), p is in state $invoked(op)$ or $done(rsp)$, if $(op, rsp, p) \in g(E')$ or $g(E) \setminus g(E')$, respectively – i.e., if p is one of $p_1, p_2, \ldots, p_k$ or not.

By construction of the path segment from u' to u'', the state of p in $u'\prime$ is $done(rsp)$. Therefore, the transition u'' to u is valid, and since p has no pending operation instance in $\Gamma(E)$ and its state in u is *idle*, (i) is satisfied

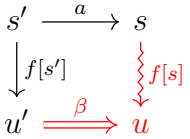
by u. That the other conditions are satisfied by u follows from the argument on the u' to u'' path segment.

Since u and E satisfy (i)-(iv), we have shown that $u \in f[s]$. Therefore, $\forall\ u' \in f[s']$, $\exists\ u \in f[s]$ such that $u' \stackrel{\hat{a}}{\Longrightarrow} u$, which shows that f is a forward simulation from A to B.

We now prove that (2) $\Rightarrow$ (1). Let f be a forward simulation from A to B. We construct a prefix-preserving linearization function $g : \mathcal{E}_A \longrightarrow \mathcal{S}_{\mathcal{D}}$ using f. We see that the state $(idle, idle, \ldots, idle, q_0) \in f[s_0]$ since $(idle, idle, \ldots, idle, q_0)$ is the only element of the set $start(B)$ and $f[s_0] \cap start(B) \neq \emptyset$ by Definition 4.4.1.

As we define g, we will associate with every execution E of A a state $u(E) \in states(B)$ such that $u(E) \in f[s]$, where s is E's terminal state. Since f is a relation, there is non-determinism in the choice of $u(E)$ that satisfies Definition 4.4.1, and we wish to resolve that non-determinism locally as we define g.

We first define $g(s_0) := \varepsilon$, where ε denotes the empty sequence, and associate $u(s_0) = (idle, idle, \ldots, idle, q_0)$ with s_0. Consider an execution $E' = s_0 a_1 s_1 \ldots s'$ for which $g(E')$ is already defined, with $u(E') \in f[s']$ associated with E', and let $E = E'as$.



By Definition 4.4.1, there is some $u \in f[s]$ and some sequence β with $u' \stackrel{\beta}{\Longrightarrow} u$ such

that $\hat{\beta} = \hat{a}$, where $\hat{\beta}$ denotes the subsequence of external actions of β. There may be several such states, hence their depiction using red in the diagram. Choose one such state u and call it $u(E)$. Let

$$g(E) := g(E')(op_1, rsp_1, p_1)(op_2, rsp_2, p_2)\dots(op_k, rsp_k, p_k)$$

where (op_i, rsp_i, p_i) is the triple associated with the ith *do* operation in β as follows: By Definition 4.3.4, every $do(op, p)$ operation is preceded by a unique $invoke(op, p)$ operation and has a well defined response $rsp \in RSP$ since δ is deterministic. In this way, a $do(op, p)$ operation is associated uniquely with a corresponding (op, rsp, p) triple. This completes the inductive definition.

By definition, g is prefix-preserving. Choose $m \in \mathbb{N}$, g has the following properties for all executions E with $|E| \leq m$ (counted by number of transitions):

Let $E = s_0 a_1 s_1 \dots a_m s_m$ and $g(E) = (op_1, rsp_1, p_1)\dots(op_k, rsp_k, p_k)$. There is an execution $u_0\beta_1 u_1 \dots \beta_m u_m$ of automaton B where, for $1 \leq i \leq m$, β_i denotes a group of actions such that

(i) $u_i \in f[s_i]$ for all $0 \leq i \leq m$.

(ii) $\hat{\beta}_i = \hat{a}_i$ for all $1 \leq i \leq m$.

(iii) Projecting $\beta_1\beta_2\dots\beta_m$ onto *do* operations produces a subsequence that yields a sequence of triples with $(op_1, rsp_1, p_1)\dots(op_k, rsp_k, p_k)$ where rsp_1 is the

response in the pair $(q_1, rsp_1) = \delta(q, op_1)$, where q is the state variable at u_0, and rsp_i is the response in the pair $\delta(q_{i-1}, op_i)$ for $1 < i \leq m$.

We can extend the domain of executions to include non-terminating executions. Let E be a non-terminating execution and, for all $m \in \mathbb{N}$, let E_m be the prefix of E of length m. Define

$$g(E) := \lim_{m \to \infty} g(E_m).$$

Since g is prefix-preserving, the above limit is well-defined. We now show that g is a linearization function for any execution E of A. By condition (ii) and Definition 4.3.4, the first condition of Definition 4.3.2 is satisfied for some extension of $complete\{\Gamma(E)\}$ – namely, the extension where invoked operation instance in $\Gamma(E)$ without a corresponding *do* operation are removed, and those with a corresponding *do* but without a response are completed with the response in their corresponding triple in $g(E)$. Suppose that $op_i <_{\Gamma(E)} op_j$. Then, by Definition 4.3.4,

$$do(op_i, p_i) < rsp(op_i, p_i) < invoke(op_j, p_j) < do(op_j, p_j).$$

Therefore, by construction of g

$$(op_i, rsp_i, p_i) <_{g(E)} (op_j, rsp_j, p_j).$$

This shows that g satisfies the second property of Definition 4.3.2 – hence, g is a linearization function. Since g is a prefix-preserving linearization function, by Definition 4.3.3, it is a strong linearization function. □

6 Application of Equivalences to Implementation of Set ADT

These equivalences allow us to use abstract linearization automata and simulations to verify the linearizability or strong linearizability of algorithms of interest, by modeling implementations of those algorithms as automata in a straightforward way. If simulations between these algorithms and the abstract linearization automata of the ADT's they implement can be shown to exist, then we can be assured that the algorithms in question have the corresponding correctness properties.

We can also use Theorem 5.2.3 to shed new light on old algorithms. For example, we show that there is a forward simulation from an automaton representing the singly-linked list algorithm due to (HHL$^+$06) implementing a set data type and presented in (CGLM06) and the abstract linearization automaton for that algorithm – contradicting the claim in (CGLM06) that there is not. Theorem 5.2.3 allows us to do this by showing that the algorithm is strongly linearizable.

In this linked-list representation, the elements of the set are stored as nodes

Algorithm 11: *contains*(k)

```
1 curr := Head;
2 while curr.key < k do
3 |   curr := curr.next;
4 end
5 if curr.key = k && ¬curr.marked then
6 |   return true
7 else
8 |   return false
9 end
```

in shared memory with four fields per node: *key* – the value, a *next* pointer, a boolean *marked*, and a lock *lock* – where a lock is an object with atomic operations lock() and unlock() where once a process p performs lock(), it is the only process permitted to perform any operations on the node until it performs unlock().

The sentinel node *Head* always has $Head.key = -\infty$ and $Head.marked = false$, while *Tail* has $Tail.key = \infty$, $Tail.marked = false$, and $Tail.next = Tail$.

We give the pseudocode for the *contains*(k), *locate*(k), *remove*(k), and *add*(k) from (CGLM06).

Colvin *et al.* give linearization points for all operations except a *contains*(k)

Algorithm 12: *locate*(k)

```
1  while true do
2      pred := Head;
3      curr := pred.next;
4      while curr.key < k do
5          pred := curr;
6          curr := curr.next;
7      end
8      pred.lock();
9      curr.lock();
10     if ¬pred.marked && ¬curr.marked && pred.next = curr then
11         return pred, curr
12     else
13         pred.unlock();
14         curr.unlock();
15     end
16 end
```

Algorithm 13: *remove*(k)

```
1  pred, curr := locate(k);
2  if curr.key = k then
3  |   curr.marked := true;
4  |   entry := curr.next;
5  |   pred.next := entry;
6  |   res := true;
7  else
8  |   res := false;
9  end
10 pred.unlock();
11 curr.unlock();
12 return res
```

Algorithm 14: *add*(k)

```
1  pred, curr := locate(k);
2  if curr.key != k then
3      entry := new Entry();
4      entry.key := k;
5      entry.next := curr;
6      pred.next := entry;
7      res := true;
8  else
9      res := false;
10 end
11 pred.unlock();
12 curr.unlock();
13 return res
```

operation that returns the result **false**. They are:

- Line 5 of Algorithm 11 for a successful *contains*(k) operation.
- Line 3 of Algorithm 13 for a successful *remove*(k) operation and line 8 of Algorithm 13 for an unsuccessful *remove*(k) operation.
- Line 6 of Algorithm 14 for a successful *add*(k) operation and line 9 of Algorithm 14 for an unsuccessful *add*(k) operation.

Moreover, the paper outlines the following invariants:

1. The list is strictly sorted by key, i.e., $Head.key < k_1 < k_2 < ... < Tail.key$.
2. The set of keys represented is exactly the set of keys in unmarked nodes that can be reached by following *next* pointers from the *Head* node.
3. The node keys are immutable.

The authors claim that an unsuccessful *contains* operation cannot be linearized by giving a step in *contains*'s pseudocode – at any proposed step, another operation may have added the key to the set. They go on to note

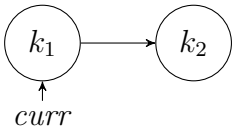
> For some algorithms and their specifications, however, there is no way to define such a forward simulation [from an implementation to an abstract linearization automaton] because for some action of [implementation] C, the actions of [the abstract linearization automaton] A that we should choose depend on future actions (i.e., actions that appear later in the execution). *As we explain later, LazyList is one such algorithm.*[Emphasis added.]

This is not, strictly speaking, true. Although it is not possible to give a linearization point in the *contains* pseudocode for the reasons the authors state, it *is* possible to linearize a failed *contains* operation instance without foreknowledge of the future. We discuss two linearization strategies for doing this, both of which linearize active *contains* operation instances at the linearization points of other operation instances – in one case a *remove* operation, and in the other an *add* operation – without requiring any more knowledge than the current state of the execution.

6.1 Linearizing *contains* Using *remove*

From examining Algorithm 11, we see that *contains*(k) returns **false** if and only if $curr.key \neq k$ or $curr.marked$ after the while loop on line 2 has concluded.

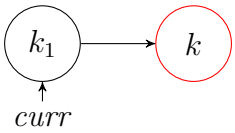
Consider the following illustrations of entries in a singly-linked list. We use a red border to denote a node that has been marked for deletion by line 3 of Algorithm 13. Suppose that *curr* is the pointer owned by a process p executing *contains*(k), with $k_1 < k < k_2$.



(6.1)

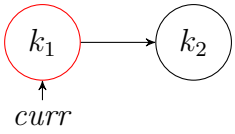
In this case, if p executes line 3 of Algorithm 11 and transitions from k_1 to k_2, then *contains*(k) can be linearized at that step, since the list does not contain k due to

the list-ordering invariant.



(6.2)

In this case, if p executes the *while* loop in Algorithm 11 and transitions from k_1 to k, then $contains(k)$ can linearize at that step, since k is absent from the set since the node is marked (logically deleted) and locked – hence no other process can add k to the list until the deletion is complete. Also, since the list is sorted, k does not occur elsewhere (marked or not) in any node accessible from *Head* by following *next* pointers.



(6.3)

The problem arises in this case, since some process can add k to the list without the *contains* operation (which is going to return **false**) being 'aware' of it. Unlike the *contains* operation, we, as linearizers, have a global view of the execution, and can define a precise point along it – without being clairvoyant – when the *contains* operation instance is linearized. To do so, we require the following concept.

Definition 6.1.1. Suppose $p \in \mathcal{P}$ is performing Algorithm 11. The *join* of p is the first unmarked node along the chain of *next* pointers from the node pointed to by $curr_p$, p's *curr* pointer.

Note that if $curr_p$ itself is unmarked, then $join(p) = curr_p$. The key idea of our

strategy is to linearize a *contains*(k) operation instance at the first moment where:

1. $join(p).key > k$ or,

2. the value **true** is read from the *marked* field of *curr* on line 5 of Algorithm 11.

This is a valid strategy by the following lemmas:

Lemma 6.1.2. *Suppose $p \in \mathcal{P}$ is performing contains(k). If contains(k) by p terminates then one of the above events must occur during the execution. If it returns* **false***, then 1 happens. If it returns* **true***, then 2 happens. Moreover, 1 and 2 cannot both occur during the same call to contains(k).*

Proof. From examining Algorithm 11, we see that it terminates by returning either **true** on line 6 or **false** on line 8. In both cases, the value returned is dependent on the test on line 5 of Algorithm 11.

We know that at the completion of the test on line 5 that $curr.key \geq k$. To see why, suppose $curr.key < k$ when the test on line 5 of Algorithm 11 is performed. By the key immutability invariant, p last set *curr* to a node with key less than k – i.e., p exited the loop on line 2 of Algorithm 11 with $curr < k$. But this violates the condition on line 2 of Algorithm 11. Therefore, $curr.key \geq k$ throughout the test on line 5. We consider the possibilities for the $curr.key$ and $curr.marked$ at the completion of the test on line 5:

- Suppose $curr.key = k$ and $\neg curr.marked$. Then the test on line 5 of Algorithm 11 was successful and event 2 occurred in the execution.

- Suppose $curr.key > k$ and $\neg curr.marked$. Then, by Definition 6.1.1, $join(p) = curr$ and $join(p).key > k$. This shows that event 1 occurred in the execution.

- Suppose $curr.key \geq k$ and $curr.marked$. Then, by Definition 6.1.1, $join(p)$ is the first unmarked node reachable from $curr$ by *next* pointers. By the list-ordering invariant, this is some node with key larger than $curr.key$ – i.e., $join(p).key > k$. This shows that event 1 occurred in the execution.

In all cases we have shown that one of the events occurs during the call. Therefore, the moment when the event *first* occurs is well-defined.

Lemma 6.1.3. *If $join(p).key > k$ during a call to contains, then $join(p).key > k$ for the remainder of that call.*

Proof. Suppose not. Let t be the first step in an execution where $join(p).key$ changes from a value greater than k to one less than or equal to k for some k.

Case I: $join(p) = curr_p$ immediately before t. The key-immutability condition implies that the only step t that can change $join(p)$ is marking $join(p)$ or advancing $curr_p$ to the next node. In both cases, the list-ordering condition implies that t does not change $join(p)$ to a node with key at most k since the keys are sorted in increasing order.

Case II: $join(p) \neq curr_p$ immediately before t. Then the chain of marked nodes from $curr_p$ to $join(p)$ (including $curr_p$) is non-empty. t cannot be advancing $curr_p$ since that would leave $join(p)$ unaltered. Similarly, $join(p).key$ cannot be changed by the key-immutability condition. Since marked nodes cannot be unmarked (there is no line of code that sets a *marked* field to **false**), the only remaining possibility is that t is an insertion of an unmarked node with key $k' \leq k$ in this chain between $curr_p$ and $join(p)$.

Let t be the insertion of a node with key $k' \leq k$ by process q – i.e., process q performs line 6 of Algorithm 14. The nodes $pred_q$ and $curr_q$ are returned on line 1 of Algorithm 14 by q calling $locate(k')$. Furthermore, by examining Algorithm 12 we see that $pred_q$ and $curr_q$ are locked and unmarked when they are returned to q. Since they are marked immediately before t, they must be marked at some step between line 1 of q's $add(k')$ and line 6 of that same call. This is impossible by the locking semantics.

This shows that no such step t exists. □

We now show that the events given on page 72 are mutually exclusive.

- If event 1 occurs first then at that moment $join(p).key > k$. By Definition 6.1.1 and the list-ordering invariant, subsequent to event 1's occurrence during that call to $contains(k)$, $join(p).key > k$ by Lemma 6.1.3. By examining the

test on line 5 of Algorithm 11, we see that it is true only if at the moment of its completion $join(p).key = k$. Therefore, event 2 never occurs subsequent to event 1.

- If event 2 occurs first then the call to $contains(k)$ terminates with $join(p).key = k$. Therefore, event 1 never occurs subsequent to event 2.

□

Lemma 6.1.4. *Let $Node = join(p)$ and $k = Node.key$ at some point in the execution. Then $remove(k)$ by q is the only operation by a process other than p that can change $join(p)$. Furthermore, $join(p)$ is changed by q only on line 3 of Algorithm 13.*

Proof. It is clear from Definition 6.1.1 that $join(p)$ is changed by a process other than p only when the *next* pointers of the data structure are altered or a node is marked. By examining the pseudocode, we see that nodes' *next* pointers are changed only on line 5 of Algorithm 13 and lines 5 and 6 of Algorithm 14. Marking occurs only on line 3 of Algorithm 13.

We argue that the *next* pointer changes do not change $join(p)$. Since *entry* is a new node created by Algorithm 14 that cannot be reached from any node in the list until q performs line 6 of Algorithm 14, changing $entry.next$ can have no effect on $join(p)$ until q performs line 6 of Algorithm 14. If $curr_p$ is pointing to an unmarked

node, then that node is the join and no alteration of the data structure (besides marking) will change $join(p)$. If $curr_p$ is marked, then consider the chain of marked nodes between $curr_p$ and $join_p$, including $curr_p$. Since $curr_p \neq join(p)$, this chain is non-empty. Let N be the first node along this chain where $N.next$ is altered by line 5 of Algorithm 13 or line 6 of Algorithm 14 by some process $q \neq p$. In both cases, $pred_q = N$ and $pred_q$ is set by q calling the subroutine *locate*.

By the test on line 10 of Algorithm 12, *pred* is not marked when it is returned by *locate* to q, since it is locked on line 8 of Algorithm 12 before it is returned. Therefore, it must have been marked at some point between q completing line 1 of Algorithm 14 (and setting $pred_q \leftarrow N$) and invoking line 6 of Algorithm 14. This is impossible by the locking semantics, since no other process can mark N until q relinquishes its hold on N on line 11 of Algorithm 14 that it obtained on line 8 of Algorithm 12. A similar argument holds for $pred_q$ if q is performing Algorithm 13. This shows that the *next* pointers along this chain are unaltered; therefore, $join(p)$ is unaltered by addition of nodes or physical removal of marked nodes.

A $remove(k)$ can change $join(p)$ by marking $join(p)$ (if $Node.key = k$). □

The following observations demonstrate that this is a valid linearization of $contains(k)$:

- If $join(p).key = k$ in the read of the *marked* field of *curr* on line 5 of Algorithm 11, then k is an unmarked node reachable from *Head* and is thus in the set.

- If $join(p).key$ is greater than k, then at least one of the conditions in the final execution line 5 of Algorithm 11 is falsified, hence the invocation will return **false**. Note that once $join(p)$ is greater than k, it remains so at all subsequent configurations until the termination of the *contains* operation instance by Lemma 6.1.3.

- The only step that can change $join(p)$ from a node with key less than or equal to k to a node with key larger than k, is line 3 of Algorithm 11. At this step, k is absent from the set by the list-ordering invariant.

- In the step where the $remove(k')$ operation instance changes $contains(k)$'s join to a node with a key larger than k, we can be sure that k is absent from the list, because of two invariant properties of the list implementation:

  i There is at most one unmarked node reachable from *Head* with value k.

  ii The nodes reachable from *Head* are sorted in non-decreasing order.

  There is no unmarked node with key k between $join(p)$ before and after the $remove(k')$ operation, else it would either be the new $join(p)$ or some node with key less than k would be the new $join(p)$. Therefore, by the list-ordering invariant, k is absent from the list at the moment $remove(k')$ marks $join(p)$.

The linearization strategy now becomes clear: Whenever an $remove(k')$ operation instance performs line 3 of Algorithm 13, we linearize all $contains(k)$ operation

instances, with $k \geq k'$, whose *join* that step causes to become larger than k, immediately after the linearization of the *remove*(k) operation instance – crucially, we can do this knowing only the present state of the execution.

This is a strong linearization, as once we make a linearization decision, we never revisit it – i.e., the linearization is prefix-preserving.

6.2 Linearizing *contains* Using *add*

This is not the only strong linearization we can give for this implementation. Just as we used *remove* operation instances to linearize pending *contains* operation instances, we can use *add* operation instances to do the same. Recall that the issue arose in case (6.3) where some process can add k to the list without the *contains* operation instance (which is going to return **false**) being 'aware' of it.

This suggests the following strategy: Whenever an *add*(k) operation instance performs line 6 of Algorithm 14, we linearize all pending *contains*(k) operation instances whose performing process's join key is larger than k immediately before we linearize the *add*(k) operation instance.

The validity of this linearization follows from Invariant (i) above: Since performing line 6 adds a node with value k to the list, no node in the list could have had that value immediately preceding this step. Therefore, it is valid to linearize a *contains*(k) that returns **false** at that point because it is guaranteed to return

false.

Contrast this strategy with the one above: In this case, $add(k)$ only linearizes $contains(k)$ operation instances, while in the other strategy $remove(k)$ can, in principle, linearize $contains(k')$ with $k' \geq k$. In the previous strategy, we linearized some *contains* operation instances immediately *after* a *remove*, while in this strategy, we linearize them immediately *before* an *add*.

We have shown the existence of two strong linearizations for this implementation, and we can easily see that the implementation is deterministic. By Theorem 5.2.3, there is a forward simulation from C to A, in Colvin *et al.*'s terminology.

7 Systems of ADT's

Having considered abstract data types and their implementations using automata in isolation, we now turn our attention to the combination of ADT implementations and control structures into complex systems and the issues of safety in such systems. What properties are preserved when, for instance, a simple, atomic model of an ADT is replaced in a system with a linearizable or strongly linearizable implementation? This is what we hope to address in this section.

Our reader will undoubtedly have an intuitive model of what is meant by a program – perhaps a short piece of code written in her programming language of choice will spring to mind. Once the pieces of code grow to prodigious size and complexity, she might choose to organize them into a system. We now develop definitions of 'program' and 'system' expressed in the language of the automata defined above that correspond to the intuitive notions.

We are primarily interested in the behaviour of strongly linearizable implementations and assume throughout this section that all abstract data types are

deterministic.

7.1 Sequential Programs, Concurrent Systems, and Schedulers

Definition 7.1.1. A *sequential program* for a process p using a set of abstract data types $\mathbb{D} = \{\mathcal{D}_i \mid i \in \mathcal{I}\}$ for some finite or infinite index set $\mathcal{I}$ is an automaton $Prog_p$ with the set of external actions

$$\{invoke_{\mathcal{D}_i}(op, p), respond_{\mathcal{D}_i}(rsp, p) \mid i \in \mathcal{I}, op \in \mathcal{D}_i.OPS, rsp \in \mathcal{D}_i.RSP\}$$

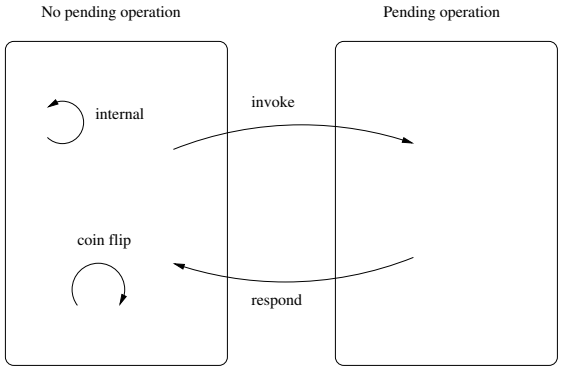
$$\bigcup$$

$$\{\text{coin flip}(p, 0), \text{coin flip}(p, 1)\}.$$

Furthermore, we require that the traces of $Prog_p$ be *well-formed*, i.e., every trace begins with a coin flip or an invocation, and every invocation, except possibly the last, is immediately followed by a corresponding response. Likewise, every response is immediately preceded by a corresponding invocation. Also, we require the automaton $Prog_p$ to make decisions based only on responses to operations it has previously invoked.

Although we study the specific case of coin flips for simplicity, the results for any probability distribution over a finite number of outcomes (e.g. die rolling) is

analogous. The state space of a sequential program $Prog_p$ can be visualized as follows:



Definition 7.1.2. A *concurrent program* $\mathcal{M}$, using a set $\mathbb{D}$ of ADT's, for a finite set of processes $\mathcal{P}$ is a set of sequential programs $\{Prog_p \mid p \in \mathcal{P}\}$.

Definition 7.1.3. A *concurrent system* (or *system*) $\mathcal{Q}$ for a finite set of processes $\mathcal{P}$ using a set of abstract data types $\mathbb{D} = \{\mathcal{D}_i \mid i \in \mathcal{I}\}$ is an automaton composed of a concurrent program $\mathcal{M}$ using $\mathbb{D}$ for $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, and automata $Objs = \{O_i \mid i \in \mathcal{I}\}$ such that each $O_i \in Objs$ is an implementation of the corresponding $\mathcal{D}_i \in \mathbb{D}$ for $\mathcal{P}$.

- $states(\mathcal{Q}) = \prod_{i=1}^{n} states(Prog_{p_i}) \times \prod_{j \in \mathcal{I}} states(O_j)$.
- $start(\mathcal{Q}) = \prod_{i=1}^{n} start(Prog_{p_i}) \times \prod_{j \in \mathcal{I}} start(O_j)$.
- $acts(\mathcal{Q}) = \bigsqcup acts(Prog_{p_i}) \cup \bigsqcup acts(O_j)$. $\bigsqcup acts(Prog_{p_i})$ and $\bigsqcup acts(O_j)$ are

disjoint unions. The set of external actions of $\mathcal{Q}$ is

$$\overline{acts}(\mathcal{Q}) = \bigsqcup_{p \in \mathcal{P}} \overline{acts}(Prog_p)$$

- We first consider steps of the form $invoke_{\mathcal{D}_i}(op, p)$, $respond_{\mathcal{D}_i}(rsp, p)$, and internal steps:

  - $(s_1, s_2, \ldots, s_n, o_1, o_2, \ldots) \xrightarrow{invoke_{\mathcal{D}_i}(op,p)} (s_1, s_2, \ldots, s'_p, \ldots, s_n, o_1, o_2, \ldots, o'_i, \ldots)$ if and only if

    $$s_p \xrightarrow{invoke_{\mathcal{D}_i}(op,p)} s'_p \in steps(Prog_p) \text{ and}$$

    $$o_i \xrightarrow{invoke(op,p)} o'_i \in steps(O_i).$$

  - $(s_1, s_2, \ldots, s_n, o_1, o_2, \ldots) \xrightarrow{respond_{\mathcal{D}_i}(rsp,p)} (s_1, s_2, \ldots, s'_p, \ldots, s_n, o_1, o_2, \ldots, o'_i, \ldots)$ if and only if

    $$s_p \xrightarrow{respond_{\mathcal{D}_i}(rsp,p)} s'_p \in steps(Prog_p) \text{ and}$$

    $$o_i \xrightarrow{respond(rsp,p)} o'_i \in steps(O_i).$$

  - $(s_1, s_2, \ldots, s_p, \ldots, s_n, o_1, o_2, \ldots) \xrightarrow{a} (s_1, s_2, \ldots, s'_p, \ldots, s_n, o_1, o_2, \ldots)$ if and only if a is an internal action or a coin flip of $Prog_p$ and $s_p \xrightarrow{a} s'_p \in steps(Prog_p)$.

  - $(s_1, s_2, \ldots, s_n, o_1, o_2, \ldots, o_i, \ldots) \xrightarrow{a} (s_1, s_2, \ldots, s_n, o_1, o_2, \ldots, o'_i, \ldots)$ if and only if a is an internal action of O_i and $o_i \xrightarrow{a} o'_i \in steps(O_i)$.

We begin our discussion of the executions produced by systems by defining *schedulers* that select continuations of the executions of systems.

Definition 7.1.4. Given a concurrent system $\mathcal{Q}$ and the set of all processes $\mathcal{P}$ in $\mathcal{Q}$, a scheduler S for $\mathcal{Q}$ is a function $S : \mathcal{E}_{\mathcal{Q}} \longrightarrow \mathcal{P} \cup \{\perp\}$, where $\mathcal{E}_{\mathcal{Q}}$ is the set of all finite executions of $\mathcal{Q}$ and $\perp$ is a special symbol unused as a process ID.

If a process p has completed all of its actions in an execution e, i.e., the automaton is in a state with no outward transitions, then $S(e') \neq p$ for all executions e' where e is a prefix of e'. If all processes have terminated in an execution e, then $S(e) = \perp$.

The function S given above has the following properties:

- $S(\varepsilon)$ is the first process that takes a step, giving a one-step execution e.
- $S(e)$ gives the next process to take a step after e, yielding the execution $e' = e \circ \mathit{action}(S(e))$ where $\mathit{action}(S(e))$ is the next action taken by process $S(e)$ in $\mathcal{Q}$ after e. Since the ADT's used by system $\mathcal{Q}$ are deterministic, *action* is uniquely determined except for coin flip outcomes.

For example, the function $S(e) = p_{|e| \mod n}$ is a round-robin scheduler. It is clear from the above definition that schedulers may arbitrarily interleave process executions, i.e., a scheduler may invoke operation *op* by process p_1, and then schedule

an operation by p_2 before op has responded to p_1. We wish to consider a restricted class of schedulers that is not permitted to do this:

Definition 7.1.5. An *atomic scheduler* S of a system $\mathcal{Q}$ is a scheduler with the restriction that if e is an execution of $\mathcal{Q}$ with an $invoke_{\mathcal{D}_i}(op, p)$ step in e without a matching response, then $S(e) = p$.

We are interested in the relationship between randomization in programs and linearizability. As such, we have previously introduced a special operation: the *coin flip*, which is local to a process, is 1 or 0 with probability p or $1-p$, respectively, for some $p \in (0, 1)$, and is independent of other coin flips. By construction of program automata, all schedulers schedule coin flips atomically, i.e., they are not permitted to interleave other operations with coin flips.

7.2 Linearizability of Systems

We have introduced a method of combining isolated implementations of (deterministic) abstract data types into systems composed of many (perhaps infinitely many) such implementations. It is necessary to ensure that the executions of the combined systems are linearizable (or strongly linearizable) if the constituent implementations are linearizable (or strongly linearizable). Fortunately, Herlihy and Wing have shown in (HW90) that linearizability is a *local* property, i.e., if the pro-

jection of the executions of the system onto an object O is linearizable for all O, then the set of executions is linearizable. Golab et al. have shown in (GHW11) that strong linearizability is also a local property.

We extend Golab et al.'s work in (GHW11) by introducing a special class of functions on executions – which we term *robust* functions – and showing that strongly linearizable implementations of programs preserve the distribution of random variables derived from these functions. We further demonstrate that strong linearizability is not a necessary condition for preserving distributions.

Definition 7.2.1. Let $\mathcal{E}$ be the set of all executions produced by concurrent systems. A function $X : \mathcal{E} \longrightarrow \mathbb{R}$ is *robust* if, for every $h, h' \in \mathcal{E}$ where h and h' have a common linearization, i.e., there is a sequential history S that is a linearization of both h and h', we have $X(h) = X(h')$.

An execution may have several valid linearizations, with some concurrently scheduled operations ordered differently in different linearizations. The properties we wish to measure should be independent of our choice of linearization, i.e., they should remain constant in all possible linearizations. This property invariance between linearizations is what we intend to capture by restricting our attention to functions that obey Definition 7.2.1. Examples of robust functions would be the number of invocations of a certain operation by a process or the return value of some operation.

In the definitions and results we will introduce shortly, we consider program-implementation-scheduler triples (abbreviated as PIST's) of the form (M, I, S), where M and I are a concurrent program and implementation, respectively, that together define a concurrent system $\mathcal{Q}$, and S is a scheduler for $\mathcal{Q}$. Such a triple completely determines a set of executions (referred to by $\mathcal{E}_{(M,I,S)}$). We consider only full executions, and not execution fragments, to be in $\mathcal{E}_{(M,I,S)}$.

We shall assume the following about all PIST's (M, I, S) we consider:

- *Finite randomness.* Every execution $h \in \mathcal{E}_{(M,I,S)}$ has only finitely many coin flip operations.

- *Completeness.* Every execution $h \in \mathcal{E}_{(M,I,S)}$ is *complete*, i.e., every operation invoked in h returns a value.

8 Probability Distribution on System Executions

We now specify the probability distribution over the executions generated by a PIST (M, I, S). Given an execution $h \in \mathcal{E}_{(M,I,S)}$, what is the probability that (M, I, S) generates the execution h?

8.1 Mappings between Coin Flips and Executions

We define a special projection Γ_{flip} from executions to sequences of coin flip steps – where $\Gamma_{\text{flip}}(h)$ denotes the subsequence of coin flip operations of h. Let

$$\mathcal{F} = \{\Gamma_{\text{flip}}(h) \mid h \in \mathcal{E}_{(M,I,S)}\}.$$

Let $\pi : \mathcal{F} \longrightarrow \{0,1\}^\star$ be the projection of sequences of $\mathcal{F}$ onto $\{0,1\}^\star$, the set of finite binary sequences; for example, $\pi((0,p_1),(1,p_7),(1,p_3),(0,p_9)) = 0110$.

Lemma 8.1.1. *Γ_{flip} is a bijection between $\mathcal{E}_{(M,I,S)}$ and $\mathcal{F}$.*

Proof. Γ_{flip} is clearly a surjection between $\mathcal{E}_{(M,I,S)}$ and $\mathcal{F}$. We now show that it is an injection.

Let $f \in \mathcal{F}$. Let $h, h' \in \Gamma_{\text{flip}}^{-1}(f)$ and suppose $h \neq h'$. Let e be the longest common prefix of h, h'. The next step of e scheduled by S is $s = \mathit{action}(S(e))$. If s is not a coin flip operation, then by Definition 7.1.4, s is the same in both h and h', contradicting the definition of e.

If s is a coin flip, then the outcome must be the same in both h and h', since $\Gamma_{\text{flip}}(h) = \Gamma_{\text{flip}}(h')$. Therefore, s is the same in both h and h', contradicting the definition of e. Since we arrive at a contradiction in both cases, our assumption that $h' \neq h$ is invalid and we conclude that $h = h'$.

This shows that Γ_{flip} is a bijection. □

Lemma 8.1.2. *There are no two elements of $\mathcal{F}$ where one is a prefix of the other.*

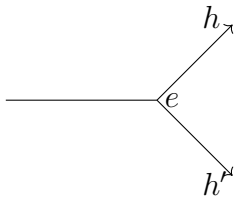
Proof. Suppose not. Let $f, f' \in \mathcal{F}$ where f is a prefix of f' and $f \neq f'$. Since Γ_{flip} is bijective by Lemma 8.1.1, $h = \Gamma_{\text{flip}}^{-1}(f) \neq \Gamma_{\text{flip}}^{-1}(f') = h'$. Since the implementations of ADT's and schedulers are deterministic, and the coin flip subsequence of h is a prefix of the coin flip sequence of h', h is a strict prefix of h'. This contradicts the assumption that $\mathcal{E}_{(M,I,S)}$ contains no execution fragments, i.e., $\mathcal{E}_{(M,I,S)}$ would not contain h. □

Lemma 8.1.3. *π is a bijection between $\mathcal{F}$ and $\pi(\mathcal{F})$*

Proof. π is clearly a surjection between $\mathcal{F}$ and $\pi(\mathcal{F})$. We now show that it is an injection.

Let $c \in \pi(\mathcal{F})$. Let $f, f' \in \pi^{-1}(c)$ and suppose $f \neq f'$. Let o be the first element of f that is different from the corresponding element o' in f' – o, o' are well-defined since, by Lemma 8.1.2, f is not a prefix of f' or vice versa – and let h, h' be $\Gamma_{\text{flip}}^{-1}(f), \Gamma_{\text{flip}}^{-1}(f')$, respectively. Since, by Lemma 8.1.1, Γ_{flip} is a bijection, $h \neq h'$.

Let e be the longest prefix common to h and h'. We claim that e is the longest prefix of h that does not contain o.



To see why, suppose e is shorter than the longest prefix g of h that does not contain o, i.e., $g = ed$ where d is non-empty, there are two cases for first step of d:

- The first step of d is not a coin flip. Since the implementations are deterministic, $action(S(e))$ is the same in both h and h', i.e., e can be extended by at least one step and remain a common prefix of both h and h'. This contradicts the definition of e.

- The first step of d is a coin flip. By construction, this coin flip is not o, i.e.,

its outcome is the same in both h and h'. Therefore, $action(S(e))$ is the same in both h and h'. Again this contradicts the definition of e.

This shows that $e = g$.

In this case, $S(e)$ would be the same in h and h', i.e., the process identity of o is the same as that of o'. Since π maps f and f' to the same sequence, the outcomes of o and o' are the same. Therefore, $o = o'$, a contradiction.

This shows that π is a bijection. □

Corollary 8.1.4. *$\pi \circ \Gamma_{flip}$ is a bijection from $\mathcal{E}_{(M,I,S)}$ to $(\pi \circ \Gamma_{flip})(\mathcal{E}_{(M,I,S)})$*

8.2 The Probability Function

We now turn our attention to defining the probability distribution itself. When we consider the space $\{0,1\}^\star$ of finite binary sequences, a probability function P and set of events $2^{\mathcal{E}_{(M,I,S)}}$ on this space immediately suggest themselves: those of a *Bernoulli process*, where for some finite sequence c of length k, $P(c) = p^l(1-p)^{k-l}$, where l is the number of 1's in c.

Since we have a bijection from $\mathcal{E}_{(M,I,S)}$ to some subset of $\{0,1\}^\star$, we define a probability distribution on $\mathcal{E}_{(M,I,S)}$ as follows:

- Probability space is $\mathcal{E}_{(M,I,S)}$.
- Set of events is $2^{\mathcal{E}_{(M,I,S)}}$.

- Probability function $\mathbb{P} = P \circ \pi \circ \Gamma_{\text{flip}}$.

To show that $\mathbb{P}$ is a probability function, we require the following lemmas:

Definition 8.2.1. A *Bernoulli tree* is a proper binary tree where, from each non-leaf node, one edge is labeled 0 and the other 1.

Lemma 8.2.2. *There is a function g from the set of PIST's to the set of Bernoulli trees $\mathcal{B}$ such that for each execution e of a PIST $\mathcal{E}$, there is a path from the root to a leaf of the Bernoulli tree $g(\mathcal{E})$ and the labels on that path form the string $\pi(\Gamma_{flip}(e))$. Furthermore, there are no other paths to a leaf in $g(\mathcal{E})$.*

Proof. Consider the complete infinite Bernoulli tree $T \in \mathcal{B}$, where every node has exactly two children, with outbound edges labeled 0 and 1. The set of finite binary sequences $\pi \circ \Gamma_{\text{flip}}(\mathcal{E}_{(M,I,S)})$ are labels of paths in T. We show that the smallest subgraph G of T that contains all of the paths in $\mathcal{E}_{(M,I,S)}$ is a Bernoulli tree.

First, G is acyclic since it is a subgraph of a tree. Furthermore, since G is a subgraph of a directed graph, G retains the directionality of the parent-child relationships inherent in T. By construction, there is a path from the root to every node in G.

To show that G is a proper binary tree, suppose not. Then there is at least one vertex $v \in V(G)$ with exactly one child. Let v be such a vertex with minimum height. By construction, there must be some path P containing v that corresponds

to an execution $e \in \mathcal{E}_{(M,I,S)}$.

Let e' be the longest execution fragment of e that does not contain the coin flip operation corresponding to v. By Definition 7.1.4, there are at least two continuations of e' in $\mathcal{E}_{(M,I,S)}$, one with the coin flip returning 1 and another with the coin flip returning 0. By definition of g, v has two children. This contradicts the definition of g. Therefore, G is a Bernoulli tree. □

Lemma 8.2.2 allows us to partition the set of PIST's into equivalence classes given by $g^{-1}(\mathcal{B})$. The following property of Bernoulli trees will allow us to prove that the function $\mathbb{P}$ is complete – i.e., the sum of probabilities of the executions of a PIST is 1.

Lemma 8.2.3. *The sum of the path label probabilities for all paths from the root to a leaf node of a finite non-empty Bernoulli tree is 1.*

Proof. The claim is true for a Bernoulli tree consisting of just one node. It is clear from Definition 8.2.1 that any binary subtree of a Bernoulli tree is a Bernoulli tree. Consider a Bernoulli tree T with more than one node. By Definition 8.2.1, T has right and left subtrees T_r and T_l, respectively.

Since T_r and T_l are both Bernoulli trees, the claim holds for them. Let $\mathcal{P}(T)$ denote the set of paths in T from the root to a leaf; for any path $p \in \mathcal{P}(T)$, $p = 0 \cdot p_l$ or $p = 1 \cdot p_r$ for paths p_l, p_r in $\mathcal{P}(T_l), \mathcal{P}(T_r)$, respectively. Therefore,

$\mathcal{P}(T) = 0 \cdot \mathcal{P}(T_l) \cup 1 \cdot \mathcal{P}(T_r)$ and

$$\begin{aligned}
\sum_{s \in \mathcal{P}(T)} P(s) &= \sum_{s \in \mathcal{P}(T_l)} P(0 \cdot s) + \sum_{s \in \mathcal{P}(T_r)} P(1 \cdot s) \\
&= \sum_{s \in \mathcal{P}(T_l)} (1-p) P(s) + \sum_{s \in \mathcal{P}(T_r)} p P(s) \\
&= (1-p) \sum_{s \in \mathcal{P}(T_l)} P(s) + p \sum_{s \in \mathcal{P}(T_r)} P(s) \\
&= 1 - p + p \\
&= 1.
\end{aligned}$$

□

Lemma 8.2.4. *$\mathbb{P}$ is a probability function on $\mathcal{E}_{(M,I,S)}$.*

Proof. Let $E \subseteq \mathcal{E}_{(M,I,S)}$.

$$\begin{aligned}
\mathbb{P}(E) &= (P \circ \pi \circ \Gamma_{\text{flip}})(E) \\
&= P(\pi(\Gamma_{\text{flip}}(E))) \\
&= P(C) && \text{for some } C \in 2^{\{0,1\}^\star} \\
&\geq 0 && \text{since } P \text{ is a probability distribution on } \{0,1\}^\star.
\end{aligned}$$

Let $E_1, E_2, \ldots, E_m \subseteq \mathcal{E}_{(M,I,S)}$ be disjoint sets. Since $\pi \circ \Gamma_{\text{flip}}$ is a bijection by

Corollary 8.1.4, $\pi \circ \Gamma_{\text{flip}}(E_1), \pi \circ \Gamma_{\text{flip}}(E_2), \ldots, \Gamma_{\text{flip}}(E_m)$ are disjoint sets.

$$\begin{aligned}
\mathbb{P}\left(\bigsqcup_{i=1}^{m} E_i\right) &= P \circ \pi \circ \Gamma_{\text{flip}}\left(\bigsqcup_{i=1}^{m} E_i\right) \\
&= P(\pi(\Gamma_{\text{flip}}\left(\bigsqcup_{i=1}^{m} E_i\right))) \\
&= P\left(\bigsqcup_{i=1}^{m} \pi(\Gamma_{\text{flip}}(E_i))\right) && \pi \circ \Gamma_{\text{flip}} \text{ is a bijection} \\
&= \sum_{i=1}^{m} P(\pi(\Gamma_{\text{flip}}(E_i))) && P \text{ is a probability distribution} \\
&= \sum_{i=1}^{m} \mathbb{P}(E_i).
\end{aligned}$$

That $\mathbb{P}(\mathcal{E}_{(M,I,S)}) = 1$ is immediate from Lemmas 8.2.2 and 8.2.3. □

We make mention here that since the σ-algebra defined on $\mathcal{E}_{(M,I,S)}$ is $2^{E_{(M,I,S)}}$, then, for any function X, $X\mid_{(M,I,S)}$ – the restriction of X to the set of executions $\mathcal{E}_{(M,I,S)}$ – is a random variable.

9 Comparing PIST's

The aim of this part of our work is to formalize the conditions necessary for preserving the properties of a program using atomic implementations of objects when those implementations are substituted for non-atomic ones. Therefore, our definitions and results are often stated using two PIST's with the same program M.

We will denote by h_c the execution of the triple (M, I, S) with coin flip vector c, i.e., restricting h_c to its subsequence of coin flips yields the coin flip vector c or, more formally, $\pi \circ \Gamma_{\text{flip}}(h_c) = c$. Hence, we denote the executions of (M, I, S) and (M, I', S') with coin flip vector c (called *corresponding executions*) by h_c and h'_c, respectively.

Definition 9.0.1. (GHW11) Two PIST's (M, I, S) and (M, I', S') are *directly linked* (called *equivalent* in (GHW11)) if, for every coin flip vector c, h_c exists if and only if h'_c exists, and h'_c and h_c have a common linearization.

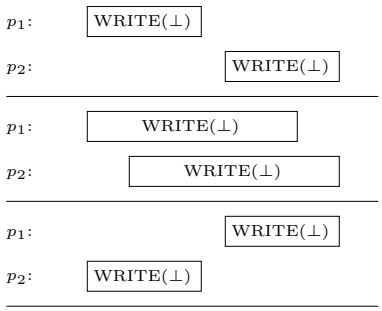
Our definition differs slightly from Golab *et al.*'s in that we do not consider two PIST's equivalent if one has an execution with a coin flip vector that is not present

in the other.

Let us consider the relation $R \subseteq \mathcal{E} \times \mathcal{E}$ – where $\mathcal{E}$ is the set of all executions – suggested by the above definition: $(h, h') \in R$ if and only if h and h' have a common linearization. By definition, R is reflexive and symmetric. However, R is not transitive in general, as the following example shows:

Consider the following set of three PIST's, where the program M has processes p_1 and p_2 perform the operation WRITE($\perp$) to the same register:

Example 9.0.2.

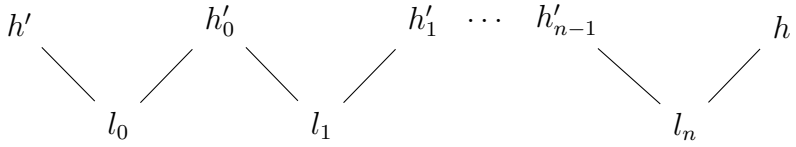


The first and second executions are directly linked, the second and third executions are directly linked, however, the first and third executions are not directly linked. It is helpful for our purposes to extend Golab *et al.*'s inaccurately named definition to a true equivalence relation by considering the transitive closure $\bar{R}$ of the relation R.

Since $\bar{R}$ is a reflexive, symmetric, and transitive relation on $\mathcal{E} \times \mathcal{E}$, it partitions $\mathcal{E}$ into equivalence classes. If we have an execution h, we term the equivalence class of $\mathcal{E}$ to which h belongs T_h.

We also have the following equivalent definition for the set T_h:

Definition 9.0.3. The *transitive closure under linearization* of an execution h' is a set of executions $T_{h'}$ such that, for all $h \in T_{h'}$, there is a collection of linearizations $l_0, l_1, \ldots, l_n$ and executions $h'_0, h'_1, \ldots, h'_{n-1}$ where h'_i has linearizations l_i and l_{i+1} for $0 \leq i \leq n-1$, l_0 is a linearization of h' and l_n is a linearization of h.



Definition 9.0.4. Two PIST's (M, I, S) and (M, I', S') are *linked* if, for every coin flip vector c, h'_c exists if and only if h_c exists and $h'_c \in T_{h_c}$.

We say that an instance of operation op in h *corresponds* to an instance of operation op' in h' if op and op' are both the k-th operation performed by process p in h and h', respectively. By $op_1 <_h op_2$, we mean that the response of operation instance op_1 precedes the invocation of operation instance op_2 in h. We omit the execution label when it is clear what execution we are referring to.

Definition 9.0.5. Let $<<_h$ be the subset of $<_h$ (the real-time ordering of execution h) where, for each pair $(op_1, op_2) \in <<_h$, op_1 and op_2 are performed by different processes. Two PIST's (M, I', S') and (M, I, S) are *cross-process consistent* if, for every coin flip vector c, $<<_{h_c} \subseteq <<_{h'_c}$ or $<<_{h'_c} \subseteq <<_{h_c}$.

In Example 9.0.2, all three PIST's are linked, since the first and last are linearizations of the second. However, the first and last executions are not cross-process

consistent because in the first execution WRITE($\perp$) by p_1 precedes WRITE($\perp$) by p_2, while in the last execution WRITE($\perp$) by p_2 precedes WRITE($\perp$) by p_1. These definitions help us capture the crucial differences between the three cases.

Definition 9.0.6. Two PIST's (M, I, S) and (M, I', S') are *distributionally equivalent* if, for every robust function $X : \mathcal{E} \longrightarrow \mathbb{R}$, $X\mid_{(M,I,S)}$ and $X\mid_{(M,I',S')}$ have the same distribution.

We remind the reader that the following two results are restricted to PIST's where all executions have a finite number of coin flips and all invoked operations eventually terminate, and we hope to extend these results by weakening these restrictions in the future. We now present the main results of this section:

Lemma 9.0.7. *For any executions h, h', if $\Gamma_{flip}(h) \neq \Gamma_{flip}(h')$, then $T_h \cap T_{h'} = \emptyset$.*

Proof. Suppose not. Since $\bar{R}$ is an equivalence relation, $T_h = T_{h'}$. Consider a chain of executions between h and h' as given by Definition 9.0.3. There are two consecutive executions g, g' in this chain with common linearization l such that $\Gamma_{\text{flip}}(g) \neq \Gamma_{\text{flip}}(g')$.

$\Gamma_{\text{flip}}(g)$ and $\Gamma_{\text{flip}}(g')$ must have the same length since l is a common linearization and coin flips are atomic, hence by Definition 4.3.2 must be linearized. Let k be the index of the first pair that is different in $\Gamma_{\text{flip}}(g)$ and $\Gamma_{\text{flip}}(g')$, and let the k-th pair of $\Gamma_{\text{flip}}(g)$ be (rsp_g, p_g) and that of $\Gamma_{\text{flip}}(g')$ be $(rsp_{g'}, p_{g'})$. It must be that $rsp_g \neq rsp_{g'}$

or $p_g \neq p_{g'}$.

Let $t = (op, rsp, p)$ be the triple in l corresponding to the k-th coin flip operation linearized in l. By Definition 4.3.2, $rsp = rsp_g = rsp_{g'}$ and $p = p_g = p_{g'}$. This is a contradiction. Hence $T_h \cap T_{h'} = \emptyset$. □

Lemma 9.0.8. *For any execution h, operations op_1, op_2, if $op_1 <_h op_2$ and the operations are done by the same process, then $op_1 <_{h'} op_2$ for every execution in $h' \in T_h$.*

Proof. Suppose not. Let h' be an execution in T_h where $op_2 \not< op_1$. Consider a chain of executions between h and h' as given by Definition 9.0.3. There are two consecutive executions g, g' in this chain with common linearization l such that $op_1 <_g op_2$ and $op_1 \not<_{g'} op_2$. Since op_1 and op_2 are performed by the same process, $op_2 <_{g'} op_1$.

We will abuse notation slightly and term the triples corresponding to op_1, op_2 in l op_1, op_2. These triples exist since both op_1 and op_2 terminate by the completeness assumption on page 87 and by Definition 4.3.2, completed operations must be linearized. By the real-time ordering condition of Definition 4.3.2, $op_1 <_l op_2$ since l is a linearization of g and $op_2 <_l op_1$ since l is a linearization of g'. This is a contradiction. □

Lemma 9.0.9. *For any execution h and operation op, if op returns rsp in h, op*

returns rsp in every execution in T_h.

Proof. Suppose not. *op* returns in every execution in T_h by the completeness assumption. Let h' be an execution in T_h where *op* does not return rsp and consider a chain of executions between h and h' as given by Definition 9.0.3. There are two consecutive executions g, g' in this chain with common linearization l such that *op* returns rsp_g in g and $rsp_{g'}$ in g' with $rsp_g \neq rsp_{g'}$.

By Definition 4.3.2, the triple (op, rsp, p) in l corresponding to the operation *op* has $rsp = rsp_g = rsp_{g'}$. This is a contradiction. □

Lemma 9.0.10. *X is a robust function if and only if, for every execution h, X is constant on T_h.*

Proof. We first prove the forward direction. Suppose not, then there is some execution h for which X is not constant on T_h. Let h' be an execution in T_h where $X(h) \neq X(h')$. Consider a chain of executions between h and h' as given by Definition 9.0.3. There are two consecutive executions g, g' in this chain with common linearization l such that $X(g) \neq X(g')$. This contradicts Definition 7.2.1 of X, since g and g' have common linearization l.

The converse follows immediately from Definition 7.2.1 and Definition 9.0.3. □

Lemma 9.0.11. *Two PIST's (M, I, S) and (M, I', S') are g-distributionally equivalent if and only if they are linked.*

Proof. Suppose that (M, I, S) and (M, I', S') are distributionally equivalent but not linked. Then, by Definition 9.0.4, there is some coin flip vector c such that $h'_c \notin T_{h_c}$ or, without loss of generality, h'_c exists but h_c does not. Consider the following function:

$$X(h) = \begin{cases} 1 & \text{for } h \in T_{h'_c} \\ \\ 0 & \text{otherwise.} \end{cases}$$

X is a robust function by Lemma 9.0.10. Let us restrict our consideration to the subdomain $E = \mathcal{E}_{(M,I,S)} \cup \mathcal{E}_{(M,I',S')}$ of X – by Lemma 9.0.7, on E, X takes the value 1 only on h'_c and 0 otherwise since h'_c is the only execution in $E \cap T_{h'_c}$ with coin flip vector c.

Since we assume that c has only finitely many coin flips, h'_c and h_c have positive probability mass. Therefore, $X\,|_{(M,I,S)}$ and $X\,|_{(M,I',S')}$ have different distributions. This contradicts our assumption – therefore, (M, I, S) and (M, I', S') are linked.

Suppose that (M, I, S) and (M, I', S') are linked and let X be a robust function. For any coin flip vector c, $X(h_c) = X(h'_c)$ by Definition 9.0.4 and Lemma 9.0.10. Therefore, since $X\,|_{(M,I,S)}$ and $X\,|_{(M,I',S')}$ have the same distribution, (M, I, S) and (M, I', S') are distributionally equivalent. □

Theorem 9.0.12. *Given two PIST's (M, I, S) and (M, I', S') with S an atomic scheduler, (M, I, S) and (M, I', S') are directly linked if and only if (M, I, S) and (M, I', S') are cross-process consistent and distributionally equivalent.*

Proof. Since S is atomic and complete by the assumptions on page 87, for every coin flip vector c, execution h_c of (M, I, S) has a unique linearization l_c with $<_{h_c}=<_{l_c}$, where $<_{l_c}$ is the ordering of operations in linearization l_c.

Suppose (M, I, S) and (M, I', S') are directly linked. Therefore, for every coin flip vector c, h_c and h'_c have a common linearization – namely, h_c's unique linearization l_c.

By the real-time ordering condition of Definition 4.3.2, $<<_{h'_c}\subseteq<<_{l_c}=<<_{h_c}$. This shows that (M, I, S) and (M, I', S') are cross-process consistent. By Lemma 9.0.11, (M, I, S) and (M, I', S') are distributionally equivalent.

Conversely, suppose that (M, I, S) and (M, I', S') are cross-process consistent and distributionally equivalent. Since S does not schedule operations concurrently, we have, from Definition 9.0.5, $<<_{h_c}\supseteq<<_{h'_c}$ for each coin flip vector c. By Lemma 9.0.11, (M, I, S) and (M, I', S') are linked.

By Lemma 9.0.8 and atomicity of S, we have that

$$<_{h_c} \setminus <<_{h_c}\supseteq<_{h'_c} \setminus <<_{h'_c} .$$

Therefore, $<_{h_c}\supseteq<_{h'_c}$. By Lemma 9.0.9, corresponding operations in h_c and h'_c have the same return values. By Definition 4.3.2, since h'_c and l_c are equivalent and $<_{h'_c}\subseteq<_{l_c}$, l_c is a linearization of h'_c. Therefore, (M, I, S) and (M, I', S') are directly linked. □

We now present a counter-example to show why the cross-process consistency condition is necessary in the above theorem.

Let us consider the following implementation of a single-reader single-writer register with READ and WRITE operations using single-reader single-writer registers with atomic **read** and **write** operations.

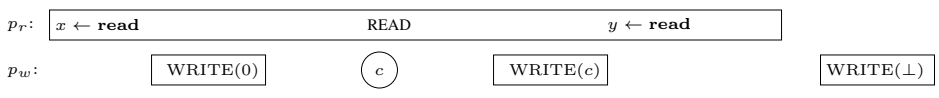
Algorithm 15: READ

```
1 x ⟵ read base register;
2 y ⟵ read base register;
3 if y = 0 then
4 |   return x
5 else
6 |   return y
7 end
```

Algorithm 16: WRITE(v)

```
1 write(v) to base register;
```

Let the base register be initialized to the value $\perp$ and suppose there are two processes p_r and p_w using the read/write register implemented using the preceding algorithms. Consider the following PIST, where c is a coin flip operation:



This pair of executions is not strongly linearizable. As the reader can verify, if

$c = 0$, p_r's READ operation returns $\perp$, while if $c = 1$, then the READ returns 1. In the former case, READ must be linearized before c, while in the latter case, READ must be linearized after c. Thus, there is no way to provide a linearization for the prefix of the execution up t (and including) the coin flip that will be prefix-preserving. This argument shows than no prefix-preserving linearization of the given schedule exists.

Consider the following PIST with an atomic scheduler, where the column labeled $c = 0$ is the schedule where the coin flip returns 0 and $c = 1$ is the schedule where the coin flip returns 1:

$c = 0$	$c = 1$
p_w: WRITE(0)	p_w: WRITE(0)
p_w: $c \leftarrow 0$	p_w: $c \leftarrow 1$
p_w: WRITE(0)	p_w: WRITE(1)
p_w: WRITE($\perp$)	p_r: READ
p_r: READ	p_w: WRITE($\perp$)

The two PIST's are not cross-process consistent, since in the case where the coin flip returns 0, p_r's READ operation completes before p_w's WRITE($\perp$) is invoked in the former PIST, while the reverse is true in the latter PIST.

Lemma 9.0.13. *The above PIST's are distributionally equivalent but not directly*

linked.

Proof. We first show that the PIST's are distributionally equivalent by showing that they are linked and using Lemma 9.0.11. We call the executions of the concurrently scheduled PIST h'_0, h'_1 and the corresponding executions of the atomic PIST h_0, h_1, respectively.

h'_1 and h_1 have a common linearization, namely:

$$(\text{WRITE}(0), \perp, p_w)(c, 1, p_w)(\text{WRITE}(1), \perp, p_w)(\text{READ}, 1, p_r)(\text{WRITE}(\perp), \perp, p_w).$$

This shows that $h'_1 \in T_{h_1}$.

Since h_0 is a complete sequential execution, by Definition 4.3.2, h_0 has only one linearization, namely:

$$(\text{WRITE}(0), \perp, p_w)(c, 0, p_w)(\text{WRITE}(0), \perp, p_w)(\text{WRITE}(\perp), \perp, p_w)(\text{READ}, \perp, p_r).$$

This is not a linearization of h'_0, since READ $<_{h'_0}$ WRITE($\perp$) while (WRITE($\perp$), $\perp, p_w$) precedes (READ, $\perp, p_r$) in the linearization, violating the real-time ordering condition of Definition 4.3.2. Therefore, h'_0 and h_0 have no common linearization and, by Definition 9.0.1, are not directly linked.

Let h''_0 an execution of the above program, where c returns 0, that is scheduled as follows:

1. p_r performs line 1 of Algorithm 15.

2. p_w performs WRITE(0), c which returns 0, WRITE(0), and WRITE($\perp$).

3. p_r performs lines 2, 3, and 6 of Algorithm 15.

h_0 and h_0'' have a common linearization

$$(\text{WRITE}(0), \perp, p_w)(c, 0, p_w)(\text{WRITE}(0), \perp, p_w)(\text{WRITE}(\perp), \perp, p_w)(\text{READ}, \perp, p_r).$$

In this case, the real-time ordering constraint is not violated since WRITE($\perp$) and READ are scheduled concurrently in h_0''. h_0' and h_0'' have a common linearization

$$(\text{READ}, \perp, p_r)(\text{WRITE}(0), \perp, p_w)(c, 0, p_w)(\text{WRITE}(0), \perp, p_w)(\text{WRITE}(\perp), \perp, p_w).$$

This shows that $h_0' \in T_{h_0}$. By Definition 9.0.4, the two PIST's are linked. □

Theorem 9.0.14. *A PIST (M, I', S') is strongly linearizable if and only if there is a PIST (M, I, S) with an atomic scheduler S where (M, I, S) is directly linked to (M, I', S').*

Proof. Suppose (M, I', S') is strongly linearizable and let g be a strong linearization function from $\mathcal{E}_{(M,I',S')}$ to $S_{\mathcal{D}}$, where $S_{\mathcal{D}}$ is the sequential specification of the set of objects $\mathcal{D}$ defined as

$$S_{\mathcal{D}} := \{S \mid S||D \in S_D \text{ for all } D \in \mathcal{D}\}$$

where $S||D$ is the projection of S onto operations invoked on object D.

Let I be the collection of objects formed by replacing all objects in I' with their atomic implementations and define a scheduler S as follows:

For every execution $h' \in \mathcal{E}_{(M,I',S')}$, construct a sequential execution h where the triples in $g(h')$ appear as atomic operations in h in the order in which they occur in $g(h')$ and no other operations appear in h. Consider

$$\mathcal{H} = \{h \mid h \text{ is constructed from some } g(h') \text{ for } h' \in \mathcal{E}_{(M,I',S')}\}.$$

By Definition 7.1.4 and Definition 4.3.3, there is some atomic scheduler S on (M, I) that generates $\mathcal{H}$ since g is prefix-preserving.

For each coin flip vector c, h_c and h'_c have a common linearization since g is a linearization function. Therefore, by Definition 9.0.1, (M, I, S) and (M, I', S') are directly linked.

Conversely, suppose that there is some PIST (M, I, S) with atomic scheduler S such that (M, I', S') and (M, I, S) are directly linked. Define a function g as follows: For each execution $E' \in \mathcal{E}_{(M,I',S')}$, let $g(E')$ be the corresponding execution in $\mathcal{E}_{(M,I,S)}$ given in the form of a sequence of triples (op, rsp, p) – this is uniquely defined since S is atomic and all executions are complete. For an execution prefix e of an execution E, let $g(e)$ be the shortest prefix of $g(E)$ that contains all completed operations in E.

We now show that g is well defined. g is well-defined on full executions by properties of PIST's. Suppose that g is not well-defined on execution prefixes. Then

there are executions $E_1', E_2' \in (M, I', S')$ and some non-empty execution prefix e of E_1', E_2' such that $g(e)$ as defined by E_1' (denoted by $g_{E_1'}(e)$) differs from $g(e)$ as defined by E_2' (denoted by $g_{E_2'}(e)$).

Suppose that e contains k coin flips. Since e is a prefix of E_1' and E_2', E_1' and E_2' are identical up to but not necessarily including the first $k+1$ coin flips, since S' is a function, i.e., deterministic. $g(E_1')$ and $g(E_2')$ (well-defined since E_1' and E_2' are full executions) have, by Lemma 9.0.7, the same first k coin flips.

$g(E_1')$ and $g(E_2')$ are identical up to but not necessarily including the first $k+1$ coin flips since S is a function, i.e., deterministic. We call this execution prefix up to but not including the $k+1$-th coin flip H.

All completed operations in e complete before the the k+1-th coin flip; otherwise e would contain more than k coin flips. Therefore,

$$\begin{aligned} g_{E_1'}(e) &= \text{ the shortest prefix of } g(E_1') \text{ that contains all completed operations in } e \\ &= \text{ the shortest prefix of } H \text{ that contains all completed operations in } e \\ &= \text{ the shortest prefix of } g(E_2') \text{ that contains all completed operations in } e \\ &= g_{E_2'}(e). \end{aligned}$$

This is a contradiction.

g is a linearization function since (M, I, S) is directly linked to (M, I', S') and S is atomic. g is prefix-preserving by its definition on execution prefixes. Therefore,

g is a strong linearization function and (M, I', S') is strongly linearizable. $\square$

10 Generalizing Robustness

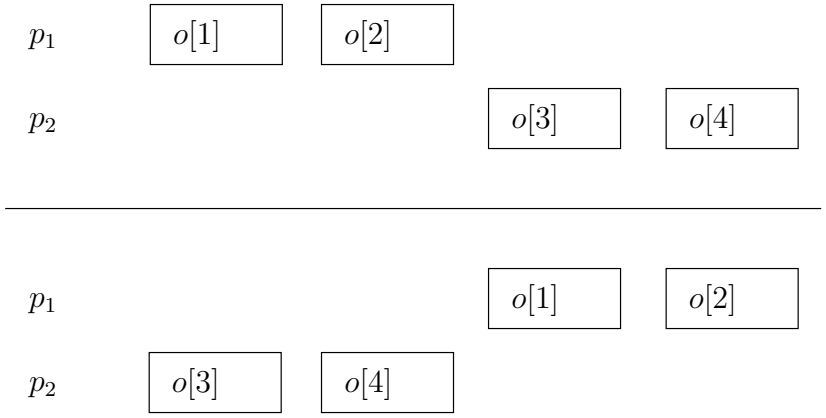
The definition of robustness we gave in Definition 7.2.1 was somewhat forced on us given the observation that the functions we are interested in should have the same value independent of our choice of linearization. Suppose that we have a sequential specification for an abstract data type $\mathcal{D}$ with a single operation o with response values in the set $\{1, 2, 3, 4, \perp\}$. Suppose that the sequential specification $S_{\mathcal{D}}$ of $\mathcal{D}$ is:

$$\{(o, 1, p_{\text{odd}}), (o, 2, p_\star), (o, 3, p_\star), (o, 4, p_\star), (o, \perp, p_\star), (o, \perp, p_\star), (o, \perp, p_\star), \ldots\}$$

$$\bigcup$$

$$\{(o, 3, p_{\text{even}}), (o, 4, p_\star), (o, 1, p_\star), (o, 2, p_\star), (o, \perp, p_\star), (o, \perp, p_\star), (o, \perp, p_\star), \ldots\}$$

by $p_{\text{odd}}, p_{\text{even}}, p_\star$ we mean processes whose identity parities are odd, even, or either, respectively. Consider the following two executions involving processes p_1 and p_2:



Example 10.0.1.

The reader can verify that the transitive closures of these executions are distinct – to see why, consider the set of executions that have a linearization in common with the first execution. It contains executions where $o[2]$ is scheduled concurrently with with $o[3]$ or $o[4]$, but no executions where $o[1]$ is scheduled concurrently with $o[4]$, since that would entail $o[2]$ being scheduled after $o[3]$ by the real-time ordering condition of Definition 4.3.2. We lack 'bridging' executions between the first and second. Therefore, a function X can take different values on each of these executions and still be robust. However, these executions are *locally indistinguishable*, meaning that the projections of the executions onto each process are identical. The only difference between these executions is the timing: in the former case, p_1's operations were completed before p_2's were invoked; *vice-versa* in the latter case.

Just as we wished the functions we considered to be invariant to linearization, we want the functions we consider 'robust' to be invariant to such global timing details. This leads us to formulate the following new definition of robustness.

Definition 10.0.2. Two executions $h, h' \in \mathcal{E}$ are said to be *locally indistinguishable* if $\Gamma(h)||p = \Gamma(h')||p$ for all $p \in \mathcal{P}$, where $\Gamma(h)||p$ is the subsequence of $\Gamma(h)$ of operations by process p. Two PIST's A and B are said to be locally indistinguishable if, for every execution in A, there is an execution in B locally indistinguishable from it and vice versa.

Definition 10.0.3. A function $X : \mathcal{E} \longrightarrow \mathbb{R}$ is *g-robust* if for every $h, h' \in \mathcal{E}$, if h, h' are locally indistinguishable, then $X(h) = X(h')$.

Just as we partitioned the set of executions $\mathcal{E}$ using the transitive closure of linearization, we can similarly partition $\mathcal{E}$ using the equivalence relation defined by local indistinguishability. In fact,

Lemma 10.0.4. *The relation R given by $(h, h') \in R$ if and only if h, h' are locally indistinguishable is an equivalence relation. Moreover, the partitioning of $\mathcal{E}$ given by transitive closure of linearization is a refinement of the partitioning given by R.*

Proof. R is clearly reflexive, symmetric, and transitive. Consider the partitioning $\mathcal{R}$ of $\mathcal{E}$ given by R. Let $h \in \mathcal{E}$ and suppose that there is no set $K \in \mathcal{R}$ such that $T_h \subseteq K$. Since the sets $\{T_h\}_{h \in \mathcal{E}}$ partition $\mathcal{E}$, there are sets $K_1, K_2 \in \mathcal{R}$ with $K_1 \neq K_2$ such that $T_h \cap K_1 \neq \emptyset$ and $T_h \cap K_2 \neq \emptyset$.

Let $h_1 \in T_h \cap K_1$ and $h_2 \in T_h \cap K_2$. By Lemmas 9.0.8 and 9.0.9, h_1 and h_2 are locally indistinguishable, they belong to the same equivalence class in $\mathcal{R}$. Therefore,

$K_1 = K_2$, a contradiction. Hence, there is some set $K \in \mathcal{R}$ such that $T_h \subseteq K$. $\square$

As we defined T_h to be the transitive closure under linearization of the execution h, let us define U_h to be the equivalence class of h under the relation of local indistinguishability. A direct consequence of Lemma 10.0.4 is that a robust function is robust.

Corollary 10.0.5. *A g-robust function X is robust*

Proof. Choose an execution h and consider an execution $h' \in T_h$. By Lemma 10.0.4, $T_h \subseteq U_h$. Therefore, $h' \in U_h$ and $X(h) = X(h')$ by Definition 10.0.3. This shows that X is constant on T_h and, by Lemma 9.0.10, X is robust. $\square$

10.1 Local Coin Flips

In the previous sections, we have used the global ordering of coin flips in an execution (e.g, Lemma 9.0.7) to prove properties of the transitive closure under linearization. However, when we consider local indistinguishability, we omit such timing information. Instead of global coin flip vectors, we use *local* coin flip vectors to make comparisons between executions.

Definition 10.1.1. A *local coin flip vector* of an execution h is an n-tuple $(c_1, c_2, \ldots, c_n)$ where, for each $1 \leq i \leq n$, c_i is the coin flip vector of $\Gamma(h)||p_i$.

The following proposition demonstrates that we do not require the global timing information in order to index executions in a PIST.

Lemma 10.1.2. *For a PIST (M, I, S), there is an injection from the coin flip vectors of $\mathcal{E}_{(M,I,S)}$ to the local coin flip vectors.*

Proof. Let $\mathcal{L}$ be the set of local coin flip vectors, and $f : \mathcal{E}_{(M,I,S)} \longrightarrow \mathcal{L}$ be defined as follows:

$$f(h) = \begin{bmatrix} \pi \circ \Gamma_{\text{flip}}(h||p_1) \\ \pi \circ \Gamma_{\text{flip}}(h||p_2) \\ \vdots \\ \pi \circ \Gamma_{\text{flip}}(h||p_n) \end{bmatrix}.$$

Suppose that f is not 1-to-1. Then there are executions $h_1, h_2 \in \mathcal{E}_{(M,I,S)}$ with $h_1 \neq h_2$, such that $f(h_1) = f(h_2)$. Let o be the first operation that is different in h_1 and h_2. o must be a coin-flip since S is deterministic; furthermore, the process performing o is the same in h_1 and h_2 for the same reason. Suppose the process performing o is p. If the coin-flip outcome is different, then $\pi \circ \Gamma_{\text{flip}}(h_1||p) \neq \pi \circ \Gamma_{\text{flip}}(h_2||p)$ – i.e., $f(h_1) \neq f(h_2)$. Therefore, o is the same in h_1 and h_2. This is a contradiction; hence, f is an injection. □

We now present the analogues to Lemma 9.0.11 and Theorem 9.0.12. Analogously to Definition 9.0.6, we say that two PIST's are *g-distributionally equivalent* if they yield the same distribution for every g-robust function.

Lemma 10.1.3. *Two PIST's (M, I, S) and (M, I', S') are g-distributionally equivalent if and only if they are locally indistinguishable.*

Proof. Suppose that (M, I, S) and (M, I', S') are g-distributionally equivalent but not locally indistinguishable. Then, by Definition 10.0.2, there is some local coin flip vector c of (M, I, S) such that h'_c does not exist or $h'_c \notin U_{h_c}$. Consider the following random variable:

$$X(h) = \begin{cases} 1 & \text{for } h \in U_{h_c} \\ 0 & \text{otherwise.} \end{cases}$$

X is a g-robust random variable by Definition 10.0.3. Let us restrict our consideration to the subdomain $E = \mathcal{E}_{(M,I,S)} \cup \mathcal{E}_{(M,I',S')}$ of X – by Lemma 10.1.2, on E, X takes the value 1 only on h_c and 0 otherwise.

Since we assume that c has only finitely many coin flips, h_c has positive probability mass. Therefore, $X \mid_{(M,I,S)}$ and $X \mid_{(M,I',S')}$ have different distributions. This contradicts our assumption – therefore, (M, I, S) and (M, I', S') are locally indistinguishable.

Suppose that (M, I, S) and (M, I', S') are locally indistinguishable and let X be a g-robust random variable. For any local coin flip vector c, $X(h_c) = X(h'_c)$ by Definition 10.0.3. Therefore, since $X \mid_{(M,I,S)}$ and $X \mid_{(M,I',S')}$ have the same distribution, (M, I, S) and (M, I', S') are distributionally equivalent. □

Theorem 10.1.4. *Given two PIST's (M, I, S) and (M, I', S') with S an atomic scheduler, (M, I, S) and (M, I', S') are directly linked if and only if (M, I, S) and (M, I', S') are cross-process consistent and g-distributionally equivalent.*

Proof. Since S is atomic and complete, for every coin flip vector c, execution h_c of (M, I, S) has a unique linearization l_c with $<_{h_c}=<_{l_c}$, where $<_{l_c}$ is the ordering of operations in linearization l_c.

Suppose (M, I, S) and (M, I', S') are directly linked. Therefore, for every coin flip vector c, h_c and h'_c have a common linearization – namely, h_c's unique linearization l_c.

By the real-time ordering condition of Definition 4.3.2, $<<_{h'_c} \subseteq <<_{l_c} = <<_{h_c}$. This shows that (M, I, S) and (M, I', S') are cross-process consistent. By Lemma 10.1.3, (M, I, S) and (M, I', S') are g-distributionally equivalent.

Conversely, suppose that (M, I, S) and (M, I', S') are cross-process consistent and g-distributionally equivalent. Since S does not schedule operations concurrently, we have, from Definition 9.0.5, $<<_{h_c} \supseteq <<_{h'_c}$ for each coin flip vector c. By Lemma 10.1.3, (M, I, S) and (M, I', S') are locally indistinguishable.

By Definition 10.0.2 and atomicity of S, we have that

$$<_{h_c} \setminus <<_{h_c} \supseteq <_{h'_c} \setminus <<_{h'_c} .$$

Therefore, $<_{h_c} \supseteq <_{h'_c}$. By Definition 10.0.2, corresponding operations in h_c and h'_c

have the same return values. By Definition 4.3.2, since h'_c and l_c are equivalent and $<_{h'_c} \subseteq <_{l_c}$, l_c is a linearization of h'_c. Therefore, (M, I, S) and (M, I', S') are directly linked. □

11 Conclusion and Future Work

In our investigation of the properties of strong linearizability, we have shown that an implementation of a deterministic abstract data type is strongly linearizable if and only if there is a forward simulation from the implementation to the ADT's abstract linearization automaton. This equivalence allows us to frame our proofs of correctness in a language much more amenable to machine-checking than conventional proofs, which would allow more computer-aided verification of complicated algorithms for correctness. Given that conventional proofs are error-prone, providing the basis for computer-aided verification could have a significant impact on how concurrent algorithms are proved correct in the future. We believe these results form a firm theoretical starting point for constructing an automated theorem prover for strong linearizability.

The second part of our investigation builds on the work of Golab *et al.* to directly show that – in a restricted setting – strong linearizability does preserve the distributions of a class of random variables – those defined by robust functions

– that encompass most, if not all, reasonable observations and measurements an observer would want to make of a distributed computational system. The obvious continuation of this work – and one we have made some progress towards – is to remove the restrictions we have placed on the computational systems we studied, namely finite randomness and completeness conditions given on page 87. We also would like to remove the condition of atomicity of coin flips.

Preliminary work has given us reason to believe that these results hold when the restrictions are removed, although more nuanced definitions of robustness and linearizability will have to be used.

Also, a deeper examination of the relationship between preserving random variable distributions and strong linearizability raises the question of the existence of a *universal* strong linearization function – a prefix-preserving linearization function defined on the set of **all** strongly linearizable executions, rather than a linearization function whose restrictions to each strongly linearizable PIST is prefix-preserving.

Bibliography

[ABD+09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[CGLM06] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 475–488, Berlin, Heidelberg, 2006. Springer-Verlag.

[CIL94] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, August 1994.

[DD15] Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19:1–19:43, September 2015.

[Doh04] Simon Doherty. Modeling and verifying non-blocking algorithms that use dynamically allocated memory. Master's thesis, Victoria University of Wellington, Wellington, New Zealand, 2004.

[DW15] Oksana Denysyuk and Philipp Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC 2015, pages 60–74, New York, NY, USA, 2015. Springer-Verlag.

[GHW11] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 373–382, New York, NY, USA, 2011. ACM.

[GR14] Rachid Guerraoui and Eric Ruppert. Linearizability is not always a safety property. In *Proceedings of the 2nd International Conference on Networked Systems*, NETYS '14, pages 57–69. Springer, 2014.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[HHL+06] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. *A Lazy Concurrent List-Based Set Algorithm*, pages 3–16. Springer-Verlag, Berlin, Heidelberg, 2006.

[HHW12] Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: Possibilities and impossibilities. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 385–394, New York, NY, USA, 2012. ACM.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[LAA87] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *In Advances in Computing Research*, 4:163–183, 1987.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations i.: Untimed systems. *Inf. Comput.*, 121(2):214–233, September 1995.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[SAGG+93] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 305–319, London, UK, UK, 1993. Springer-Verlag.

[Vid88] K. Vidyasankar. Converting Lamport's regular register to atomic register. *Inf. Process. Lett.*, 28(6):287–290, August 1988.