Memory Consistency Models for High Performance Distributed Computing

by

Victor Luchangco

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1995)

S.B., Computer Science and Engineering
Massachusetts Institute of Technology
(1995)

S.B., Mathematics
Massachusetts Institute of Technology
(1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author .....
Department of Electrical Engineering and Computer Science
September 7, 2001

Certified by .....
Nancy A. Lynch
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Memory Consistency Models for High Performance Distributed Computing

by

Victor Luchangco

Submitted to the Department of Electrical Engineering and Computer Science
on September 7, 2001, in partial fulfillment of the
requirements for the degree of
Doctor of Science in Electrical Engineering and Computer Science

Abstract

This thesis develops a mathematical framework for specifying the consistency guarantees of high performance distributed shared memory multiprocessors. This framework is based on *computations*, which specify the operations requested and constraints on how these operations may be applied; we call the framework *computation-centric*. This framework is expressive enough to specify high level synchronization mechanisms such as locks.

We use the computation-centric framework to specify and compare several memory models, to characterize programming disciplines, and to prove that weakly consistent systems provide strong consistency guarantees when certain programming disciplines are obeyed. Specifically, we define computation-centric versions of several memory models from the literature, including *sequential consistency*, *weak ordering* and *release consistency*, and we give a computation-centric characterization of *data-race-free programs*. We prove that when running data-race-free programs, weakly ordered systems appear sequentially consistent. We also define memory models that have higher level guarantees such as locks and transactions. The strongly consistent versions of these models make guarantees that are stronger than sequential consistency, and thus are easier for programmers to use. We introduce a new model called *weak sequential locking*, which has very weak guarantees, and prove that it guarantees sequential consistency and mutually exclusive locking for programs that protect memory accesses using locks. We also show that by using two-phase locking, programmers can implement serializable transactions on any memory system with weak sequential locking.

The framework is intended primarily to help programmers of such systems reason about their programs. It supports a high level of abstraction, insulating programmers from system details and enhancing the portability of their programs. The framework is also useful for implementors of such systems, in determining what guarantees their implementations provide and in assessing the advantages of providing one memory model rather than another.

Thesis Supervisor: Nancy A. Lynch
Title: Professor of Computer Science and Engineering

Contents

Acknowledgments

It's hard to believe that I'm actually done. There are so many people to thank and not nearly enough time or space to do so. But it's easy to know where to start: Nancy Lynch has been an incredible advisor, both for her example as a researcher and clear thinker and for her patience and encouragement and faith in my ability. I had never thought of going to graduate school; if not for Nancy, I doubt I ever would have. When I first joined Nancy's research group, she gave me several papers on consensus and patiently answered my questions over the course of several weeks. At the end of the summer, she had me give a talk to the group, saying "You're an expert on this topic." I was a sophomore! Since then, her support has never flagged, though I rarely deserved it. Thank you, Nancy.

I owe a large debt to my thesis committee: Arvind, Butler Lampson, Charles Leiserson and, of course, Nancy. They identified many important points that were missing from my thesis or obscured by my writing, and they cut through a lot of the fog in my thinking. Despite the warnings I had gotten about the impossibility of trying to find a three-hour time slot mutually agreeable to four faculty members, scheduling was quite easy, even though Arvind and Charles were on leave working full-time at start-ups. I appreciate their generosity with their time. Surprisingly, I had a great time at my thesis defense. Butler definitely had all the most memorable quotes.[1]

Charles Leiserson has also been an example and role model for me, especially for his gusto and clarity in teaching. It was from Charles that I first learned that writing a thesis was also hard for other people, and that courage and perseverance were as important as intelligence.

The perspective of Xiaowei Shen and Arvind as computer architects was invaluable to me in improving the accessibility of this thesis to that audience. They also helped me understand the implications of relaxing various memory consistency guarantees. Larry Rudolph and Matteo Frigo have also helped me understand real computer systems better.

The Theory of Distributed Systems group (TDS), has been my research "home" for several years, and I would like to thank the group, especially Paul Attie, Idit Keidar, Roger Khazan, Carl Livadas, Alex Shvartsman and Josh Tauber, for many stimulating discussions over the years. I'd also like to thank members who have since left, including Roberto De Prisco, Alan Fekete, Rainer Gawlick, Sergio Rajsbaum, Roberto Segala, Nir Shavit, Mark Smith, Mark Tuttle, Mandana Vaziri, George Varghese and HB Weinberg. Alan Fekete deserves special thanks for reading the early chapters of my thesis—as much as I had written—during his visit last year, and for generally being supportive in my whole endeavor.

Another person who deserves extra thanks is Joanne Talbot. She keeps everything running smoothly for us in TDS. I owe her specially for helping me schedule my thesis defense and getting my thesis to the committee members when I couldn't do it.

TDS is part of the larger Theory of Computation (TOC) group, and we have more fun with the whole TOC group. That's mostly because crazy people like Eric Lehman hold Theory Jeopardy contests and spontaneously pick snowball fights with our neighbors. And also because Be Blackburn watches over the group with a mother's eye, and throws parties

[1] "There are only two things that are real: the spec and quantum electrodynamics."

like the Corn Fest to welcome everyone in. I also had a great time with my various TOC officemates, including Adam Klivans, Stas Jarecki and Matt Levine. I'm sure Matt will get a lot more work done now that I'm not around to talk to him for hours.

Many late nights this past year, Piotr Indyk and I were the only souls alive on the third floor, in our adjacent offices. My excuse was that I was trying to finish up. I don't know what his was—maybe he hasn't adjusted to being a professor yet. But it was nice to have someone else around. And he helped me with my thesis defense one of those nights.

I'd also like to thank Greg Shomo, who keeps our computer system running smoothly, and patiently (and promptly) answers all my questions about the network.

Steve Garland helped me write my first conference paper, and I learned a lot from that process. It took me a while to get going—and I think he worried that I wasn't going to finish that paper—but we worked it out, and Steve has been very supportive of me ever since. Thanks too to John Guttag, who checks up on me every now and again. Last year, I told him I was graduating this June for sure. I'm a bit late, but at least I am done.

Marilyn Pierce, who runs the EECS grad office and has also served as my graduate counselor for the past couple of years, deserves my very hearty thanks for her patience and concern, and for making sure my registration and all my other paperwork went through even though I didn't always do the right things. I'm sure she's glad that I'm finally done. Thanks too to Monica Bell, who takes care of all the paperwork.

The International Students Office has also been very helpful, especially Maria Brennan, who has handled my case since Milena Levak retired. This office is one of the best offices in all of MIT. They were the first ones to welcome me thirteen years ago as a freshman and I only have good things to say about them.

Since I had planned to finish sooner, I've already started working at Sun Microsystems Laboratories, in the Java Technologies Group under Steve Heller. That's funny: I have never (yet) programmed in Java. The group, and the whole lab, has been wonderfully supportive, especially when my work visa finally came through the week before my thesis defense. Steve is a great manager, and I'm looking forward to learning my way around the corporate world. Mark Moir has also been very supportive; he's the one who convinced me to come to Sun in the first place.

When I interviewed at Sun, Dave Detlefs pointed me to the work on the Java memory model, which I hadn't known about before. I'd almost gotten away from thinking about memory models for programmers, which was my main interest initially, so his pointer was well-timed and very much appreciated.

My time at MIT has hardly been all work. I have been fortunate to have many friends who've made my tenure at MIT an enjoyable one. In the last couple of years, as I've been really scrambling to finish, I confess I've neglected my friends somewhat. But I am genuinely humbled by their continued goodwill and affection. When I moved house a month ago, for example, Mike Lopez, Nick and Terri Matsakis, David Stephenson and Ben Goh showed up with only a few hours' notice and helped pack and move my stuff. Danilo Almeida and Vicki Gomez put me up for two weeks while I was in-between places.

Luis Sarmenta and I were comrades-in-arms this past year, making that final push to finish our respective theses. It was good to have someone to share the load.

Mike Rowlands has borne the brunt of my unreasonableness this past year, being my roommate. Yet he has always been supportive, even staying up with me much of Thursday

night, as I made a final push to finish. He also read a draft of my thesis all the way through, so I know someone other than my committee has read my thesis.

David Robison has been my confidant for the past two-and-a-half years, through some of the hardest times in my life. I'm glad that those times are behind me now, but I remain grateful for his friendship then, and in the lighter times as well.

Mike Lopez is a bastion of steadiness and utter reliability. I know I can count on Mike when something needs to be done right, and I've relied on him often recently. Now if only we could figure out how to keep him from bursting into laughing fits during prayer . . .

I am thankful to Danilo Almeida for many things, but recently, I'm especially thankful that he keeps letting me borrow his car. I think I've used it more in the past month than he has. I could have never managed the thesis moonlighting without his help.

David Stephenson and Ben Goh have been nice to me over the last few years, often inviting me over for dinner and letting me play games at their place.

Nick and Terri Matsakis have been generous to me in many ways. Among other things, I want to thank Terri for wonderful Easter dinners, and Nick for lending me the laptop that I used to prepare and present my thesis defense.

For several years, Catholic Fellowship (CF) has been my main social outlet; many of my good friends are from that group: Danilo Almeida, Christa Beranek, Keith Dalbey, Vicki Gomez, Sunil Konath, Desmond Lim and Beata Tao, Mike Lopez, Dave Matheu, Nick and Terri Matsakis, Jake Morzinski, Luis and Michelle Sarmenta, Mike Steinberger and Erik Thoen, and also Nick Austriaco OP, Lawrence Chang, Marissa Long (now Schwartz), Robert Meagher, Robert Palazzolo and Darren Pierre OP, who went off to other places. This group of people, united in a common faith, has been a beacon of light and hope for me, and I have been privileged to be a part of CF. CF has definitely been a big part of what has made my MIT experience rich and wonderful.

I also have several other friends, from various places, that have helped make my MIT experience great: Nate Ahlgren, Chris Anderson, Alan Asbeck, Jee Bang, Anca Brad, Jesse Byler, Ien Cheng, Matt Deeds, Erik Duerr, Matthew Dyer, Jim, Tim and Daniel Derksen, Tony Falcone, Derek Fox, Joey George, Joseph Gomes, Aziz Hassanali, Jane Hsu, Brad Ito, Nicole Lazo, Tom Lee, Jeremy Lilley, Janet Lou, Matthew Mishrikey, Trisha Montalbo, Gene Osgood, Sanjay Pahuja, Adam and Colleen Powell, Doug Ricket, Todd Rider, Partha and Wendy Seshaiah, Andrew and Jenn Steele, Corissa Thompson, Jen Thurber, Rob White, Randall Whitman, Jeremy Wong and Susan Woodmansee.

My family, of course, made it possible for me to come to MIT in the first place, and they have always supported me even when they didn't agree with my choices. My sister Triccie especially helped me think through what I wanted to do after I graduated; I am grateful for her wisdom. I am particularly grateful to my parents for their unconditional, unflagging love for me.

For all the wonderful people He has placed in my life, I thank God, Who is the source of all good things and Who gives me so much more than I need. I have not deserved His grace and have constantly fallen short of His glory. I pray that these efforts of mine may be pleasing to Him still. To Him be all glory and praise.

Chapter 1

Introduction

To improve performance, many shared memory systems expose programmers to inconsistent views of the memory. To be useful, a system must guarantee some properties about the values returned by the memory; these guarantees are specified by its *memory consistency model*. Reasoning about programs running on systems with weak consistency guarantees is difficult, both because the nondeterminism and inconsistencies may lead to surprising behaviors, and because the models are often complicated and imprecise. In this thesis, we show how to model shared memory systems clearly and precisely, and how to reason about the behavior of programs running on these systems.

From bus-based symmetric multiprocessors (e.g., [76]) to distributed nonuniform memory access systems (e.g., [71]) to shared virtual memory systems implemented in software over a network of workstations (e.g., [64]), shared memory multiprocessor systems are the dominant architecture for high performance computing [72, 26]. Even more than for uniprocessors, memory latency is the critical performance factor for multiprocessor systems [72]. Unfortunately, many techniques for reducing or hiding memory latency on a uniprocessor—caching, write buffering and instruction reordering—can compromise the consistency of the memory. Many multiprocessors trade consistency for high performance. Similarly, shared memory is an attractive paradigm for multithreaded languages such as Cilk [105] and Java [49], but many compiler transformations that are transparent to sequential programs change the semantics of concurrent programs. Our framework provides a unified setting in which to specify, reason about and compare these models, and to study what guarantees are actually useful for programming concurrent systems.

Concurrent programs are difficult to write correctly [28, 70]. Techniques for structuring and reasoning about concurrent programs have been developed [28, 21, 91, 67], but they assume strong consistency guarantees [68]. The loss of these guarantees has resulted in systems with complex and subtle semantics, increasing the complexity of programming these systems, which "already stretch[es] the intellectual limits of programmers" [56, p. 33]. We show how to carry the techniques developed for strongly consistent systems over to weakly consistent systems, and to identify the properties needed for this to work.

High level multithreaded languages allow structured concurrency that cannot be expressed easily in other frameworks proposed in the literature [44, 11, 1, 38, 41, 55, 62]. These frameworks adopt a *processor-centric view* of memory, in which there is a fixed set of sequential processors issuing operations and there are no control dependencies between

operations at different processors, whereas these languages allow threads to be created and assigned to processors dynamically.[1] We believe that models should capture the programmer's intent, as expressed in the program, as closely as possible. Thus, our framework allows great flexibility in the kinds of properties it can specify.

1.1 Background

In this section, we present some of the relevant background to this thesis. Our research draws primarily from the work on modeling shared memory multiprocessors, which could fill several books. Because the way we think about concurrent systems is shaped by the pioneering work on concurrent programming, which arose in the study of operating systems in the 1960s, we begin with a discussion of that early work and the assumptions that were implicit in it.

1.1.1 Concurrent Programming from the 70s and 80s

Programmers of early concurrent systems developed structures and disciplines to organize and reason about their programs. In this subsection, we introduce some of the basic notions and the assumptions they made to reason about concurrent systems. We present the "traditional" view of concurrent systems adopted by most programmers of these systems. This view serves as the basis of much of the informal discussion of concurrent systems later in this thesis.

The basic unit of control in a concurrent system is a *thread*, which is the sequence of operations resulting from executing a section of sequential code. A thread may create new threads, which execute concurrently. It may also *synchronize* with concurrent threads. Together with the sequential order of each thread, the pattern of thread creation and synchronization specifies the *control dependencies* between operations in an execution of a program. The control dependencies define a partial order, called the *program order*, on the operations.

Threads communicate using shared data or by passing messages; shared memory systems are generally deemed easier to program [72]. Synchronization between threads is usually implemented by communication rather than using special synchronization hardware.[2] Thus, the control dependency appears as a data dependency in the program.

Concurrent programming is hard [28, 22, 70]. Unexpected behavior arises most commonly when a piece of data is accessed by two unsynchronized concurrent threads. If either thread changes the data, the result depends on the timing or scheduling of the threads, which the programmer cannot predict. This situation is called a *data race*; the threads *compete* for access to the data. Because of the unpredictability they introduce, data races are usually considered bugs. There has been a lot of research on algorithms to detect data races [31, 96, 25].

[1] Processor-centric memory models have been given for Java [82, 94], but these do not model control dependencies arising from thread creation and joins.

[2] This is *not* true, for example, for SIMD or synchronous multiprocessors, which are beyond the scope of this thesis. (Bräunl [20] gives an overview of such machines.)

Early programmers developed disciplines to avoid data races in the first place [30, 21]. The most common discipline is to designate regions of code that access shared data as *critical* [13, 103], and to use a *lock* to restrict entry into critical sections, so that at most one thread executes critical code at any time. This is still the recommended way to program concurrent systems [16, 70].

If shared data is accessed only within critical sections, then the system guarantees that the code within each critical section appears *atomic*. That is, a programmer may assume that operations within a critical section execute consecutively, without any intervening operations of other threads. This assumption makes reasoning about the program much simpler.

Lamport noted that early methods [28, 91, 67] for designing and reasoning about concurrent programs assumed that

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program [68, p. 690].

This property, called *sequential consistency*, is *not* guaranteed by "the sequentiality of each individual processor."

Without sequential consistency, dependencies due to synchronization may not be respected because threads use the memory to synchronize; synchronization relies on data dependencies rather than control dependencies, and is thus contingent on the consistency guarantees of the memory. In this case, Lamport warned,

> Protocols for synchronizing the processors must be designed at the lowest level of the machine instruction code, and verifying their correctness becomes a monumental task [68, p. 691].

A related approach to concurrent programming was developed in the area of transaction processing for large database systems. To maintain the consistency of the database even in the face of failures, researchers developed the notion of an *atomic transaction* [75]. An atomic transaction is a sequence of operations of which either all or none are effective. Partial transactions, which make the database inconsistent, are never seen. In the concurrent setting, this guarantee is extended to prevent transactions from interfering with each other; that is, transactions cannot see partial effects of other transactions. Transactions have been proposed as a basis for structuring distributed applications, which are necessarily concurrent. Gray and Reuter claim that "without transactions, distributed systems cannot be made to work for typical real-life applications." [51, p. xxiii]

1.1.2 Weak Consistency

Despite Lamport's admonition, weakly consistent memories exist because guaranteeing sequential consistency entails a significant performance cost [74, 42, 108, 63]. In this subsection, we discuss some of these systems, why they do not guarantee sequential consistency, and what they guarantee instead. This discussion touches on only a small fraction of the vast amount of work done in this area.

Almost all the work on consistency models for shared memory multiprocessors, as presented in books and survey papers [87, 2, 72, 52, 26], takes a *processor-centric* view of

a system: Each processor issues a sequence of operations, and there are no control dependencies between operations of different processors. Thus, the program order defines a sequence for each processor rather than an arbitrary partial order. This restriction is reasonable when viewed from the architectural level. Also, in the processor-centric view, memory operations are usually reads or writes to single locations.

Guaranteeing sequential consistency is expensive because it inhibits many techniques that reduce or hide memory latency, which are critical for performance [72]. These techniques can be grouped roughly into two categories: reordering instructions and maintaining inconsistent copies of memory. For example, the IBM System/370 allows a write to complete before a read issued before it, provided they access different locations [58]. On the other hand, a SPARC processor forwards the value of a buffered write to later reads [104, 107], even though this value is not visible to other processors because it has not yet been written to memory. Thus, the write buffer acts as an incoherent cache.

More recent multiprocessors, such as the PowerPC [84], the SPARC V9 [107] and the Alpha [101], allow almost all operations to be reordered. They provide *fence* instructions to inhibit reordering. Except for the problem with write buffers mentioned above, these machines maintain *cache coherence*, which means that at any time, each memory location has at most one valid value in the caches of all the processors.[3] Fences can be used to synchronize processors in a cache coherent system.

Unlike sequential consistency, the guarantees of these multiprocessors are not given as abstract consistency conditions. Rather, they are derived from informal descriptions of the operation of the hardware. Although the guarantees of each system are different, researchers identified and gave more abstract characterizations of a few general classes of systems. For example, some multiprocessors maintain the program order and require the processors to agree on the order of writes to the same location but allow them to disagree about the order of writes to different locations. Goodman called this condition *processor consistency*[4] [47] and noted that many programs execute correctly on processor consistent memory. Other systems are *weakly ordered* [33], which means they distinguish between *synchronization operations* and *data operations*. Data operations may be reordered among themselves, but not with synchronization operations, which must be *strongly ordered*.[5]

Other researchers have attempted to characterize the consistency guarantees implied cache coherence alone, as a kind of minimal consistency requirement. Goodman defined *cache consistency* as the guarantee that accesses to each location are strongly ordered [47]. Today this property is simply called *coherence* [43, 87, 8, 52, 26]. Other researchers define coherence as the property that all processors see all the writes to each location in the same order [43, 41, 52]. These definitions are equivalent if no operations are reordered [55], but not necessarily otherwise, as we shall see in Chapter 6. Unfortunately, there are other, often informal, definitions for coherence (e.g., [33]), together with many vague statements about

[3] We give a different definition of coherence below. These two notions are related but distinct. Cache coherence is a property of an implementation while memory coherence is an abstract consistency guarantee. We study coherence formally in Section 5.4.

[4] Goodman's defined processor consistency informally, and at least three different interpretations appeared [43, 12, 8]. We adopt the definition by Ahamad, et al. [8], who compare the various definitions.

[5] The original definition of strong ordering did not guarantee sequential consistency for systems with nonatomic memory access [3, 34]. Dubois and Scheurich gave new conditions in a later paper [32].

coherence. (See Section 5.4 for a few examples of such statements.)

One of the most influential relaxed models is *release consistency*, which captures the guarantees made by the Stanford DASH machine [43]. Release consistency extends weak ordering by classifying synchronization operations as *acquire*, *release*, and *nsync* operations, each with weaker restrictions than the synchronization operations of weak ordering. For a special class of programs, called *properly labeled programs*, a release consistent memory system appears sequentially consistent. That is, properly labeled programs cannot distinguish release consistency and sequential consistency.

Release consistency has been extended or modified to describe the consistency guarantees of other systems, particularly for software implementations of distributed shared memory (DSM),[6] sometimes called *shared virtual machines (SVMs)*. Examples of these include entry consistency in Midway [15], eager release consistency in Munin [23], lazy release consistency in TreadMarks [64], and scope consistency [59].

Weakly consistent systems also arise for replicated data services [17, 95, 66, 106, 83], for which the primary system criterion, availability, is incompatible with strong consistency. Our initial work on weak consistency arose from attempts to state precisely the guarantees of a lazy replication algorithm by Ladin, et al. [66], which led to our definition of *eventually-serializable data services* [36]. At the 1996 joint panel discussion of the *International Symposium on Computer Architecture* and the *Symposium on Principles of Distributed Computing* [60], we realized that the techniques we used to specify and reason about weakly consistent data services are applicable to weakly consistent shared memory multiprocessors.

Several researchers have adopted different approaches to defining memory models. The *Commit-Reconcile & Fences (CRF)* model is a "mechanism-oriented" model designed to give architects and compiler writers both control and flexibility [100]. It can be implemented efficiently and can also serve as a target for compilers of high-level languages. CRF is both upward and downward compatible with many memory models in the following sense: Programs written assuming a different memory model typically have a straightforward translation into a program for CRF. Conversely, since other memory models can often be viewed as implementations of CRF, programs written assuming CRF execute correctly on systems implementing those models [99].

The designers of the shared memory for Cilk [105] focused on guaranteeing properties that "would be sufficient for the types of problems that are naturally expressed in a multithreaded programming environment [61, p. 127]." Because Cilk is a high level language that does not have explicit processors, their shared memory model is not processor-centric.[7] Rather, it is defined on a directed acyclic graph (dag) of operations that represents the structure of an execution of a Cilk program. Instead of defining a memory model by how the system might execute, they define *dag consistency* by ruling out behaviors that produce what they deem unacceptably anomalous behavior [18]. Together with Frigo, defined a family of dag-consistent models [40]. Every member of this family is strictly weaker than coherent memory. We showed that any implementation of the strongest of these models guarantees coherence,[8] leading Frigo to conclude that coherence is the "weakest reason-

[6] Hardware implementations of DSM are often called *nonuniform memory access (NUMA)* machines [72].

[7] Our use of the term "processor-centric" originates in its use in the Cilk project [61, 18]. In Chapter 6, we give their informal term a precise technical meaning.

[8] Coherence is called *location consistency* in that paper.

able memory model" [39].

Adve and Hill redefined weak ordering as a set of formal constraints on programs, such that a system with weak ordering appears to be sequentially consistent to programs satisfying the constraints [4]. This method for defining memory models is called a *sequential consistency normal form* for the memory model [1]. Informally, the program must have "enough synchronization." For example, a program is *data-race-free-0 (DRF0)* if in every execution on a sequentially consistent system, competing nonsynchronization operations are separated by a synchronization operation. Data-race-free-0 is a sequential consistency normal form for weak ordering.[9] There are no guarantees for programs that do not satisfy the constraints. Adve and Gharachorloo call this the *programmer-centric approach* to specifying memory consistency models [2]. Using this approach, the conditions for properly labeled programs [43] define a sequential consistency normal form for release consistency.

1.1.3 Methods for Modeling Memory Consistency

Broadly speaking, there are three approaches to defining memory models in the literature, as state machines, as sets of *legal histories*, and as conditions that a program must satisfy to guarantee sequentially consistent semantics for the program. In this subsection, we discuss the merits and shortcomings of these various approaches.

The state machine approach is attractive because state machines are well understood throughout computer science and they are widely used to model computer systems. A state machine that closely corresponds to the actual system is relatively straightforward, albeit tedious, to construct. Abstract state machines can also be used to specify system requirements. Different formalisms have been employed to describe abstract state machines, including I/O automata [45, 44], specification languages [70] and term rewriting systems [100, 99].

State machine models have not been so successful at specifying memory consistency models because the state machine representation of a system does not make clear what consistency properties the system guarantees. As a result, the most common approach to specifying the memory consistency is to characterize the "legal" behaviors of a system [8, 41, 11, 54, 46]. That is, a behavior that is exhibited by the execution of a program on a correct implementation of a memory model is a *legal history* of the system. The set of legal histories defines the memory model. This approach derives from early work by Misra [86] and Herlihy and Wing [53], and there are many variations of it.

Characterizing the legal histories of a system is a static approach to specifying memory consistency in that it considers a fixed set of operations and their responses. In contrast, the state machine approach captures the dynamic interaction between a memory system and the clients accessing it. Although legal histories seem to be easier to use than state machines in reasoning about the possible executions of a system and especially in understanding the properties guaranteed by the system, they do not capture the dynamic interaction between the memory and its clients. This drawback is significant because the memory operations that are requested later may depend on the values returned for earlier

[9] A memory model may have several sequential consistency normal forms. For example, data-race-free-1 is also a sequential consistency normal form for weak ordering (and release consistency) [5].

operations. Modeling this interaction properly requires a model for the processor as well as the memory system.

The third approach, of describing the consistency model of a system by conditions that programs must satisfy to ensure sequentially consistent behavior when running on that system, is the *programmer-centric approach* discussed at the end of the previous subsection. A programmer-centric model makes no guarantees for programs that do not satisfy its conditions, but allows the programmer to reason assuming sequential consistency for programs that do satisfy the conditions. The conditions on programs specified by programmer-centric models are typically easier to understand and reason about than the weak consistency properties specified by the legal histories approach.

There are at least two weaknesses with the programmer-centric approach. First, it fixes sequential consistency as the strongest consistency guarantee for any system. Systems that guarantee only sequential consistency are difficult to program [70], and there is no consensus yet on a stronger guarantee to use as a base model. Second, a programmer-centric model does not restrict the behavior of programs that do not meet the required conditions. Although most programs on the system may satisfy the conditions, programs that do not satisfy the conditions should not be allowed to have completely arbitrary behavior.

Regardless of the approach taken to model memory consistency, almost all the memory models in the literature are processor-centric; they assume a fixed set of processors, each executing a sequential program. Other than our own work [77, 40, 36], the only work that we know of that is not processor-centric is earlier work on modeling the guarantees of the Cilk system [18, 61], and that of Gibbons and Merritt [44]. The latter work retains most of the characteristics of processor-centric models, except that the operations at each processor are not assumed to be totally ordered.

1.2 Research Goals

We want a framework for specifying the consistency guarantees of a memory system and for reasoning about the behavior of programs executed on such a system. The framework should be precise, so that we can construct rigorous proofs using the models defined, and it should be easy to use, so that these proofs are tractable. The framework should not be processor-centric; we should be able to express the constraints implied by structured multithreaded programs.

Within this framework, we want to specify memory models that have been proposed, including those of real multiprocessors. We also want to characterize precisely the programming disciplines that programmers use to get stronger guarantees from the memory, particularly the discipline of writing programs without data races to achieve sequential consistency, and of two phase locking to achieve serializable transactions. We can then prove rigorously that a program obeying a discipline does indeed get the stronger guarantees. By examining these proofs, we can identify which properties a memory system should guarantee and which are superfluous.

Specifically, the goals of this research are:

- To define an interface between programs and memory that expresses the logical structure of a program's execution on a memory system, including the operations

specified by the program and the values returned by the memory.

- To show how to specify the requirements and guarantees of memory systems with this interface precisely, along with a theory of how to reason about these systems; that is, to develop a framework based on this interface in which memory consistency models can be precisely defined and reasoned about.
- To specify, within the framework developed, several important memory models that have been proposed in the literature, including sequential consistency, weak ordering, and memories with locks.
- To investigate and clarify the various meanings of coherence, how they compare, and what they imply for consistency.
- To formally characterize data races and data-race-free programs.
- To characterize, within the framework, disciplines used to structure programs and avoid unexpected behaviors, particularly the discipline of writing programs without data races.
- To prove rigorously that by obeying the disciplines characterized, programmers of some systems may assume stronger consistency properties than the system guarantees for arbitrary programs.
- To identify the properties of the memory system that are necessary to ensure that these disciplines are valid.

The last goal is important not only for programmers, but especially for system designers, to determine what properties a memory system should guarantee.

A framework based on the structure of a program's execution rather than the program itself, or at least a model that captures the dynamic interaction between an executing program and the memory, is not completely adequate for reasoning about programs. Hence we also want to show how the framework can be used as the basis for a richer framework that does capture this dynamic interaction.

1.3 Computations

The cornerstone of our framework is a precise and unambiguous interface between the memory and the clients that access it. The clients specify a *computation*, which consists of the operations to be applied to the memory and constraints on how these operations may be applied. The memory specifies return values for the operations of a computation. Computations form the basis for specifying and reasoning about memory consistency guarantees; we call our framework *computation-centric*. In this section, we informally describe computations and the computation-centric framework, and discuss the strengths and weaknesses of our approach.

A computation is an abstract representation of the way a memory is accessed when a program is executed on a system. It is a mathematical object not tied to any particular language or system. Computations provide a general form for expressing how a memory

system is accessed by its clients. Computations can model the interface of shared memory multiprocessors, as well as high level parallel programming languages. Almost any constraint understood by a memory system can be expressed using a computation. These constraints are expressed explicitly in the computation, rather than being derived from the time that operations are requested. Thus, using computations allows us to separate the memory semantics from linguistic and scheduling issues.

We characterize a memory system by the values it may return for the operations of the computation specified by its clients. This characterization provides an unambiguous specification for a memory system, which can be used both by programmers, to reason about their programs, and by implementors, to verify the correctness of their implementations.

There are several ways in which our approach differs from most of the work in this area. First, we are completely formal, down to the specification of the data type and up to the level of proving that a system implements its specification. Second, the framework is not processor-centric. We can model processor-centric memories within the framework, but we can also express a richer set of constraints, such as those implied by high level multithreaded languages like Cilk [105] and Java [49], in which threads can be created and assigned to processors dynamically. Third, our framework is not tied to any language or architecture. Instead, it provides a client-memory interface that can be extended easily to model almost any language or architecture. Fourth, we require the precedence and other contraints to be given explicitly for each operation. In particular, these constraints are not determined by the time or order in which the operations are requested. With good language design, stating the constraints explicitly need not be too cumbersome. Fifth, the ordering that determines the effects of each operation may be completely different from the ordering for any other operation. This point is related to the previous point, in which the programmer or the memory model must explicitly specify what constraints each operation must respect. Sixth, we do not assume that sequential consistency embodies the strongest guarantees that programmers wish to have. Rather, we can model systems with stronger guarantees, such as transactional semantics, that support higher levels of abstraction.

Like the work based on legal histories, the computation-centric approach takes a static approach to modeling the memory consistency guarantees. Computation-centric models do not capture the dynamic dependence that the computations may have on the values returned by the memory. In a sense, a computation is a generalized history without the return values. This last difference is significant: Because computations do not include the return values, they model only the clients' side of the client-memory interface. Thus, computation-centric models maintain a clean split between the clients and the memory.

To bridge the gap between computations and computation-centric models on one hand and programs and shared memory systems on the other, we define a state machine model that preserves much of the flavor of the computation-centric approach but captures the dynamic interaction between clients and memory. Defining such a model is possible because of the clean split between the clients and the memory supported by computations. We show how to construct computation-centric versions of the state machine models of the memory and its clients, and we prove that any safety property of the system deduced from the computation-centric models is a property of the original state machine models.

1.4 Contributions and Thesis Organization

The contributions of this thesis can be classified into two types, the development of a theory of memory consistency and the application of that theory. In this section, we briefly discuss some of the main results.

The main theoretical contribution is, of course, the development of the *computation-centric framework* for modeling the consistency guarantees of a memory system. Innovative features of this framework include

- the clean split between the memory and its clients afforded by using computations to model the clients' requests;
- the introduction of computation transformations as a device for defining memory models;
- the definition of enclosures of computations and the use of this definition in formalizing the well-formedness conditions for memories with locks or transactions; and
- a generic "translation" between computation-centric and state machine models of a shared memory system that preserves the implementation relation.

Important applications of this theory include

- computation-centric versions of memory models in the literature, including sequential consistency, coherence, weak ordering, release consistency and processor consistency, and also of data-race-free clients;
- a restatement and rigorous proof of results in the literature, particularly that data-race-free programs are guaranteed to have sequentially consistent behavior even on systems with weak consistency guarantees;
- the modeling of high level guarantees such as locking and transactions directly in the memory model;
- a new memory model called *weak sequential locking*, which provides very weak guarantees, and a proof that programs that protect memory accesses with locks have sequentially consistent semantics when running on a memory that provides only weak sequential locking;
- a formal characterization of integrity constraints for transactional memory and new weak transactional memory models that preserve any integrity constraints preserved by serializable transactions; and
- an investigation of the notion of coherence, particularly of the relationship between two definitions of coherence in the literature.

Parts of this work has been published previously [77, 40, 78]. The original computation-centric models were developed with Matteo Frigo. The seeds of many of these ideas were developed in the Theory of Distributed Systems group, and particularly from a paper on

eventually-serializable data services with Alan Fekete, David Gupta, Nancy Lynch and Alex Shvartsman [36].

The rest of the thesis is organized into three parts. The first part, consisting of Chapters 2 to 4, develops the basic computation-centric framework. The second part, consisting of Chapters 5 to 8, builds up the repertoire of computation-centric models and techniques by modeling particular types of systems. Most of the application results are found in this part. The last part, Chapter 9, shows how to define state machine models that leverage the computation-centric framework and can capture the dynamic interaction between the memory and its clients.

Chapter 2

Serial Semantics of Memory

In this chapter, we develop the formal tools to specify the behavior of memory in the absence of concurrency. A memory is a *(data) object* whose state is accessed and modified by *operators*. An operator applied to a state of the object yields a new state and a return value. We consider only deterministic data objects; that is, objects for which the new state and the return value are uniquely determined by the old state and the operator. In the serial setting, an object is completely characterized by its current state and its *data type*, which specifies its operators and possible states. The data type also specifies an object's initial state.

The interface between the clients and the memory is defined by the operators and return values. In the serial setting, the clients specify a sequence of operators to the memory, and the memory yields return values for each operator in the sequence. Because the data type is deterministic, the return values are determined by the operator sequence specified by the clients.

Clients do not observe the state of the memory directly; they can only infer it from the values returned for the operators they invoke. If two operator sequences with the same operators yield the same return values for every operator, the sequences are equivalent from the clients' point of view. This notion of equivalence is fundamental to our study of concurrent

Our definition of data types is simple and general. The simplicity makes it easy to reason about the behavior of memory. The generality allows programmers to define arbitrary operators for the memory, which is particularly important for programmers using high level languages with powerful abstraction mechanisms. We develop the theory in this thesis almost entirely in the context of these general definitions.

At the instruction set level of multiprocessor architectures, an access to memory is usually a load from or a store to a single *location* in memory, not an arbitrary operator that may read and write several locations. This restriction greatly reduces the possible interactions between operators and allows several efficient implementations. We show how to specialize our definitions to these kinds of memories, and we identify the properties of these operators that are used to establish various consistency guarantees in the concurrent setting.

Outline: Section 2.1 defines notation and terminology for common concepts used in this thesis. Section 2.2 formally introduces the notion of a data type. Section 2.3 shows how to apply sequences of operators to data objects. Section 2.4 identifies some properties useful in characterizing operators, and Section 2.5 introduces *validity*, a property we assume for all operator sequences used in this thesis. Section 2.6 defines equivalence for operator sequences, and Section 2.7 gives several results used to prove equivalence, including the main theorem of this chapter, used throughout the thesis. Section 2.8 defines equivalence for data types. Section 2.9 discusses data types that can be partitioned into independent locations, and Section 2.10 shows how these data types can be viewed as *compositions* of simpler data types. Section 2.11 discusses alternative approaches for specifying memories.

Reading Guide: The formal definitions and proofs in this chapter serve as the foundation for the results in later chapters. The concepts introduced here, though many, are simple. A large part of this chapter consists of examples and explanation to relate these concepts to commonly understood informal notions. The main technical result in this chapter is Theorem 2.16 in Section 2.7, which, together with its corollaries, establishes sufficient conditions for the equivalence of two operator sequences. The material in Sections 2.8 and 2.10 while useful for understanding data types, is not needed for the rest of the thesis and can be skipped without loss of continuity.

2.1 Preliminary Mathematics

In this section, we define notation and terminology that is used throughout this thesis. Although the concepts are familiar, the literature contains a diversity of notation, and some concepts have a few common variations. This section is intended primarily as a reference for the particular choices made in this thesis. In a few cases, we introduce nonstandard terminology and notation for commonly used concepts. In particular, a permutation of a set is referred to as a *serialization* in this thesis.

We denote the *natural numbers* by $\mathbb{N} = \{0, 1, 2, \ldots\}$, the *integers* by $\mathbb{Z}$, and the *real numbers* by $\mathbb{R}$. The ***power set*** of a set S is the set of all subsets of S, denoted by 2^S.

We denote the *cartesian product* of two sets A and B by $A \times B$. For a family $\{S_i\}_{i\in I}$ of sets, we denote the *cartesian product* by $\prod_{i\in I} S_i$. An element of this set is called a ***tuple***, and is denoted by $(x_i)_{i\in I}$, where $x_i \in S_i$.

The ***composition*** of two functions $f\colon A \to B$ and $g\colon B \to C$ is the function $g \circ f\colon A \to C$, where $(g \circ f)(a) = g(f(a))$ for all $a \in A$. For $f\colon A \to B$, the ***range*** of f is $range(f) = \{f(a) : a \in A\}$, and the ***inverse image*** of $B' \subseteq B$ under f is $f^{-1}(B') = \{a \in A : f(a) \in B'\}$. The ***restriction*** of f to $A' \subseteq A$ is the function $f|_{A'}\colon A' \to B$ such that $f|_{A'}(a) = f(a)$ for all $a \in A'$. We say that f ***extends*** g, or is an ***extension*** of g, if g is a restriction of f. We denote that f is a *partial function* from A to B by $f\colon A \to B_\perp$, where $B_\perp = B \cup \{\perp\}$ and $f(a) = \perp$ denotes that f is not defined at $a \in A$. The element $\perp$ is special and does not belong to any basic set under discussion, so this notation is unambiguous. The ***domain*** of f is $domain(f) = \{a \in A : f(a) \neq \perp\} = f^{-1}(B)$.

For any binary relation R on a set S, the ***restriction*** of R to $S' \subseteq S$ is $R|_{S'} = R \cap (S' \times S')$. A ***partial order*** $\prec$ on a set S is any antisymmetric and transitive binary relation on S. A

partial order is ***strict*** if it is also irreflexive. A ***total order*** is a partial order $\prec$ in which, for any two distinct elements $x, y \in S$, either $x \prec y$ or $y \prec x$. Two partial orders are ***consistent*** if there is a partial order that contains both of them; that is, $\prec_1$ and $\prec_2$ are consistent if there exists a partial order $\prec$ such that $\prec_1 \cup \prec_2 \subseteq \prec$. For any relation R, we denote its *transitive closure* by $TC(R)$. Thus, $\prec_1$ and $\prec_2$ are consistent if $TC(\prec_1 \cup \prec_2)$ is a partial order.

For a sequence α, we use $|\alpha|$ to denote the length of α and $\alpha_{[i]}$ for the ith element of α, which is well-defined if $1 \leq i \leq |\alpha|$. If the sequence is nonempty, then we denote the last element by α_{last}. We denote the *empty sequence* by ϵ. We use S^* to denote the set of *finite sequences* whose elements are taken from the set S; we use S^+ to denote the set of *nonempty finite sequences*, so that $S^+ = S^* - \{\epsilon\}$. The *concatenation* of two sequences α and β is denoted by $\alpha{\cdot}\beta$. We use the same notation for adding a single element to the front or end of a sequence. The ***projection*** of a sequence α onto a set S, denoted by $\alpha|_S$, is the subsequence of α consisting of exactly those elements in S. When a sequence is given explicitly by listing its values, we sometimes delimit it using angle brackets to eliminate ambiguity (e.g., $f(\langle a_1, a_2, \ldots\rangle)$, $\langle a_1, a_2, \ldots\rangle|_S$, $\langle a_1, a_2, \ldots\rangle{\cdot}\langle b_1, b_2, \ldots\rangle$).

The set of elements in a sequence α is denoted by $elems(\alpha)$. A sequence α that contains each element of a set S exactly once is called a ***serialization*** of S. We denote the *prefix* of a serialization α ending in $x \in elems(\alpha)$ by $pref(\alpha, x)$; the prefix of α ending in x is well-defined because x appears only once in α. A serialization α of S defines a strict total order $<_\alpha$ on S; its reflexive closure is denoted $\leq_\alpha$. Also, α is ***consistent*** with a partial order $\prec$ if $\leq_\alpha$ is consistent with $\prec$. With a serialization, it is not necessary to look at long chains for cycles; a serialization is consistent with a partial order as long as the partial order does not order any two elements in the sequence differently.

Lemma 2.1 Suppose $\prec$ is a partial order on S and α is a serialization of $S' \subseteq S$. Then α is consistent with $\prec$ if and only if there do not exist $x, y \in S'$ such that $x <_\alpha y$ and $y \prec x$.

Proof: If α is consistent with $\prec$ then let $\prec' = TC(\prec \cup \leq_\alpha)$. If $x <_\alpha y$ and $y \prec x$ then $x \prec' y \prec' x$ and $x \neq y$ so $\prec'$ is not a partial order, and $\prec$ and α are not consistent.

If α is not consistent with $\prec$ then, because $\leq_\alpha$ and $\prec$ are transitive, there exists a sequence $x_1, x_2, \ldots, x_{2n}, x_{2n+1}$ such that $x_{2n+1} = x_1$ and $x_{2i-1} \prec x_{2i}$ and $x_{2i} \leq_\alpha x_{2i+1}$ for $i = 1, 2, \ldots, n$. Because $\leq_\alpha$ is a partial order, $x_{2i-1} \not\leq_\alpha x_{2i}$ for some $i = 1, 2, \ldots, n$. But $x_{2i-2} \leq_\alpha x_{2i-1}$ and $x_{2i-1} \leq_\alpha x_{2i}$, so $x_{2i-1}, x_{2i} \in S'$, which $\leq_\alpha$ totally orders. Thus $x_{2i} <_\alpha x_{2i-1}$, as required. ■

Throughout this thesis, when variables appear unbound, they are assumed to be universally quantified over an appropriate domain, which should be obvious from context.

2.2 Serial Data Types

The *data type* of an object specifies its serial semantics. It specifies the operators and the possible states of the object, and also the effect of each operator on every possible state. In particular, the data type of a memory consists of a set of states, a set of operators, a set of return values, and a function that specifies, for each state and operator, the new state and return value resulting from applying the operator to the state. The data type also specifies the initial state of the object, before any operators are applied.

Formally, a ***(serial) data type*** $\mathcal{D}$ is a quintuple $(\Sigma, \hat{\sigma}, O, R, \tau)$ consisting of:

- a set Σ of ***(object) states***,
- a distinguished ***initial state*** $\hat{\sigma} \in \Sigma$,
- a set O of ***operators***,
- a set R of ***return values***, and
- a ***transition function*** $\tau\colon \Sigma \times O \to \Sigma \times R$.

We use $.s$ and $.v$ selectors to extract the state and return value components from the result of the transition function; that is, $\tau(\sigma, o) = (\tau(\sigma, o).s,\ \tau(\sigma, o).v)$. Because we use a (total) function to specify the semantics of operators, the effect of every operator is defined for every state; that is, every operator can be applied to every state, yielding a new state and a return value. Also, the result of applying an operator to any state is unique, which constrains data objects to be deterministic.

Example 2.1 (Read/Write Register) An integer read/write register stores an integer, initially 0, and has one read operator and a $\mathsf{write}(k)$ operator for each $k \in \mathbb{Z}$. The read operator returns the stored value and leaves it unchanged. The $\mathsf{write}(k)$ operator changes the stored value to k and returns an acknowledgment, which we denote by the constant ACK. Thus, the set of possible return values is $\mathbb{Z} \cup \{\text{ACK}\}$.

Formally, the data type of the register is $\mathcal{R} = (\mathbb{Z}, 0, O, \mathbb{Z} \cup \{\text{ACK}\}, \tau)$, where $O = \{\mathsf{read}\} \cup \{\mathsf{write}(k) : k \in \mathbb{Z}\}$, and for all $n, k \in \mathbb{Z}$, $\tau(n, \mathsf{read}) = (n,\ n)$ and $\tau(n, \mathsf{write}(k)) = (k,\ \text{ACK})$. Notice that the return value of every write operator is always ACK. ■

Example 2.2 (Read/Write Memory) A read/write memory has several addresses, each of which can be read or written separately. That is, for each address a, there is an operator, $\mathsf{read}(a)$, that returns the value at that address, and one operator, $\mathsf{write}(a, k)$, for each $k \in \mathbb{Z}$, that changes the value stored at a to k and returns an acknowledgment, denoted ACK.

Formally, the data type of a read/write memory with address set A is

$$\mathcal{M} = \left(\prod_{a \in A} \mathbb{Z},\ (0)_{a \in A},\ O,\ \mathbb{Z} \cup \{\text{ACK}\},\ \tau \right),$$

where $O = \{\mathsf{read}(a) : a \in A\} \cup \{\mathsf{write}(a, k) : a \in A \text{ and } k \in \mathbb{Z}\}$
and for each $(n_a)_{a \in A} \in \prod_{a \in A} \mathbb{Z}$, $a' \in A$, and $k \in \mathbb{Z}$:

$$\tau((n_a)_{a \in A}, \mathsf{read}(a')) = ((n_a)_{a \in A},\ n_{a'})$$
$$\tau((n_a)_{a \in A}, \mathsf{write}(a', k)) = ((n'_a)_{a \in A},\ \text{ACK}), \text{ with } n'_{a'} = k \text{ and } n'_a = n_a \text{ for all } a \neq a'.$$

Each operator accesses a single address, and it is said to be *performed on* that address; that is, $\mathsf{read}(a)$ and $\mathsf{write}(a, k)$ are performed on address a. It is natural to think of each address as a read/write register, and the memory as the composition of these registers. We show how to express this intuition formally in Section 2.10. ■

Example 2.3 (Special Registers) Some registers have special read-modify-write operators, such as *test-and-set*, *fetch-and-increment*, and *compare-and-swap*, which both read and write the register. Each of these operators returns the value in the register, and then may change it to some other value. The

test-and-set operator sets it to 1; the fetch-and-increment operator increments it by 1; the compare-and-swap operator is parameterized by the new value to be stored in the register.

A register with such an operator can be defined as an extension of the standard read/write register $\mathcal{R}$. The data type of the *test-and-set register* is

$$\mathcal{TS} = (\mathbb{Z}, 0, O \cup \{\text{t\&s}\}, \mathbb{Z} \cup \{\text{ACK}\}, \tau), \text{ where } \tau(n, \text{t\&s}) = (1, n) \text{ for all } n \in \mathbb{Z}.$$

The data type of the *fetch-and-increment register* is

$$\mathcal{FI} = (\mathbb{Z}, 0, O \cup \{\text{f\&i}\}, \mathbb{Z} \cup \{\text{ACK}\}, \tau), \text{ where } \tau(n, \text{f\&i}) = (n+1, n) \text{ for all } n \in \mathbb{Z}.$$

The data type of the *compare-and-swap register* is

$$\mathcal{CS} = (\mathbb{Z}, 0, O \cup \{\text{c\&s}(k) : k \in \mathbb{Z}\}, \mathbb{Z} \cup \{\text{ACK}\}, \tau), \text{ where } \tau(n, \text{c\&s}(k)) = (k, n) \text{ for all } n, k \in \mathbb{Z}. \blacksquare$$

Even if it does not make sense to apply an operator to a particular state, the data type must still specify the result of such an application. It may return an error message and the new state may reflect that an error has occurred. In this case, the error message must be included in the set of return values, and any error states must be in the set of states. Also, every operator must have a return value, though it may be constant, as for the write operators in the examples above.

Example 2.4 (Bank Account) This example illustrates how errors can be modeled. We give the data type of a simple bank account that maintains the balance in the account, and allows a client to check the balance, and to deposit or withdraw money. A withdrawal succeeds, however, only if there is enough money in the account. Otherwise no change is made to the account, and an error message is returned.

Formally, the data type of a simple bank account is

$$\mathcal{B} = (\mathbb{Z}, 0, O, \mathbb{N} \cup \{\text{ACK}, \text{ERR}\}, \tau),$$

where $O = \{\text{balance}\} \cup \{\text{deposit}(k) : k \in \mathbb{N}\} \cup \{\text{withdraw}(k) : k \in \mathbb{N}\}$,
and for all $n, k \in \mathbb{N}$:

$$\begin{aligned} \tau(n, \text{balance}) &= (n, n) \\ \tau(n, \text{deposit}(k)) &= (n+k, \text{ACK}) \\ \tau(n, \text{withdraw}(k)) &= \begin{cases} (n-k, k) & \text{if } k \le n \\ (n, \text{ERR}) & \text{if } k > n. \end{cases} \end{aligned}$$

A deposit is simply acknowledged. A withdrawal returns either the amount withdrawn or an error message, depending whether there is enough money in the account. Because the amount withdrawn is specified by the operator, we could have modeled the return value of a successful withdrawal by a constant acknowledgment as well. The choice is merely a matter of taste. ■

No-op operator. In later chapters, it is sometimes useful to have an operator that conveys information to the memory but does not access the data; that is, the operator does not change, nor return any information about, the state of the data object. We model these cases by extending the data type with a noop operator, where $\tau(\sigma, \text{noop}) = (\sigma, \text{ACK})$ for all $\sigma \in \Sigma$.

2.3 Operator Sequences

A data object begins in its initial state and then evolves as operators are applied to it. Because the effects of the operators are deterministic, the current state of an object is determined by the sequence of operators that have been applied to it, and the value returned for each operator is determined by the state of the object at the time it is applied. In this section, we look at examples of an object evolving through a sequence of operators applied to it, and we define notation for reasoning about operator sequences.

We begin by looking at some examples of operator sequences for the data types defined in the previous section. The operators are applied in order to the state resulting from the application of the previous operator.

Example 2.5 Consider the read/write register data type $\mathcal{R}$ from Example 2.1. Suppose the following sequence of operators is applied to an object of type $\mathcal{R}$ in its initial state:

$$\mathsf{read}, \mathsf{write}(1), \mathsf{read}, \mathsf{write}(2), \mathsf{write}(3), \mathsf{read}, \mathsf{read}, \mathsf{write}(4)$$

Initially the state of the register is 0. After the first read, the state is still 0, and the value 0 is returned. After the write(1), the state is 1, and the ACK is returned. The second read leaves the state as 1 and returns 1. The write(2) and write(3) change the state to 2 and then 3, and each of them return ACK. The next two read operators leave the state at 3, and they both return 3. Finally, the write(4) changes the state to 4 and returns ACK. We denote this more compactly as follows:

$$0 \xrightarrow[0]{\mathsf{read}} 0 \xrightarrow[\text{ACK}]{\mathsf{write}(1)} 1 \xrightarrow[1]{\mathsf{read}} 1 \xrightarrow[\text{ACK}]{\mathsf{write}(2)} 2 \xrightarrow[\text{ACK}]{\mathsf{write}(3)} 3 \xrightarrow[3]{\mathsf{read}} 3 \xrightarrow[3]{\mathsf{read}} 3 \xrightarrow[\text{ACK}]{\mathsf{write}(4)} 4$$

Arrows represent the application of the operator above the arrow, to the state preceding the arrow, yielding the state following the arrow and the return value indicated below the arrow. ■

Example 2.6 An operator sequence of the fetch-and-increment register $\mathcal{FI}$ applied to its initial state:

$$0 \xrightarrow[0]{\mathsf{f\&i}} 1 \xrightarrow[1]{\mathsf{read}} 1 \xrightarrow[\text{ACK}]{\mathsf{write}(5)} 5 \xrightarrow[5]{\mathsf{read}} 5 \xrightarrow[5]{\mathsf{f\&i}} 6 \xrightarrow[\text{ACK}]{\mathsf{write}(3)} 3 \xrightarrow[3]{\mathsf{f\&i}} 4 \xrightarrow[4]{\mathsf{f\&i}} 5$$

■

Example 2.7 An operator sequence of the bank account data type $\mathcal{B}$ applied to its initial state:

$$0 \xrightarrow[\text{ACK}]{\mathsf{deposit}(15)} 15 \xrightarrow[7]{\mathsf{withdraw}(7)} 8 \xrightarrow[8]{\mathsf{balance}} 8 \xrightarrow[\text{ERR}]{\mathsf{withdraw}(10)} 8 \xrightarrow[\text{ACK}]{\mathsf{deposit}(5)} 13 \xrightarrow[13]{\mathsf{balance}} 13$$

■

We now define notation useful for reasoning about operator sequences. The set of finite operator sequences is O^*, and the set of nonempty finite sequences is $O^+ = O^* - \{\epsilon\}$. The function $\tau^+ : \Sigma \times O^+ \to \Sigma \times R$ yields the final state and the return value of the last operator resulting from applying a finite operator sequence in order. Formally, $\tau^+(\sigma, \langle o \rangle) = \tau(\sigma, o)$ and $\tau^+(\sigma, \langle o_1, o_2, \ldots \rangle) = \tau^+(\tau(\sigma, o_1).s, \langle o_2, \ldots \rangle)$. We also define functions that yield the state or the return value directly. The state function is extended to handle the empty sequence as well.

- $\mathfrak{T}^*_\Sigma : \Sigma \times O^* \to \Sigma$ such that $\mathfrak{T}^*_\Sigma(\sigma, \epsilon) = \sigma$ and $\mathfrak{T}^*_\Sigma(\sigma, \alpha) = \tau^+(\sigma, \alpha).s$ for $\alpha \in O^+$.
- $\tau^+_R : \Sigma \times O^+ \to R$ such that and $\tau^+_R(\sigma, \alpha) = \tau^+(\sigma, \alpha).v$ for $\alpha \in O^+$.

Using this notation, the current state of an object is $\tau_\Sigma^*(\hat{\sigma}, \alpha)$, where α is the operator sequence that have been applied to the object.

Example 2.8 For the read/write register $\mathcal{R}$ from Example 2.5, if

$$\alpha = \mathsf{read}, \mathsf{write}(1), \mathsf{read}, \mathsf{write}(2), \mathsf{write}(3),$$

then $\tau^+(0, \alpha) = (3, \text{ACK})$, so $\tau_\Sigma^*(0, \alpha) = 3$ and $\tau_R^+(0, \alpha) = \text{ACK}$. ■

Because an operator sequence is applied by applying each operator in the sequence consecutively, applying the concatenation of two operator sequences yields the same final result as applying the second sequence to the final state resulting from applying the first. Formally, we have the following lemma:

Lemma 2.2 For all $\sigma \in \Sigma$, $\alpha \in O^*$, and $\alpha' \in O^+$, we have $\tau^+(\sigma, \alpha{\cdot}\alpha') = \tau^+(\tau_\Sigma^*(\sigma, \alpha), \alpha')$.

Proof: This lemma follows by induction on the length of α: If $|\alpha| = 0$ then $\alpha = \epsilon$ and $\tau^+(\sigma, \epsilon{\cdot}\alpha') = \tau^+(\sigma, \alpha') = \tau^+(\tau_\Sigma^*(\sigma, \epsilon), \alpha')$. Otherwise, assume the lemma is true for sequences of length $k \in \mathbb{N}$ and that $|\alpha| = k + 1$. Let $\alpha = \alpha''{\cdot}o$. Then we have:

$$\begin{aligned}
\tau^+(\sigma, \alpha{\cdot}\alpha') &= \tau^+(\sigma, \alpha''{\cdot}o{\cdot}\alpha') && \text{by the definition of } \alpha'' \text{ and } o \\
&= \tau^+(\tau_\Sigma^*(\sigma, \alpha''), o{\cdot}\alpha') && \text{by the inductive hypothesis} \\
&= \tau^+(\tau(\tau_\Sigma^*(\sigma, \alpha''), o).s, \alpha') && \text{by the definition of } \tau^+ \\
&= \tau^+(\tau^+(\tau_\Sigma^*(\sigma, \alpha''), \langle o \rangle).s, \alpha') && \text{by the definition of } \tau^+ \\
&= \tau^+(\tau^+(\sigma, \alpha''{\cdot}o).s, \alpha') && \text{by the inductive hypothesis} \\
&= \tau^+(\tau_\Sigma^*(\sigma, \alpha''{\cdot}o), \alpha') && \text{by the definition of } \tau_\Sigma^* \\
&= \tau^+(\tau_\Sigma^*(\sigma, \alpha), \alpha') && \text{by the definition of } \alpha'' \text{ and } o
\end{aligned}$$

■

The following corollary expresses a few ways that this lemma is commonly used.

Corollary 2.3 For all $\sigma \in \Sigma$, $o \in O$, $\alpha \in O^*$, and $\alpha' \in O^+$:

- $\tau^+(\sigma, \alpha{\cdot}o) = \tau(\tau_\Sigma^*(\sigma, \alpha), o)$
- $\tau_\Sigma^*(\sigma, \alpha{\cdot}\alpha') = \tau_\Sigma^*(\tau_\Sigma^*(\sigma, \alpha), \alpha')$
- $\tau_R^+(\sigma, \alpha{\cdot}\alpha') = \tau_R^+(\tau_\Sigma^*(\sigma, \alpha), \alpha')$

2.4 Reachable States and Properties of Data Type Operators

In this section, we identify some properties of operators that are useful in reasoning about system behavior. In particular, we identify when two operations can be reordered without affecting the effect on the state or the return values. Such operations are said to be *independent*. Operations that are not independent *conflict*.

Because an object begins in the initial state and can be modified only by the operators specified by its data type, it may not be possible to achieve some states of the object. We are not interested in such states. A state of the data type is *reachable* if it can be the current state of an object, that is, if it is the state resulting from applying some operator sequence to the initial state. Formally, a state $\sigma \in \Sigma$ is ***reachable*** if $\sigma = \tau_\Sigma^*(\hat{\sigma}, \alpha)$ for some $\alpha \in O^*$.

Example 2.9 For the bank account data type $\mathcal{B}$ from Example 2.4, the reachable states are the natural numbers. The negative integers are unreachable because the state is initially 0, and the operators that reduce this value, withdraw(k) for $k \in \mathbb{N}$, only do so if it will not become negative. ■

We now identify properties that are useful in determining when operators may be reordered without affecting the return values or the final state. An operator o is ***oblivious*** to operator o' if the return value of o is unaffected by whether o' is applied before it, that is, if $\tau_R^+(\sigma, \langle o', o\rangle) = \tau_R^+(\sigma, \langle o\rangle)$ for all reachable states $\sigma \in \Sigma$. Two operators ***commute*** if the final state resulting from applying them in sequence is the same regardless of the order in which they are applied; that is, o and o' commute if $\tau_\Sigma^*(\sigma, \langle o, o'\rangle) = \tau_\Sigma^*(\sigma, \langle o', o\rangle)$ for all reachable states $\sigma \in \Sigma$. Two operators are ***independent*** if they commute and are oblivious to each other. Two operators ***conflict*** if they are not independent.

Example 2.10 For the register data type $\mathcal{R}$ from Example 2.1, every write(k) operator is oblivious to all operators and commutes with itself. The read operator commutes with all operators and is oblivious to itself. Thus, every operator conflicts with every other operator, but not with itself. ■

Example 2.11 For the read/write memory data type $\mathcal{M}$ from Example 2.2, every operator is independent of all operators performed on different addresses. In addition, every write operator is oblivious to all operators and commutes with itself, and every read operator commutes with all operators and is oblivious to itself. Thus, an operator conflicts with every operator performed on the same address except itself.

In the literature, two operations of a read/write memory are often defined to conflict if they access the same location and at least one of them writes the location [97]. This is identical to our definition, except that we do not say the operations conflict if they write the same value. ■

Example 2.12 For $\mathcal{TS}$, $\mathcal{FI}$ and $\mathcal{CS}$ from Example 2.3, every write(k) operator is oblivious to all operators and commutes with itself, and read operator commutes with all operators and is oblivious to itself.

For $\mathcal{TS}$, the t&s operator is oblivious to read and commutes with itself, read and write(1).

For $\mathcal{FI}$, the f&i operator is oblivious to read and commutes with itself and read.

For $\mathcal{CS}$, the c&s(k) operator is oblivious to read and commutes with itself, read and write(k). ■

Example 2.13 For the bank account data type $\mathcal{B}$, the balance operator commutes with all operators and is oblivious to itself. The deposit(k) operator is oblivious to all operators and commutes with balance and all deposit(k′) operators. The withdraw(k) operator commutes with balance and itself, and is only oblivious to balance. ■

Notice that an operator always commutes with itself, but it may not be oblivious to itself. This is demonstrated by the following lemma and example:

Lemma 2.4 Given a data type $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$, any operator $o \in O$ commutes with itself.

Proof: Immediate from the definition. ■

Example 2.14 The test-and-set operator t&s of $\mathcal{TS}$ from Example 2.3 is not oblivious to itself: $\tau_R^+(n, \langle t\&s, t\&s\rangle) = 1$, but $\tau_R^+(n, \langle t\&s\rangle) = n$. ■

If an operator o is oblivious to every operator in a sequence α of operators, then its return value is unaffected by whether α is applied before it. Similarly, if o commutes with every operator in α, the same final state results from applying o then α as from applying α then o. Formally, we have the following lemmas:

Lemma 2.5 If $o \in O$ is oblivious to every operator in $\alpha \in O^*$ then $\tau_R^+(\sigma, \alpha{\cdot}o) = \tau_R^+(\sigma, \langle o \rangle)$.

Proof: Straightforward by induction on the length of α. ■

Lemma 2.6 If $o \in O$ commutes with every operator in $\alpha \in O^*$ then $\tau_\Sigma^*(\sigma, \alpha{\cdot}\ o) = \tau_\Sigma^*(\sigma, o{\cdot}\alpha)$.

Proof: Straightforward by induction on the length of α. ■

There are other properties of data type operators that may be useful in reasoning about operator sequences, but which we do not define formally here. For example, a *read-only* or *transparent* operator does not change the state of the object, and the state after applying an *obliterating* operator is independent of the state before applying the operator.[1] Thus, for $\mathcal{R}$, the read operator is read-only, and every write(k) operator is obliterating. We do not discuss these or other properties further, as they are not used in this thesis.

2.5 Return Value Functions and Validity

Operators and return values define the interface between the clients and the memory. In the serial setting, we model this interface using an operator sequence, specified by the clients, and a *return value function*, which specifies the values returned by the memory for each operator in the sequence. In this section, we define return value functions and a restriction on operator sequences, called *validity*, that is needed for return value functions to be well-defined.

A ***partial return value function*** for a set X of operators is a partial function $f\colon X \to R_\perp$. If f is defined for all $x \in X$, then it is a ***(total) return value function*** for X. The clients see only the return value function specified by the memory; they do not see the order in which the operators are applied nor the state of the data object. Partial return value functions model, among other things, the view of the clients before a value has been returned for every operator, and the view of a single client, which may only see the values returned for some of the operators.

Example 2.15 For the set {deposit(5), deposit(15), withdraw(7), withdraw(10), balance} of operators of the bank account data type $\mathcal{B}$ from Example 2.4, the following function f is a return value function:

$$f(\text{deposit}(5)) = \text{ACK} \qquad f(\text{withdraw}(7)) = 7 \qquad f(\text{balance}) = 13$$
$$f(\text{deposit}(15)) = \text{ACK} \qquad f(\text{withdraw}(10)) = 10$$

■

[1] "Transparent" and "obliterating" operations are identified by Lynch, et al. [80], though their definitions differ from ours because their operations include return values.

Given an operator sequence, we can determine the return value of each operator in the sequence by applying the operators in the sequence specified. If no operator appears more than once in an operator sequence, we can use a return value function to capture these values. We say that a sequence is ***valid*** if no element appears more than once. That is, an operator sequence α is valid if it is a serialization of $\mathit{elems}(\alpha)$. Henceforth, we restrict our attention to valid operator sequences. Requiring validity does not significantly restrict the expressiveness of data types, as we discuss at the end of this section.

For any valid sequence $\alpha \in O^*$, $\sigma \in \Sigma$, and $x \in \mathit{elems}(\alpha)$, we define the ***value returned for*** x ***by*** α ***from*** σ to be $\mathit{retval}_\sigma(x, \alpha) = \tau_R^+(\sigma, \alpha')$, where $\alpha' = \mathit{pref}(\alpha, x)$, the prefix of α ending in x. That is, $\mathit{retval}_\sigma(x, \alpha)$ is the value returned for x when the operator sequence α is applied to σ. When σ is not specified, it is assumed to be the initial state $\hat{\sigma}$.

Example 2.16 As seen in Example 2.7, for the bank account data type $\mathcal{B}$, if

$$\alpha = \mathsf{deposit}(15), \mathsf{withdraw}(7), \mathsf{balance}, \mathsf{withdraw}(10), \mathsf{deposit}(5)$$

then
$$\begin{aligned}
\mathit{retval}(\mathsf{deposit}(15), \alpha) &= \text{ACK} \\
\mathit{retval}(\mathsf{withdraw}(7), \alpha) &= 7 \\
\mathit{retval}(\mathsf{balance}, \alpha) &= 8 \\
\mathit{retval}(\mathsf{withdraw}(10), \alpha) &= \text{ERR} \\
\mathit{retval}(\mathsf{deposit}(5), \alpha) &= \text{ACK}
\end{aligned}$$

If this sequence of operators is applied to a bank account with 20 dollars, we get different results; in particular, $\mathit{retval}_{20}(\mathsf{balance}, \alpha) = 28$, and $\mathit{retval}_{20}(\mathsf{withdraw}(10), \alpha) = 10$. ■

Because applying the concatenation of two operator sequences is equivalent to applying the two sequences in order, the value returned for $x \in \mathit{elems}(\beta)$ by $\alpha{\cdot}\beta$ from any state σ is the value returned for x by β from the final state resulting from applying α to σ.

Lemma 2.7 If $\alpha{\cdot}\beta$ is a valid operator sequence of $\mathcal{D}$ and $x \in \mathit{elems}(\beta)$ then for all $\sigma \in \Sigma$, we have $\mathit{retval}_\sigma(x, \alpha{\cdot}\beta) = \mathit{retval}_{\tau_\Sigma^*(\sigma,\alpha)}(x, \beta)$.

Proof: Immediate from Corollary 2.3 and the definition of *retval*. ■

Given a set of operators and a return value function for that set, we can determine whether an operator sequence that includes these operators yields the values specified by the return value function. Formally, a serialization α of a set X of operators ***explains*** a partial return value function $f\colon X \to R_\perp$ from a state $\sigma \in \Sigma$ if $f(x) = \mathit{retval}_\sigma(x, \alpha)$ for all $x \in \mathit{domain}(f)$. We sometimes simply say that α explains f if it explains f from $\hat{\sigma}$.

It is possible for several sequences to explain the same return value function. Clients, which see only the return values, cannot distinguish such sequences. This observation is the basis for allowing implementations of a memory to reorder operators. We explore this idea further in the next few sections, and throughout this thesis.

Example 2.17 Consider the return value function f from Example 2.15. The sequence

$$\mathsf{deposit}(15), \mathsf{deposit}(5), \mathsf{withdraw}(7), \mathsf{balance}, \mathsf{withdraw}(10)$$

explains f from 0, while the sequence

$$\mathsf{deposit}(15), \mathsf{deposit}(5), \mathsf{withdraw}(7), \mathsf{withdraw}(10), \mathsf{balance}$$

explains f from 10. ■

Example 2.18 For the set $X = \{\mathsf{deposit}(5), \mathsf{deposit}(15), \mathsf{withdraw}(7), \mathsf{withdraw}(10), \mathsf{balance}\}$ of operators of $\mathcal{B}$, consider the following two partial return value functions:

$$\begin{aligned} f(\mathsf{deposit}(5)) &= \text{ACK} & \qquad g(\mathsf{withdraw}(7)) &= \text{ERR} \\ f(\mathsf{deposit}(15)) &= \text{ACK} & \qquad g(\mathsf{withdraw}(10)) &= 10 \\ f(\mathsf{balance}) &= 8 \end{aligned}$$

The sequence

$$\mathsf{deposit}(15), \mathsf{withdraw}(7), \mathsf{balance}, \mathsf{withdraw}(10), \mathsf{deposit}(5)$$

explains f (from 0), while the sequence

$$\mathsf{withdraw}(10), \mathsf{deposit}(15), \mathsf{withdraw}(7), \mathsf{balance}, \mathsf{deposit}(5)$$

explains g. Note that no sequence explains both f and g (from 0). ■

Although validity may seem to be a strong requirement on operator sequences, it does not significantly restrict the expressiveness of data types. Of course, for some data types, we want to be able to invoke an operator several times. For example, in the read/write register $\mathcal{R}$ of Example 2.1, the only way to access the state of the register is by the read operator. A data object in which this operator could only be invoked once would not be very useful. However, we can extend any such data type to allow multiple invocations of each operator by adding unique identifiers to the operators. Formally, given a data type $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$ and a set I of ***operation identifiers***, we define a new data type $\mathcal{D} = (\Sigma, \hat{\sigma}, I \times O, R, \tau')$, where $\tau'(\sigma, (id, o)) = \tau(\sigma, o)$ for all $\sigma \in \Sigma$, $id \in I$ and $o \in O$. To keep the data type descriptions in our examples simple, we will continue to define them without the identifiers. When we give example operator sequences of these data types, we distinguish multiple invocations of an operator by indexing them.

Example 2.19 Indexing the operators in the operator sequence from Example 2.5 yields

$$\alpha = \mathsf{read}_1, \mathsf{write}(1)_1, \mathsf{read}_2, \mathsf{write}(2)_1, \mathsf{write}(3)_1, \mathsf{read}_3, \mathsf{read}_4, \mathsf{write}(4)_1.$$

Thus, we can distinguish the various invocations of read in α, so $\mathit{retval}(\mathsf{read}_1, \alpha) = 0$, while $\mathit{retval}(\mathsf{read}_4, \alpha) = 3$.

From now on, we assume that all data types defined are extended with identifiers as described above. In the examples, however, we show the indices only for operators that are invoked multiple times. Thus, we write the sequence above as

$$\alpha = \mathsf{read}_1, \mathsf{write}(1), \mathsf{read}_2, \mathsf{write}(2), \mathsf{write}(3), \mathsf{read}_3, \mathsf{read}_4, \mathsf{write}(4).$$

The indices need not appear in order as long as they are unique. Thus,

$$\mathsf{read}_7, \mathsf{write}(1), \mathsf{read}_2, \mathsf{write}(2), \mathsf{write}(3), \mathsf{read}_1, \mathsf{read}_4, \mathsf{write}(4)$$

is a valid operator sequence, but distinct from α. ■

2.6 Equivalences for Operator Sequences

Often operators can be applied in different orders and still yield the same results. Relating operator sequences that yield the same results defines a kind of equivalence between different serializations of a set of operators. If clients cannot distinguish equivalent serializations, then an implementation can reorder operators as long as it preserves equivalence. In this section, we formally define equivalence for operator sequences.

Informally, two sequences are *equivalent* if the clients cannot distinguish them. Equivalent sequences must have the same operators and they must yield the same return value for each operator. More precisely, this property is called *observational equivalence.*

However, if sequences are only observationally equivalent, the final state of the object after applying these sequences may not be the same. In this case, the return value for a later operator may reveal which of the two sequences had been applied. Operator sequences which result in the same final state are *internally equivalent*. Sequences that are both observationally and internally equivalent are *strongly equivalent*.

Formally, two serializations α and β of a set X of operators are ***observationally equivalent*** if $\mathit{retval}_\sigma(x, \alpha) = \mathit{retval}_\sigma(x, \beta)$ for all reachable states $\sigma \in \Sigma$ and $x \in X$, and they are ***internally equivalent*** if $\tau^*_\Sigma(\sigma, \alpha) = \tau^*_\Sigma(\sigma, \beta)$ for all reachable states $\sigma \in \Sigma$. Two serializations of X are ***strongly equivalent*** if they are both observationally and internally equivalent. We write $\alpha \equiv_{\text{obs}} \beta$ if α and β are observationally equivalent, $\alpha \equiv_{\text{int}} \beta$ if they are internally equivalent, and $\alpha \equiv_{\text{str}} \beta$ if they are strongly equivalent.

Example 2.20 For $\mathcal{M}$, let
$$\alpha = \mathsf{read}(x), \mathsf{write}(x,1), \mathsf{write}(x,2), \mathsf{read}(y)$$
$$\beta = \mathsf{write}(x,1), \mathsf{read}(x), \mathsf{write}(x,2), \mathsf{read}(y)$$
$$\gamma = \mathsf{write}(x,2), \mathsf{write}(x,1), \mathsf{read}(x), \mathsf{read}(y)$$
$$\gamma' = \mathsf{write}(x,2), \mathsf{write}(x,1), \mathsf{read}(y), \mathsf{read}(x)$$
Then $\alpha \equiv_{\text{int}} \beta$, $\beta \equiv_{\text{obs}} \gamma$, $\beta \equiv_{\text{obs}} \gamma'$ and $\gamma \equiv_{\text{str}} \gamma'$. ■

Example 2.21 For $\mathcal{B}$, let
$$\alpha = \mathsf{deposit}(10), \mathsf{withdraw}(7), \mathsf{withdraw}(3), \mathsf{balance}$$
$$\beta = \mathsf{balance}, \mathsf{deposit}(10), \mathsf{withdraw}(7), \mathsf{withdraw}(3)$$
$$\gamma = \mathsf{deposit}(10), \mathsf{balance}, \mathsf{withdraw}(7), \mathsf{withdraw}(3)$$
Then $\alpha \equiv_{\text{str}} \beta$, $\alpha \equiv_{\text{int}} \gamma$, and $\beta \equiv_{\text{int}} \gamma$.

Note that the sequence

$$\mathsf{withdraw}(7), \mathsf{deposit}(10), \mathsf{balance}, \mathsf{withdraw}(3)$$

is neither internally nor observationally equivalent with any of α, β or γ; if the state is originally 6 or less, then ERR will be returned for $\mathsf{withdraw}(7)$, and the state will remain unchanged. ■

As their names suggest, all these notions of equivalence define equivalence relations on the set of serializations of a set of operators.

Lemma 2.8 The relations $\equiv_{\text{int}}$, $\equiv_{\text{obs}}$ and $\equiv_{\text{str}}$ are equivalence relations.

Proof: It is straightforward to check the reflexivity, symmetry and transitivity of $\equiv_{\text{int}}$ and $\equiv_{\text{obs}}$ from the definitions. Since $\equiv_{\text{str}}$ is $\equiv_{\text{int}} \cap \equiv_{\text{obs}}$, it too is an equivalence relation. ■

Observational equivalence guarantees that the clients cannot distinguish two operator sequences based on the values returned for the operators in that sequence. Internal equivalence is more of a guarantee for the future: Because the final state of an object is the same after applying either of two internally equivalent operator sequences, a client cannot determine from the return values for subsequent operators which sequence was applied. That is, if two observationally equivalent sequences are each preceded by internally equivalent sequences, the values returned for the operators by each of the observationally equivalent sequences are the same.

Lemma 2.9 If $\alpha{\cdot}\beta$ is a valid operator sequence, $\alpha \equiv_{\text{int}} \alpha'$ and $\beta \equiv_{\text{obs}} \beta'$ then for any reachable state $\sigma \in \Sigma$ and $x \in \mathit{elems}(\beta)$, we have $\mathit{retval}_\sigma(x, \alpha{\cdot}\beta) = \mathit{retval}_\sigma(x, \alpha'{\cdot}\beta')$.

Proof: We have

$$\begin{aligned}
\mathit{retval}_\sigma(x, \alpha{\cdot}\beta) &= \mathit{retval}_{\tau^*_\Sigma(\sigma,\alpha)}(x, \beta) && \text{by Lemma 2.7}\\
&= \mathit{retval}_{\tau^*_\Sigma(\sigma,\alpha)}(x, \beta') && \text{since } \beta \equiv_{\text{obs}} \beta'\\
&= \mathit{retval}_{\tau^*_\Sigma(\sigma,\alpha')}(x, \beta') && \text{since } \alpha \equiv_{\text{int}} \alpha'\\
&= \mathit{retval}_\sigma(x, \alpha'{\cdot}\beta') && \text{by Lemma 2.7.}
\end{aligned}$$

■

It follows immediately that two strongly equivalent operator sequences followed by two observationally equivalent sequences are observationally equivalent.

Corollary 2.10 If $\alpha{\cdot}\beta$ is a valid operator sequence, $\alpha \equiv_{\text{str}} \alpha'$ and $\beta \equiv_{\text{obs}} \beta'$ then $\alpha{\cdot}\beta \equiv_{\text{obs}} \alpha'{\cdot}\beta'$.

Proof: For $x \in \mathit{elems}(\beta)$, we have $\mathit{retval}_\sigma(x, \alpha{\cdot}\beta) = \mathit{retval}_\sigma(x, \alpha'{\cdot}\beta')$ by the previous lemma. For $x \in \mathit{elems}(\alpha)$ and any reachable state $\sigma \in \Sigma$, we have $\mathit{retval}_\sigma(x, \alpha{\cdot}\beta) = \mathit{retval}_\sigma(x, \alpha) = \mathit{retval}_\sigma(x, \alpha') = \mathit{retval}_\sigma(x, \alpha'{\cdot}\beta')$ because $\alpha \equiv_{\text{obs}} \alpha'$. ■

Internal equivalence also has the nice property that it is preserved by concatenation.

Lemma 2.11 If $\alpha{\cdot}\beta$ is a valid operator sequence, $\alpha \equiv_{\text{int}} \alpha'$ and $\beta \equiv_{\text{int}} \beta'$ then $\alpha{\cdot}\beta \equiv_{\text{int}} \alpha'{\cdot}\beta'$.

Proof: For any reachable state $\sigma \in \Sigma$, we have

$$\begin{aligned}
\tau^*_\Sigma(\sigma, \alpha{\cdot}\beta) &= \tau^*_\Sigma(\tau^*_\Sigma(\sigma, \alpha), \beta) && \text{by Corollary 2.3}\\
&= \tau^*_\Sigma(\tau^*_\Sigma(\sigma, \alpha'), \beta) && \text{since } \alpha \equiv_{\text{int}} \alpha'\\
&= \tau^*_\Sigma(\tau^*_\Sigma(\sigma, \alpha'), \beta') && \text{since } \beta \equiv_{\text{int}} \beta' \text{ and } \tau^*_\Sigma(\sigma, \alpha') \text{ is reachable}\\
&= \tau^*_\Sigma(\sigma, \alpha'{\cdot}\beta') && \text{by Corollary 2.3.}
\end{aligned}$$

■

From the two previous results, we immediately get that equivalence is preserved by concatenation. This implies that one equivalent sequence can be substituted for another without changing the values returned to the clients. These results are captured in the following two corollaries:

Corollary 2.12 If $\alpha{\cdot}\beta$ is a valid operator sequence, $\alpha \equiv_{\text{str}} \alpha'$ and $\beta \equiv_{\text{str}} \beta'$, then $\alpha{\cdot}\beta \equiv_{\text{str}} \alpha'{\cdot}\beta'$.

Corollary 2.13 If $\gamma{\cdot}\alpha{\cdot}\gamma'$ is a valid operator sequence and $\alpha \equiv_{\text{str}} \beta$, then $\gamma{\cdot}\alpha{\cdot}\gamma' \equiv_{\text{str}} \gamma{\cdot}\beta{\cdot}\gamma'$.

2.7 Proving Operator Sequences Equivalent

This section includes the main results of this chapter, which state conditions under which operators can be reordered while preserving equivalence. Informally, the idea is that if two operators are independent, they can be reordered without changing the effects of the operator sequences. Thus, two operator sequences are equivalent as long as they order conflicting operators in the same way.

We first prove that internal equivalence is preserved if only commutative operators are reordered.

Lemma 2.14 Suppose α and β are serializations of a set X of operators such that for all $x, y \in X$, if $x <_\alpha y$ and $y <_\beta x$ then x and y commute. Then we have $\alpha \equiv_{\text{int}} \beta$.

Proof: By induction on the size of X (also the lengths of α and β): The base case, when $X = \emptyset$, is trivial because $\alpha = \beta = \epsilon$.

For the inductive step, let $x = \alpha_{\text{last}}$, and let $\alpha = \alpha' \cdot x$ and $\beta = \gamma \cdot x \cdot \gamma'$. By the inductive hypothesis, $\alpha' \equiv_{\text{int}} \gamma \cdot \gamma'$, since α' and $\gamma \cdot \gamma'$ are just α and β without x. For $y \in \mathit{elems}(\gamma')$, we have $y <_\alpha x$ and $x <_\beta y$, so x and y commute. Thus, by Lemma 2.6, we have $\gamma' \cdot x \equiv_{\text{int}} x \cdot \gamma'$. and by Lemma 2.11, we have $\alpha = \alpha' \cdot x \equiv_{\text{int}} \gamma \cdot \gamma' \cdot x \equiv_{\text{int}} \gamma \cdot x \cdot \gamma' = \beta$. ■

To prove the corresponding result when only independent operators are reordered, we need the following lemma, which says that moving an operator before a sequence of independent operators preserves equivalence.

Lemma 2.15 If $\alpha \cdot x$ is a valid operator sequence and x is independent of every operation in α then $x \cdot \alpha \equiv_{\text{str}} \alpha \cdot x$.

Proof: By induction on the length of α: The base case, when $\alpha = \epsilon$, is trivial. Otherwise, let $\alpha = \alpha' \cdot y$. By the inductive hypothesis, $\alpha' \cdot x \equiv_{\text{str}} x \cdot \alpha'$. Since x and y are independent, $\langle x, y \rangle \equiv_{\text{str}} \langle y, x \rangle$. Thus, by Corollary 2.12, $\alpha \cdot x = \alpha' \cdot \langle y, x \rangle \equiv_{\text{str}} \alpha' \cdot \langle x, y \rangle \equiv_{\text{str}} x \cdot \alpha' \cdot y = x \cdot \alpha$. ■

We now show that equivalence is preserved if the order between conflicting operators is preserved; that is, if only independent operators are reordered. This key theorem is used to prove the main results about computations and memory models in the rest of this thesis.

Theorem 2.16 Suppose that α and β are serializations of a set X of operators such that for all $x, y \in X$, if $x <_\alpha y$ and $y <_\beta x$ then x and y are independent. Then we have $\alpha \equiv_{\text{str}} \beta$.

Proof: By induction on the size of X (also the lengths of α and β): This proof follows exactly the proof of Lemma 2.14. The base case, when $X = \emptyset$, is trivially true.

For the inductive step, let $x = \alpha_{\text{last}}$, and let $\alpha = \alpha' \cdot x$ and $\beta = \gamma \cdot x \cdot \gamma'$. For $y \in \mathit{elems}(\gamma')$, we have $y <_\alpha x$ and $x <_\beta y$, so x and y are independent, and by Lemma 2.15, $\gamma' \cdot x \equiv_{\text{str}} x \cdot \gamma'$. By the inductive hypothesis, $\alpha' \equiv_{\text{str}} \gamma \cdot \gamma'$. Thus, by Corollary 2.12, $\alpha = \alpha' \cdot x \equiv_{\text{str}} \gamma \cdot \gamma' \cdot x \equiv_{\text{str}} \gamma \cdot x \cdot \gamma' = \beta$. ■

The property required for Theorem 2.16, that two sequences order conflicting operators in the same way, is called *conflict equivalence* [14]. It is the primary notion of equivalence used in the database literature [35, 14, 51]. Theorem 2.16 says that conflict equivalence implies strong equivalence, which is also called *view equivalence* [14].

Theorem 2.16 has another, perhaps more intuitive proof, which we sketch here. It is based on "transforming" one serialization into another by "swapping" adjacent elements.

Specifically, note that any serialization of a set can be transformed into any other serialization of the same set by a sequence of swaps of adjacent elements. That is, for any two serializations α and β, there exists a finite sequence of intermediate serializations $\gamma_0, \gamma_1, \ldots, \gamma_k$ such that $\gamma_0 = \alpha$, $\gamma_k = \beta$, and for each $i = 1, \ldots, k$, γ_{i-1} and γ_i are the same except for two adjacent elements, whose order has been swapped. Furthermore, this transformation can be done without swapping elements that are in the same order in both α and β. Because the order of conflicting operators is the same in α and β, each intermediate serialization is equivalent to the next in the sequence. By transitivity, this proves that $\alpha \equiv_{str} \beta$. A similar proof can be made for Lemma 2.14.

Often we are given a partial order that orders all the conflicting operators. We can restate Theorem 2.16 so that this partial order is explicit.

Corollary 2.17 Suppose that $\prec$ is a partial order on a set X of operators such that for all $x, y \in X$, either $x \prec y$, $y \prec x$, or x and y are independent. Then all serializations of X consistent with $\prec$ are equivalent.

2.8 Data Type Equivalence

Because clients can access data objects only by their operators and can infer information about their state only from the return values for these operators, clients may not be able to distinguish different data types. We consider such data types *equivalent*. In this section, we introduce two formal notions of data type equivalence: a general and natural notion of equivalence, and a stronger notion that preserves data type properties that the general notion does not. The material in this section is used only in Section 2.10; these sections can be skipped without loss of continuity.

We begin with the general notion of equivalence: Two data types are ***(weakly) equivalent*** if any operator sequence applied to their initial states return the same values for each operator. Formally, $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$ and $\mathcal{D}' = (\Sigma', \hat{\sigma}', O', R', \tau')$ are equivalent if $O = O'$ and for all $\alpha \in O^+$, $\tau^+_R(\hat{\sigma}, \alpha) = \tau'^+_R(\hat{\sigma}', \alpha)$.

Example 2.22 (Statistical Accumulator) The data type of an object that accumulates real numbers and can give the sum and mean of the numbers accumulated is

$$\mathcal{A} = (\ \mathbb{R}^*,\ \epsilon,\ O,\ \mathbb{R} \cup \{\text{ACK}, \text{ERR}\},\ \tau\ ),$$

where $O = \{\mathsf{new}(x) : x \in \mathbb{R}\} \cup \{\mathsf{sum}, \mathsf{mean}\}$, and for all $x \in \mathbb{R}$ and $\alpha \in \mathbb{R}^*$,

$$\tau(\alpha, \mathsf{new}(x)) = (\alpha{\cdot}x,\ \text{ACK})$$

$$\tau(\alpha, \mathsf{sum}) = (\alpha,\ \sum_{i=1}^{n} \alpha_{[i]}) \quad \text{where } n = |\alpha|$$

$$\tau(\alpha, \mathsf{mean}) = \begin{cases} (\alpha,\ \sum_{i=1}^{n} \alpha_{[i]}/n) & \text{where } n = |\alpha| > 0 \\ (\alpha,\ \text{ERR}) & \text{if } \alpha = \epsilon \end{cases}$$

Because the operators only give access to the sum and mean of the statistics accumulated, we can summarize the state using the sum and the number of statistics that have been accumulated.

The mean can be derived from these two values. Thus, although it maintains much less state, the following data type is equivalent to $\mathcal{A}$:

$$\mathcal{A}^* = (\ \mathbb{R} \times \mathbb{N},\ (0,0),\ O,\ \mathbb{R} \cup \{\text{ACK}, \text{ERR}\},\ \tau\ ),$$

where for all $x, s \in \mathbb{R}$ and $n \in \mathbb{N}$,

$$\begin{aligned}
\tau((s,n), \mathsf{new}(x)) &= ((s+x,\ n+1),\ \text{ACK}) \\
\tau((s,n), \mathsf{sum}) &= ((s,n),\ s) \\
\tau((s,n), \mathsf{mean}) &= \begin{cases} ((s,n),\ s/n) & \text{if } n > 0 \\ ((s,n),\ \text{ERR}) & \text{if } n = 0 \end{cases}
\end{aligned}$$

■

There is a natural correspondence between the states of two equivalent data types such that if two objects are in corresponding states and the same operator is applied to both, then they will return the same value and end up in corresponding states. The existence of this correspondence is an alternative characterization of data type equivalence.

Lemma 2.18 Two data types, $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$ and $\mathcal{D}' = (\Sigma', \hat{\sigma}', O, R', \tau')$, are equivalent if and only if there exists a relation $E \subset \Sigma \times \Sigma'$ between states of $\mathcal{D}$ and states of $\mathcal{D}'$ such that $(\hat{\sigma}, \hat{\sigma}') \in E$ and for all reachable states $\sigma \in \Sigma$ and $\sigma' \in \Sigma'$ and all operators $o \in O$, if $(\sigma, \sigma') \in E$ then $(\tau(\sigma, o).s, \tau'(\sigma', o).s) \in E$ and $\tau(\sigma, o).v = \tau'(\sigma', o).v$.

Proof: If $\mathcal{D}$ and $\mathcal{D}'$ are equivalent then let $E = \left\{(\tau^*_\Sigma(\hat{\sigma}, \alpha), \tau'^*_\Sigma(\hat{\sigma}', \alpha)) : \alpha \in O^*\right\}$. We get $(\hat{\sigma}, \hat{\sigma}') \in E$ using $\alpha = \epsilon$. If $(\sigma, \sigma') \in E$ then there exists $\alpha \in O^*$ such that $\sigma = \tau^*_\Sigma(\hat{\sigma}, \alpha)$ and $\sigma' = \tau'^*_\Sigma(\hat{\sigma}', \alpha)$. Thus $(\tau(\sigma, o).s, \tau'(\sigma', o).s) = (\tau^*_\Sigma(\hat{\sigma}, \alpha{\cdot}o), \tau'^*_\Sigma(\hat{\sigma}', \alpha{\cdot}o)) \in E$ and, by the definition of equivalence, $\tau(\sigma, o).v = \tau^+_R(\hat{\sigma}, \alpha{\cdot}o) = \tau'^+_R(\hat{\sigma}', \alpha{\cdot}o) = \tau'(\sigma', o).v$.

If such a relation E exists, then we prove as a sublemma the following statement: For all $\alpha \in O^*$, $(\tau^*_\Sigma(\hat{\sigma}, \alpha), \tau'^*_\Sigma(\hat{\sigma}', \alpha)) \in E$. This sublemma follows by induction on the length of α. If $|\alpha| = 0$ then $\alpha = \epsilon$ and $(\tau^*_\Sigma(\hat{\sigma}, \alpha), \tau'^*_\Sigma(\hat{\sigma}', \alpha)) = (\hat{\sigma}, \hat{\sigma}') \in E$. If $|\alpha| = k + 1 > 0$ then assume the sublemma is true for sequences of length k. Let $\alpha = \alpha'{\cdot}o$, $\sigma = \tau^*_\Sigma(\hat{\sigma}, \alpha')$, and $\sigma' = \tau'^*_\Sigma(\hat{\sigma}', \alpha')$. Then we have $(\sigma, \sigma') \in E$, so $(\tau^*_\Sigma(\hat{\sigma}, \alpha), \tau'^*_\Sigma(\hat{\sigma}', \alpha)) = (\tau(\sigma, o).s, \tau'(\sigma', o).s) \in E$.

If $\alpha \in O^+$ then let $\alpha = \alpha'{\cdot}o$. By the sublemma just proved, we have $(\tau^*_\Sigma(\hat{\sigma}, \alpha'), \tau'^*_\Sigma(\hat{\sigma}', \alpha')) \in E$, so by the definition of E, $\tau^+_R(\hat{\sigma}, \alpha) = \tau(\tau^*_\Sigma(\hat{\sigma}, \alpha'), o).v = \tau'(\tau'^*_\Sigma(\hat{\sigma}', \alpha'), o).v = \tau'^+_R(\hat{\sigma}', \alpha)$. ■

Example 2.23 The correspondence between reachable states of $\mathcal{A}$ and reachable states of $\mathcal{A}^*$ that relates a state $\alpha \in \mathbb{R}^*$ of $\mathcal{A}$ to a state $(s, n) \in \mathbb{R} \times \mathbb{N}$ of $\mathcal{A}^*$ if $n = |\alpha|$ and $s = \sum_{i=1}^{n} \alpha_{[i]}$ meets the conditions specified in Lemma 2.18. ■

Because commutativity depends on the states that result from applying operators, and not merely the return values, weak equivalence need not preserve commutativity. That is, two operators may be commutative in one data type but not in an equivalent one, as we can see in the following example:

Example 2.24 For any $x, y \in \mathbb{R}$, $\mathsf{new}(x)$ and $\mathsf{new}(y)$ commute in $\mathcal{A}^*$ but not in $\mathcal{A}$ (unless $x = y$). ■

We can give a stronger definition of equivalence that does preserve commutativity.[2] This definition uses a correspondence between the states of the two data types, as in the alternative definition of equivalence given in Lemma 2.18, strengthened to preserve commutativity. Two data types, $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$ and $\mathcal{D}' = (\Sigma', \hat{\sigma}', O, R', \tau')$, are ***strongly equivalent*** if there is a bijection f from the reachable states of $\mathcal{D}$ to the reachable states of $\mathcal{D}'$ such that $f(\hat{\sigma}) = \hat{\sigma}'$ and for all reachable states $\sigma \in \Sigma$ and $o \in O$, $f(\tau(\sigma, o).s) = \tau'(f(\sigma), o).s$ and $\tau(\sigma, o).v = \tau'(f(\sigma), o).v$. It follows immediately from Lemma 2.18 that strong equivalence implies (weak) equivalence.

Corollary 2.19 If $\mathcal{D}$ and $\mathcal{D}'$ are strongly equivalent then they are equivalent.

Proof: Use the bijection as the relation needed in Lemma 2.18. ■

Example 2.25 We give a strongly equivalent data type for the bank account data type from Example 2.4. Assuming $\mathcal{B}$ maintained the balance in dollars as the state, this data type maintains the balance in cents.

$$\mathcal{B}' = (\ \mathbb{N},\ 0,\ O,\ \mathbb{N} \cup \{\text{ACK}, \text{ERR}\},\ \tau\ ),$$

where $O = \{\mathsf{balance}\} \cup \{\mathsf{deposit}(k) : k \in \mathbb{N}\} \cup \{\mathsf{withdraw}(k) : k \in \mathbb{N}\}$,
and for all $n, k \in \mathbb{N}$:

$$\begin{aligned}
\tau(n, \mathsf{balance}) &= (n,\ n/100) \\
\tau(n, \mathsf{deposit}(k)) &= (n + 100k,\ \text{ACK}) \\
\tau(n, \mathsf{withdraw}(k)) &= \begin{cases} (n - 100k,\ k) & \text{if } 100k \leq n \\ (n,\ \text{ERR}) & \text{if } 100k > n. \end{cases}
\end{aligned}$$

Note that the reachable states of $\mathcal{B}'$ are the multiples of 100. The bijection from reachable states of $\mathcal{B}$ to reachable states of $\mathcal{B}'$ maps n to $100n$. ■

Strong equivalence does preserve operator commutativity.

Lemma 2.20 If $\mathcal{D}$ and $\mathcal{D}'$ are strongly equivalent then operators that commute in $\mathcal{D}$ also commute in $\mathcal{D}'$.

Proof: Let $o, o' \in O$ be two operators that commute in $\mathcal{D}$, and f be a bijection from reachable states of $\mathcal{D}$ to the reachable states of $\mathcal{D}'$ that establishes the strong equivalence of $\mathcal{D}$ and $\mathcal{D}'$. For any reachable state $\sigma' \in \Sigma'$ of $\mathcal{D}'$, $\sigma = f^{-1}(\sigma')$ is a reachable state of $\mathcal{D}$, so

[2] Alternatively, we can redefine commutativity so that changing the order of two operators results in "equieffective" states—states that cannot be distinguished by the return values for any operator sequence applied to them [80]. This preserves all the essential properties of commutativity and is preserved by weak equivalence. However, we do not need this additional complexity for this thesis.

$$\begin{aligned}
\tau'^{*}_{\Sigma}(\sigma', \langle o, o' \rangle) &= \tau'(\tau'(f(\sigma), o).s, o').s && \text{by the definitions of } \tau'^{*}_{\Sigma} \text{ and } \sigma \\
&= \tau'(f(\tau(\sigma, o).s), o').s && \text{by strong equivalence} \\
&= f(\tau(\tau(\sigma, o).s, o').s) && \text{by strong equivalence} \\
&= f(\tau^{*}_{\Sigma}(\sigma, \langle o, o' \rangle)) && \text{by the definition of } \tau^{*}_{\Sigma} \\
&= f(\tau^{*}_{\Sigma}(\sigma, \langle o', o \rangle)) && \text{because } o \text{ and } o' \text{ commute in } \mathcal{D} \\
&= f(\tau(\tau(\sigma, o').s, o).s) && \text{by the definition of } \tau^{*}_{\Sigma} \\
&= \tau'(f(\tau(\sigma, o').s), o).s && \text{by strong equivalence} \\
&= \tau'(\tau'(f(\sigma), o').s, o).s && \text{by strong equivalence} \\
&= \tau'^{*}_{\Sigma}(\sigma', \langle o', o \rangle) && \text{by the definitions of } \tau'^{*}_{\Sigma} \text{ and } \sigma
\end{aligned}$$

■

2.9 Locations

Some data types have a natural notion of *locations* such that each operator acts on a single location. In this section, we formalize this notion and explore properties of these data types that make them easier to reason about.

We first formalize the intuition that a data type can be partitioned into independent locations. A ***location partition*** $\{O_\ell\}_{\ell \in \mathcal{L}}$ of $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$ is a partition of O such that elements from different classes are independent; that is, for all $o \in O_\ell$ and $o' \in O_{\ell'}$ with $\ell \neq \ell'$, o and o' are independent. The set $\mathcal{L}$ is called the set of ***locations***, and $o.loc$ denotes the location ℓ such that $o \in O_\ell$. We say that o is ***performed on*** $o.loc$. The notation $o.loc$ is well-defined only for a given location partition; a data type may have several location partitions. We also use $\alpha|_\ell$ and $S|_\ell$ as shorthand for $\alpha|_{O_\ell}$ and $S \cap O_\ell$ respectively.

Example 2.26 For the read/write memory $\mathcal{M}$, each address is a location. Formally, $\{O_a\}_{a \in A}$ is a location partition of $\mathcal{M}$, where $O_a = \{\mathsf{read}(a)\} \cup \{\mathsf{write}(a, k) : k \in \mathbb{Z}\}$. With this location partition, $\mathsf{read}(a)$ and $\mathsf{write}(a, k)$ are performed on a, which justifies the original use of this terminology in Example 2.2. ■

For a data type with a location partition, we usually imagine that the state is divided into independent locations, each of which is affected only by operators performed on that location. To determine the effect of an operator, it is sufficient to know the operators that were applied before it on the same location. That is, for any operator sequence, the values returned for the operators of one location by that sequence are the same as the values returned for those operators by the subsequence in which only those operators appear; they are not affected by the application of any operators to other locations. Formally, we have the following lemma and corollary:

Lemma 2.21 If $\{O_\ell\}_{\ell \in \mathcal{L}}$ is a location partition of $\mathcal{D}$ then $\tau^{+}_{R}(\sigma, \alpha \cdot o) = \tau^{+}_{R}(\sigma, \alpha|_\ell \cdot o)$, where $\ell = o.loc$.

Proof: By induction on the length of α. For $\alpha = \epsilon$, this is trivially true. Otherwise, assume that the lemma is true for all sequences shorter than α, and let α' be such that $\alpha = o' \cdot \alpha'$. We have:

$$\begin{aligned}
\tau^{+}_{R}(\sigma, \alpha \cdot o) &= \tau^{+}_{R}(\tau(\sigma, o').s, \alpha' \cdot o) && \text{by the definitions of } \tau^{+} \text{ and } \tau^{+}_{R} \text{ since } \alpha = o' \cdot \alpha' \\
&= \tau^{+}_{R}(\tau(\sigma, o').s, \alpha'|_\ell \cdot o) && \text{by the inductive hypothesis} \\
&= \tau^{+}_{R}(\sigma, o' \cdot \alpha'|_\ell \cdot o) && \text{by the definitions of } \tau^{+} \text{ and } \tau^{+}_{R}
\end{aligned}$$

If $o'.loc = \ell$ then $\alpha|_\ell = o' \cdot \alpha'|_\ell$, which completes the proof. Otherwise, $\alpha|_\ell = \alpha'|_\ell$, o' commutes with every operator in $\alpha'|_\ell$, and o' and o are independent. In this case, we have:

$$\begin{aligned}
\tau_R(\tau_\Sigma^*(\sigma, o' \cdot \alpha'|_\ell), o) &= \tau_R(\tau_\Sigma^*(\sigma, \alpha|_\ell \cdot o'), o) && \text{by Lemma 2.6 and since } \alpha|_\ell = \alpha'|_\ell \\
&= \tau_R^+(\tau_\Sigma^*(\sigma, \alpha|_\ell), \langle o', o \rangle) && \text{by Lemma 2.2} \\
&= \tau_R^+(\tau_\Sigma^*(\sigma, \alpha|_\ell), \langle o \rangle) && \text{because } o \text{ is oblivious to } o' \\
&= \tau_R^+(\sigma, \alpha|_\ell \cdot o) && \text{by Lemma 2.2}
\end{aligned}$$

■

Corollary 2.22 If $\{O_\ell\}_{\ell \in \mathcal{L}}$ is a location partition of $\mathcal{D}$, α is a valid operator sequence of $\mathcal{D}$, and $x \in \mathit{elems}(\alpha)$ then $\mathit{retval}(x, \alpha) = \mathit{retval}(x, \alpha|_\ell)$, where $\ell = x.\mathit{loc}$.

Because operators on different locations are independent, we can reorder any operator sequence to obtain an equivalent sequence in which all operators performed on the same location are applied consecutively.

Lemma 2.23 If $\{O_\ell\}_{\ell \in \mathcal{L}}$ is a location partition of $\mathcal{D}$ and $\alpha \in O^*$ with operators performed on locations $\ell_1, \ell_2, \ldots, \ell_n$ then $\alpha \equiv_{\mathrm{str}} \alpha|_{\ell_1} \cdot \alpha|_{\ell_2} \cdot \cdots \cdot \alpha|_{\ell_n}$.

Proof: Immediate from Corollary 2.17 using $\bigcup_{\ell \in \mathcal{L}} \leq_{\alpha|_\ell}$ as the partial order. ■

It follows immediately from this lemma and Corollary 2.12 that operators sequences that are equivalent for each location are equivalent.

Corollary 2.24 If $\alpha|_\ell \equiv_{\mathrm{str}} \beta|_\ell$ for all $\ell \in \mathcal{L}$ then $\alpha \equiv_{\mathrm{str}} \beta$.

2.10 Data Type Composition

An object with a data type that can be partitioned into locations may also be viewed as the composition of several independent objects. Thus, the data type of the composite object may be characterized as the composition of several data types. In this section, we define *data type composition*, and show how this is related to location partitions. The material in this section is not used in the rest of this thesis and may be skipped without loss of continuity.

A family $\{\mathcal{D}_i\}_{i \in I}$ of data types, with $\mathcal{D}_i = (\Sigma_i, \hat{\sigma}_i, O_i, R_i, \tau_i)$, is ***compatible*** if the sets of operators of the data types are pairwise disjoint, that is, $O_i \cap O_j = \emptyset$ when $i \neq j$. The ***composition*** of a compatible family of data types is

$$\prod_{i \in I} \mathcal{D}_i = \left(\prod_{i \in I} \Sigma_i,\ (\hat{\sigma}_i)_{i \in I},\ \bigcup_{i \in I} O_i,\ \bigcup_{i \in I} R_i,\ \tau \right)$$

where, for $o \in O_j$, $\tau((\sigma_i)_{i \in I}, o) = ((\sigma_i')_{i \in I},\ \tau_j(\sigma_j, o).v)$, with $\sigma_j' = \tau_j(\sigma_j, o).s$ and $\sigma_i' = \sigma_i$ for $i \neq j$. The state of the composite data type is a tuple with one component for each data type being composed, where the ith component is a state of $\mathcal{D}_i$, and the initial state is tuple with the initial states of the component data types. Each operator of the composite data type belongs to exactly of the component data types by the compatibility requirement. Its effect is to be applied to the appropriate component of the composite state, modifying it and determining a return value, and leaving all other components unchanged.

Example 2.27 (Read/Write Memory revisited) A read/write memory $\mathcal{M}$ with addresses A is the composition of a family of read/write registers indexed by A. Formally, $\mathcal{M} = \prod_{a \in A} \mathcal{R}_a$, where $\mathcal{R}_a = (\mathbb{Z}, 0, O_a, \mathbb{Z} \cup \{\text{ACK}\}, \tau_a)$, and $O_a = \{\mathsf{read}(a)\} \cup \{\mathsf{write}(a,k) : k \in \mathbb{Z}\}$, and $\tau_a(n, \mathsf{read}(a)) = (n, n)$ and $\tau_a(n, \mathsf{write}(a,k)) = (k, \text{ACK})$. ■

Because each operator affects only its own component, each component of the state is determined by the subsequence of operators that have been applied from that component.

Lemma 2.25 For any $\alpha \in (\bigcup_{i \in I} O_i)^*$ and $\sigma_i \in \Sigma_i$, $\tau^*_\Sigma((\sigma_i)_{i \in I}, \alpha) = (\tau^*_{i\Sigma}(\sigma_i, \alpha|_{O_i}))_{i \in I}$.

Proof: Straightforward induction on the length of α. ■

Location partitions and composite data types are two ways of expressing the same notion. Specifically, the component data types give a location partition of a composite data type. Similarly, any location partition defines a corresponding compatible family of data types whose composition is equivalent to the original data type. These are established by the following lemmas:

Lemma 2.26 $\{O_i\}_{i \in I}$ is a location partition of $\prod_{i \in I} \mathcal{D}_i$.

Proof: Suppose $o \in O_j$ and $o' \in O_{j'}$ for some $j \neq j'$. We need to show that o and o' are independent; that is, that o and o' commute and are oblivious to each other.

Suppose $(\sigma_i)_{i \in I} \in \prod_{i \in I} \Sigma_i$ is a reachable state of $\prod_{i \in I} \mathcal{D}_i$. Let $\sigma'_j = \tau_j(\sigma_j, o)$ and $\sigma'_i = \sigma_i$ for $i \neq j$, and $\sigma''_j = \tau_j(\sigma_j, o)$, $\sigma''_{j'} = \tau_{j'}(\sigma_{j'}, o')$, and $\sigma''_i = \sigma_i$ for $i \neq j, j'$. Note that $\tau^*_\Sigma((\sigma_i)_{i \in I}, \langle o, o' \rangle) = \tau(\tau((\sigma_i)_{i \in I}, o).s, o').s = \tau((\sigma'_i)_{i \in I}, o').s = (\sigma''_i)_{i \in I}$. Similarly, we can show that $\tau^*_\Sigma((\sigma_i)_{i \in I}, \langle o', o \rangle) = (\sigma''_i)_{i \in I}$, which proves o and o' commute.

To see that o' is oblivious to o, note that $\tau^+_R((\sigma_i)_{i \in I}, \langle o, o' \rangle) = \tau((\sigma'_i)_{i \in I}, o').v = \tau_{j'}(\sigma'_{j'}, o').v = \tau_{j'}(\sigma_{j'}, o').v = \tau((\sigma_i)_{i \in I}, o').v = \tau^+_R((\sigma_i)_{i \in I}, \langle o' \rangle)$, Similarly, we can show that o is oblivious to o'. ■

Lemma 2.27 Suppose $\{O_\ell\}_{\ell \in \mathcal{L}}$ is a location partition of $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$. Then $\mathcal{D}$ is equivalent to $\prod_{\ell \in \mathcal{L}} \mathcal{D}_\ell$, where $\mathcal{D}_\ell = (\Sigma, \hat{\sigma}, O_\ell, R, \tau|_{\Sigma \times O_\ell})$.

Proof: Consider the relation $E = \{(\tau^*_\Sigma(\hat{\sigma}, \alpha), (\tau^*_\Sigma(\hat{\sigma}, \alpha|_\ell))_{\ell \in \mathcal{L}}) : \alpha \in O^*\}$. We get $(\hat{\sigma}, (\hat{\sigma})_{\ell \in \mathcal{L}}) \in E$ using $\alpha = \epsilon$. If $(\sigma, (\sigma_\ell)_{\ell \in \mathcal{L}}) \in E$ then there exists $\alpha \in O^*$ such that $\sigma = \tau^*_\Sigma(\hat{\sigma}, \alpha)$ and $\sigma_\ell = \tau^*_\Sigma(\hat{\sigma}, \alpha|_\ell)$ for all $\ell \in \mathcal{L}$. Let τ' be the transition function of $\prod_{\ell \in \mathcal{L}} \mathcal{D}_\ell$. Then, for all $o \in O$, $(\tau(\sigma, o).s, \tau'((\sigma_\ell)_{\ell \in \mathcal{L}}, o).s) = (\tau^*_\Sigma(\hat{\sigma}, \alpha \cdot o), (\tau^*_\Sigma(\hat{\sigma}, (\alpha \cdot o)|_\ell))_{\ell \in \mathcal{L}}) \in E$. By Lemma 2.21 and the definition of data type composition, $\tau(\sigma, o).v = \tau^+_R(\hat{\sigma}, \alpha \cdot o) = \tau^+_R(\hat{\sigma}, \alpha|_{o.loc} \cdot o) = \tau(\sigma_{o.loc}, o).v = \tau'((\sigma_\ell)_{\ell \in \mathcal{L}}, o).v$. Thus, by Lemma 2.18, $\mathcal{D}$ and $\prod_{\ell \in \mathcal{L}} \mathcal{D}_\ell$ are equivalent. ■

The transformation of locations into separate data types may not preserve strong equivalence, as demonstrated by the following example:

Example 2.28 Consider the data type $(\mathbb{Z}, 0, \{\mathsf{inc}, \mathsf{dec}\}, \{\text{ACK}\}, \tau)$, where $\tau(n, \mathsf{inc}) = (n+1, \text{ACK})$ and $\tau(n, \mathsf{dec}) = (n-1, \text{ACK})$. The inc and dec operators are independent, so $\{\{\mathsf{inc}\}, \{\mathsf{dec}\}\}$ is a location partition of this data type. However, there is no bijection between the reachable states of these data types, because $(0,0)$ and $(1,-1)$, both reachable states in the composite data type, must correspond to the single state 0 in the original data type. ■

2.11 Discussion

The rigor of our treatment of data types is unusual in the development of memory models. Most memory models assume a particular data type, usually read/write memory, without formally defining the properties of this data type [33, 2]. Instead, they rely on intuitive properties of read/write memory. This approach is attractive since most memories are some variant of read/write memory, and the serial semantics of these memories is simple. However, there are several advantages to the formal approach taken in this thesis.

First, defining the serial semantics formally allows us to reason with complete rigor. There is no ambiguity, for example, caused by primitives such as test-and-set, which both read and write the memory. In release consistency [43, 41], for example, such primitives must be treated as combinations of reads and writes. This rigor is helpful for exposing assumptions about the system, as it was in our analysis of a lazy replication algorithm for a highly available data service [36, 66].

Second, this abstract approach to data types provides greater modularity, and affords the possibility of memories with high-level data types. This is especially important when the memory being modeled is not a shared memory multiprocessor but a virtual shared memory for a high level programming language with data abstraction facilities. Our theory of data types allows a clear separation of the serial and the concurrent aspects of the memory semantics.

Third, we are able to identify the exact properties that the results we derive depend on. For example, Shasha and Snir define *conflict* in terms of reads and writes [97], and establish results for operators that do not conflict. In this thesis, we redefine the conflicts relation for general operators, and show that these results hold using the new definition. In addition, we show that two writes to the same location do *not* conflict if they write the same value. Although this distinction is insignificant in practice, it demonstrates that this abstract approach can improve our understanding of data type properties; in some cases, there may be significant practical consequences.

In formalizing data types, we have kept the definitions simple and straightforward. We maintain a clean split between what the clients specify—the operator sequence—and what the memory specifies in response—the return value function. This split allows us to focus on equivalence, which will be crucial when we study concurrent memory.

Two other approaches to formally defining the serial semantics of data types have been explored in the literature. The first is to describe the memory as a state machine including both the serial and concurrent aspects. This approach allows more general data types than our serial data types, because the state machine may be nondeterministic and can block. However, it can be difficult to isolate the properties inherent in the data type from those that arise because of concurrent access of the data; both types of properties are expressed by a state machine. This approach has been used to describe strongly consistent memories with arbitrary data types [69, 79, 70] and weakly consistent memories with read/write memory [45, 44, 70]; we do not know of any cases where it has been used to study weakly consistent memories with arbitrary data types.

The second approach, introduced by Herlihy and Wing [53], defines the *sequential specification* for a data object to be a prefix-closed set of *legal sequential histories* for that object

where a sequential history is a sequence of invocation-response event pairs.[3] In a legal sequential history, each invocation specifies the operator to be applied to the object, and it is immediately followed by a matching response, which specifies the return value resulting from applying the operator to the current state of the object. Other researchers have adapted their general framework for modeling weakly consistent memories, but then specify guarantees in terms of reads and writes [11, 55].

Characterizing data types by the legal sequential histories is also more general than our approach because it allows data objects that may block or respond nondeterministically to an invocation. On the other hand, it is harder to identify operator properties such as independence from the legal histories.

In their work on nested transactions, Lynch, et al. combine some of the advantages of all these methods by defining a data type using *object automata* [80]. Object automata have a lot of structure that we do not need in this thesis, and the semantics of the data object is defined by the behaviors of its object automaton, which is less clean a split than the operator sequences and return value functions in this chapter.

The theory of data types presented here is only a beginning, sufficient for the purposes of this thesis. There are several directions that seem worth pursuing. We could characterize data types and their operators in other ways, such as the read-only and transparent operators mentioned in Section 2.4, and see what results we can derive from them. Similarly, we might define alternative notions of operator sequence equivalence. For example, it may be useful, especially in the study of *transactional memories* introduced in Chapter 8, to designate some operators as externally visible and consider two operator sequences equivalent when they yield the same return values for the externally visible operators. Finally, a more ambitious task would be to extend this theory to nondeterministic and blocking data types, in which there may be multiple (in the case of nondeterministic data types) or no (in the case of blocking data types) return values and final states for an operator applied to a specific state.

[3] Herlihy and Wing call these pairs *operations*. As we see in Chapter 3, we use that term to denote only the invocation of a data type operator, without the response.

Chapter 3

Computations

In this chapter, we define the interface between the clients and the memory system when the clients may access the memory concurrently. In the serial setting, the clients specify a sequence of data type operators, and the memory specifies a return value function for the operators in the sequence. In the concurrent setting, the memory still specifies a return value function, but the clients specify the operators using a *computation*. We define computations, illustrate their use with several examples, describe how they are generated by programs, and prove a few results that are useful in developing the theory of memory models described in this thesis. Because a memory model depends on the interface with the clients, computations are the foundation of our framework.

A computation represents the clients' requests to the memory. It specifies the operators to be applied to the data object and restrictions on the order in which these operators may be applied. The memory system has no access to the state of the clients except as specified by the computation. To determine return values for the operators, the system applies the operators according to a *schedule* that respects the constraints specified by the computation. Because the clients may access the memory concurrently, a computation requires a structure richer than a sequence; there may be several allowable schedules.

Computations have two mechanisms to constrain the possible schedules: *precedence dependencies* and *annotations*. We show how to use these mechanisms to model the clients' requests to the memory as computations. A precedence dependency specifies that one operator should be applied before another. Together, the precedence dependencies define a partial order on the operators, which corresponds to a logical notion of time. Operations are *concurrent* if the computation does not specify the order in which they must be applied.

Annotations provide a general mechanism for specifying other constraints provided by systems. Because the constraints allowed by systems vary, the possible annotations depend on the system being modeled. The set of computations for a memory system is parameterized by the data type and the annotations for that system.

As in Chapter 2, clients can access the state of the memory only through the operators; they see only the values returned by the memory. The state of the memory and the order in which the operators are applied can only be inferred from the return values. Equivalent schedules, therefore, can never be distinguished by the clients.

Our use of computations is derived its use in the Cilk project [105].

We also identify *races* in computations, and show that computations without any races are *determinate*; that is, all their schedules are equivalent. As we see in later chapters, determinate computations are easy to reason about.

Outline: Section 3.1 defines graph theoretic notation used in this chapter and throughout the thesis. Section 3.2 formally defines computations. Section 3.3 contrasts computations with programs and parallel instruction streams. Section 3.4 discusses annotations in more detail, and gives several examples of possible annotations. In Section 3.5, we give several simple programs and the computations they generate. Section 3.6 defines schedules of computations, which describe the case when the computation is executed on a sequential machine. In Section 3.7, we formally define races in computations, and show how the behavior of computations is easier to reason about in the absence of races. Finally, in Section 3.8, we contrast computations with alternative approaches to modeling client requests, and examine various issues that motivate the definition of computations.

Reading Guide: Despite its length, this chapter defines only a few simple concepts: computations, schedules, races and determinacy. It is possible to follow the rest of the thesis after reading only Sections 3.2, 3.6, and 3.7. The rest of the chapter motivates and explains, with lots of examples, our use of computations as the basis for a framework for memory consistency models. In particular, Section 3.5 illustrates the relationship between computations and programs, whose behavior is ultimately the object of programmers' interest. Also, many examples in Sections 3.4 and 3.5 prefigure some of the memory models defined in Chapters 5, 6, 7, and 8.

3.1 Preliminary Graph Theory

This section presents terminology and notation for familiar graph theoretic concepts used in this thesis. It is intended primarily as a reference for later sections.

All graphs in this thesis are *directed*. For a graph G, we use V_G to denote the set of *vertices* and E_G to denote the set of *edges*, which is a binary relation on V_G. The *subgraph (of G) induced by* $S \subseteq V_G$ is denoted by $G|_S = (S, E_G|_S)$.

A ***path*** in a graph G from a vertex u to v is a finite nonempty sequence $u_0, u_1, \ldots, u_k$ of vertices such that $u = u_0$, $v = u_k$ and $(u_{i-1}, u_i) \in E_G$ for all $i = 1, 2, \ldots, k$. A vertex u is ***reachable*** from another vertex v if there is a path from v to u. A vertex is a ***root*** if every vertex is reachable from it. It is an ***inverse root*** if it is reachable from every vertex.

A ***dag*** is an acyclic graph; that is, G is a dag if $\mathrm{TC}(E_G)$, the transitive closure of the edge relation, is a strict partial order, called the ***precedence order***. We denote the precedence order by $\prec_G$ and its reflexive closure by $\preceq_G$. We may omit the subscript if the dag is clear from context. We have $u \preceq v$ if and only if there is a path in G from u to v; we say that u is ***ordered by*** G ***before*** v. If $u \not\preceq v$ and $v \not\preceq u$, we say u and v are ***not ordered*** by G.

A ***topological sort*** of a dag G is any serialization of V_G consistent with $\prec_G$. A ***prefix*** of a dag G is any induced subgraph G' closed under adding predecessors; that is, if $u \preceq_G v$ and $v \in V_{G'}$ then $u \in V_{G'}$. The dag G ***extends*** G', or is an ***extension*** of G', if G' is a prefix of G.

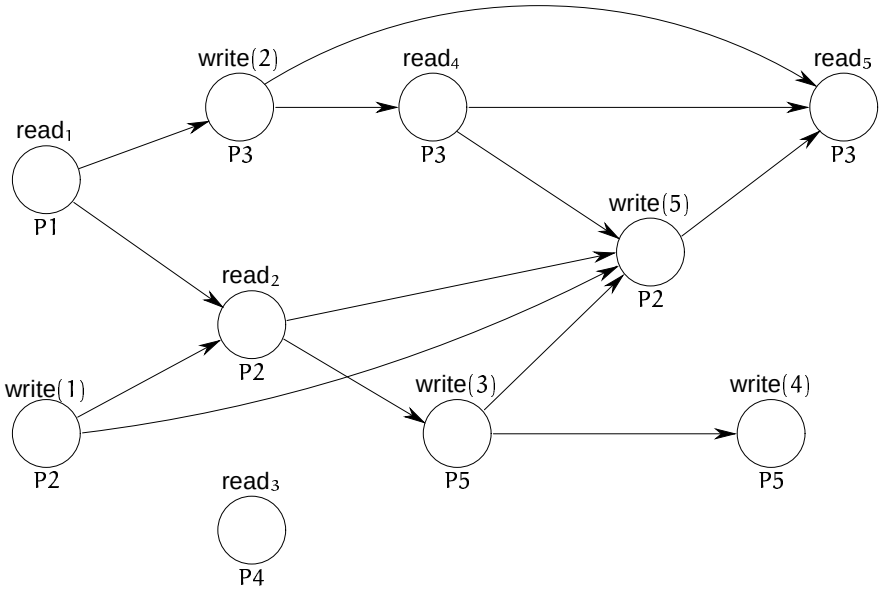
3.2 Definition

A *computation* describes the clients' requests to the memory. It specifies the operators invoked, along with constraints on how these operators may be applied. The operators that may be invoked are specified by the data type of the memory. A client may specify a *precedence dependency* between two operators, that is, that one operator must be applied before the other may be applied. Other constraints are specified using *annotations*. The possible annotations vary from system to system, and must be specified by the memory.

Formally, given a data type $\mathcal{D}$ and a set A of annotations, a ***computation*** C on $\mathcal{D}$ with A is an annotated dag whose vertices V_C are operators of $\mathcal{D}$. The edges E_C of the dag encode the ***precedence dependencies***, and the ***annotation function*** $ann_C : V_C \to A$ encodes the annotations. We use $\mathfrak{C}_A^{\mathcal{D}}$ to denote the set of computations on $\mathcal{D}$ with A.

Though they are just data type operators, we often refer to the vertices of computations as ***operations***. That is, we use *operation* to refer to an operator in the context of a particular computation, which associates precedence dependencies and an annotation with the operator. We think of an operation as being an invocation of an operator.

Example 3.1 A graphical representation of a computation on $\mathcal{R}$ with $\{P1, P2, P3, P4, P5\}$, where the annotation indicates the process requesting the operation:



Each circle represents a vertex of the computation, with the operation indicated above the circle, and the annotation indicated below it. The arrows represent the edges of the computation.

Note that the data type of this computation is actually $\mathcal{R}$ extended with subscripts to distinguish multiple invocations of the read operator, as described in Section 2.5. ■

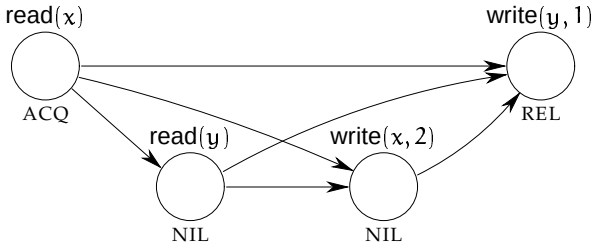
Given the order in which the operations are applied, the data type specifies the effect on the data object and the return value for each operation. The precedence dependencies and the annotations specify restrictions on how the operations may be applied. Precedence constraints—that one operation should be applied before another—are expressed using precedence dependencies. Other constraints are expressed using the annotations.

Because the constraints that clients may express vary among systems, the annotation set appropriate for each system also varies.

The clients may specify arbitrary precedence dependencies, including totally ordering all operations, or not specifying any dependencies at all. The only requirement is that there may not be any cyclic dependencies; that is, the computation must be a dag.

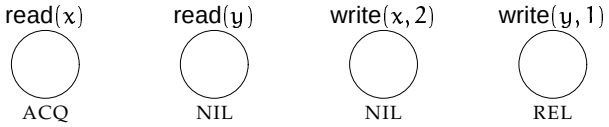
Example 3.2 Some systems provide *locks* that can be *acquired* and *released*. Informally, a lock that has been acquired must be released before it can be acquired again. We discuss how to model this formally in later chapters, particularly Chapter 7. For now, we model such systems using the annotation set $\{\text{ACQ}, \text{REL}, \text{NIL}\}$, where NIL is the annotation for operations that neither acquire nor release the lock.

Here is a computation on $\mathcal{M}$ with $\{\text{ACQ}, \text{REL}, \text{NIL}\}$ whose operations are totally ordered:



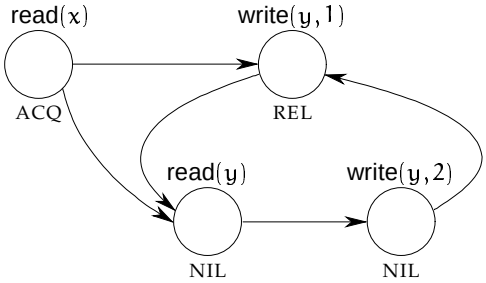
■

Example 3.3 The computation from the previous example with no precedence dependencies:



■

Example 3.4 (Cyclic Dependencies) The following is *not* a computation because it has cyclic dependencies.



■

Because the constraints that clients may express vary among memory systems, the annotation set appropriate for each system also varies. In previous examples, we saw computations annotated by processors and computations with a lock. Some systems provide mechanisms, such as *synchronization variables*, which require different annotation sets. We introduce a few types of annotations informally in Section 3.4, and we develop them formally in later chapters.

Some systems place additional restrictions on the ways that computations may be annotated. These restrictions, called *well-formedness conditions*, vary from system to system, even among systems that have the same annotation set. In some cases, the well-formedness condition merely reflects a limitation in the interface between the clients and the memory. In other cases, the constraints corresponding to the computations that are not well-formed are not meaningful. We discuss these conditions more carefully in the context of the memory models that use them.

Example 3.5 Consider a system in which processors are *blocking*; that is, a processor has at most one outstanding operation requested. In such systems, the memory is often required to preserve the order in which operations are requested by each processor. We model this behavior by annotating each operation with the processor that requests it, and requiring operations annotated by the same processor to be totally ordered. In this case, the computation from Example 3.1 is well-formed. ■

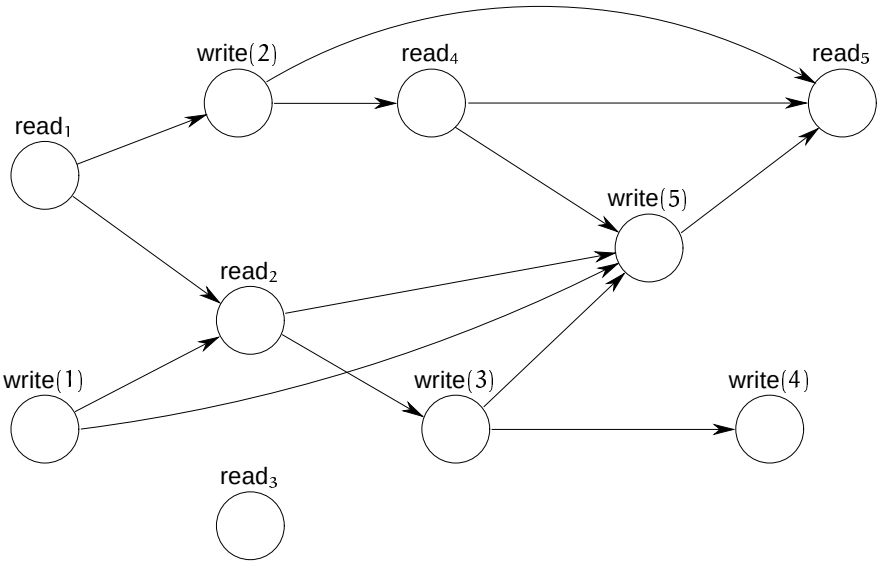
Example 3.6 Consider a system in which processors request operations independently; that is, an operation cannot depend on an operation requested by a different processor. The well-formedness condition for such a system is that only operations requested by the same processor may be ordered. This condition arises from the inability for processors to explicitly express dependencies on operations requested by other processors. In this case, the computation from Example 3.1 is *not* well-formed. ■

Example 3.7 Consider the system described in Example 3.2, in which a lock may be acquired and released. Such a system typically requires a lock to be acquired before it can be released. We model this behavior by a well-formedness condition requiring every operation annotated by REL to be preceded by an operation annotated by ACQ. In this case, the computation from Example 3.2 is well-formed, but the computation from Example 3.3 is not. ■

Often, we want to annotate only some of the operations. Formally, we use a default annotation NIL for the operations that do not need to be annotated. Henceforth, we assume that every annotation set includes NIL. We say that an operation annotated by NIL has no annotation, and we often omit writing the NIL, either in a diagram or in the annotation set.

Some systems allow clients to specify only precedence dependencies. That is, the only constraints on the way operations may be applied can be expressed by precedence dependencies. Formally, the annotation set is the singleton set $\{\text{NIL}\}$. In this case, we often leave out the annotations altogether, writing $\mathfrak{C}^{\mathcal{D}}$ instead of $\mathfrak{C}^{\mathcal{D}}_{\{\text{NIL}\}}$ or $\mathfrak{C}^{\mathcal{D}}_{\emptyset}$, for example.

Example 3.8 The computation from Example 3.1 without annotations:



■

The precedence dependencies define a partial order on the operations of a computation, which corresponds to a logical notion of time. An operation is logically prior to operations that depend on it and logically after operations it depends on. No other timing information is preserved by the computation.

Formally, we denote the partial order induced by the precedence dependencies of a computation C by $\prec_C = \mathrm{TC}(E_C)$, the transitive closure of the edge relation; its reflexive closure is $\preceq_C$. We drop the subscript when the computation is clear from context. An operation x ***(logically) precedes*** another operation y in a computation C if $x \prec_C y$. Two distinct operations are ***(logically) concurrent*** in C if neither precedes the other; that is, if $x \not\preceq_C y$ and $y \not\preceq_C x$. An operation is not concurrent with itself.

Precedence in logical time neither implies nor is implied by precedence in real time. Two logically concurrent operations might not actually overlap in time. Conversely, a nonblocking client may specify that a new operation depends on a previously requested but still outstanding operation. Thus, two operations that are not logically concurrent may be outstanding at the same time.

Precedence dependencies are intended to capture "control" dependencies, which depend on the structure of the program generating the computation, rather than "data" dependencies, which depend on the values returned by the memory. The distinction between control dependencies and data dependencies is not always so clear, however. Section 3.5 gives several examples of computations corresponding to programs that are helpful in understanding this distinction.

Because precedence dependencies represent the control dependencies derived from the program, the partial order defined by the precedence dependencies is sometimes called the *program order*. However, as we discuss in Section 3.5, the relationship between a computation and the program that generated it can be difficult to characterize formally. Thus, we use this term only informally.

Annotations provide a general mechanism for expressing constraints that cannot be expressed using precedence dependencies. With an appropriate choice of the annotation set, almost any kind of constraint on the way operations may be applied can be expressed

using annotations. For example, precedence dependencies can be specified by annotating each operation with the operations that must precede it. We treat precedence dependencies specially because they are common to all memory systems, and it is convenient to take advantage of the greater structure provided by a dag to express them.

A computation is a static representation of the clients' requests. It can be thought of as a snapshot of an execution, with everything other than the requested operations and the constraints on how they may be applied, abstracted away. A computation makes no guarantees about the values that the memory returns for the operations; these guarantees are specified by the memory model. Although the clients select a computation assuming a particular memory model, this model may not characterize the actual guarantees of the memory being accessed. For example, we show in various results throughout this thesis that clients that obey certain restrictions in how they access the memory may assume strong consistency guarantees even when the guarantees of the actual system are weak. Providing a framework for specifying and reasoning about memory models is the goal of this thesis, and these results are its main results.

3.3 What Computations Are Not

Because computations are abstract representations of the requests that the clients make of the memory, they may be confused with programs and instruction streams. However, although computations bear similarities to both programs and instruction streams, they differ in important ways.

Computations are not programs. A program generates a computation every time it executes, but it may not generate the same computation each time. Rather, the computation it generates may depend on nondeterministic choices made in the inputs to the program, or by the memory in returning values for the operations, or by the processor running the program, possibly due to unpredictable timing conditions, or *races*.

Furthermore, a computation has no control structures such as loops or branches. Each operation in the computation is invoked exactly once. If a statement in a program is executed several times, perhaps because it appears in the body of a loop, it generates several vertices in the computation, one for every time it is executed. If it is never executed, perhaps because it appears in a branch not taken, it does not generate any vertex in the computation. In Section 3.5, we consider several programs and the computations they generate.

Computations are not instruction streams, though they resemble instruction streams more than they resemble programs. An instruction stream is a sequence of instructions sent to a processor. Computations model the clients' requests to memory. In a shared memory multiprocessor, for example, the clients of the memory are the processors, and computations model the requests of the processors to access memory.

Computations also differ from instruction streams in that they explicitly note the dependencies between operations. Many high performance processors reorder the instructions they receive to achieve better performance. High performance memories may also

apply operations in different orders, but a computation specifies a partial order to be respected, rather than a total order that may be relaxed.

3.4 Some Types of Annotations

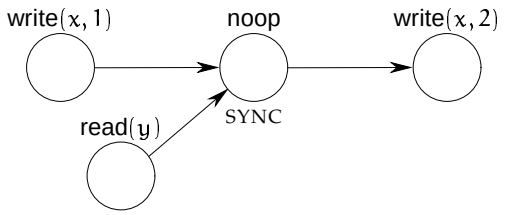
As mentioned earlier, the annotation set depends on the kinds of constraints the memory system understands, and must be specified by the memory system along with the data type. A few types of annotations can express the constraints understood by most systems. In this section, we introduce some of these types of annotations briefly and informally. In later chapters, we give formal definitions and discuss in greater depth the interpretation of specific annotations in the context of particular memory models.

Precedence-based systems. Some systems allow only precedence dependencies to be expressed. We call such systems *precedence-based*. In this case, the computation has no annotations. Example 3.8 gives such a computation. We discuss precedence-based systems in Chapter 5.

Memory barriers and synchronization. A system may provide special *synchronization* operations, such as *memory barriers*. We use annotations to indicate whether an operation synchronizes with other operations; the annotation set is {SYNC}. Typically, systems guarantee greater consistency for synchronization operations than for other operations; the exact guarantees vary among systems and are specified by the memory model.

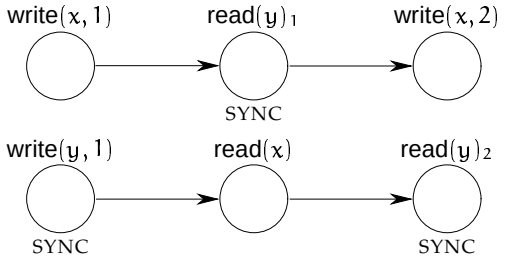
Although some systems allow clients to specify synchronization independently of the operation, many systems provide explicit mechanisms to specify which operations synchronize. These mechanisms may introduce well-formedness conditions. For example, some systems provide *synchronization variables*; access to those variables is always synchronized, while access to other variables is never synchronized. Other systems provide explicit instructions to synchronize, and these instructions often do not affect the memory directly. We discuss a few systems of this type in Section 5.5.

Example 3.9 A computation on $\mathcal{M}$ with {SYNC} for a system with a special instruction to synchronize. This instruction does not affect the data object, so we use noop for the data type operator. Recall that when no annotation is given, it is assumed to be NIL, which is implicitly included in every annotation set.



■

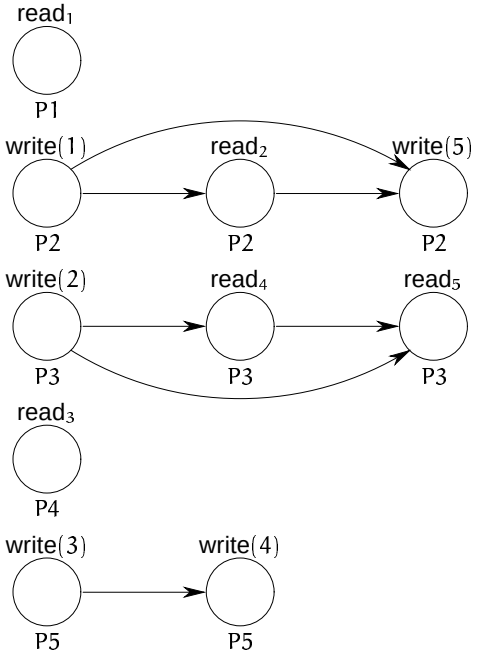
Example 3.10 A computation on $\mathcal{M}$ with {SYNC} for a system with synchronization variables. In this example, y is a synchronization variable and x is not. Recall that the subscripts differentiate multiple invocations of a data type operator.



■

Processor-centric systems. Many systems distinguish instructions by the the process or processor that issue them. Typically, instructions issued by the same processor are totally ordered, and instructions issued by different processors are not ordered. We call such systems, and the models that describe them, *processor-centric*. In our parlance, "issuing an instruction" is requesting an operation, which we model by annotating each operation with the processor that invoked it, as illustrated in Example 3.1. The annotation set is the set of processors. The requirement that the operations of each processor are totally ordered, and that operations of different processors are not ordered, is a well-formedness condition.

Example 3.11 A computation on $\mathcal{R}$ for a processor-centric system that requires operations to be ordered if and only if they are requested by the same processor:

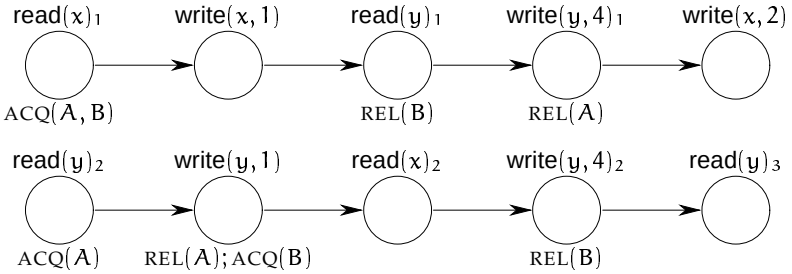


■

Many processor-centric memories allow some instructions to complete out of order unless they are separated by a special *fence* instruction. We augment the annotation set to specify these fences. The exact semantics of fences is specified by the memory model. We discuss processor-centric systems in Chapter 6.

Locks. Many systems provide *locks*, which may be *acquired* and *released*. For a system with a single system-wide lock, we can use $\{\text{ACQ}, \text{REL}\}$ for the annotation set, as in Example 3.2. For systems with multiple locks, we indicate for each lock, whether it is being acquired or released.

Example 3.12 A computation for a system with two locks A and B. If a lock is not mentioned in the annotation, it is neither acquired nor released.

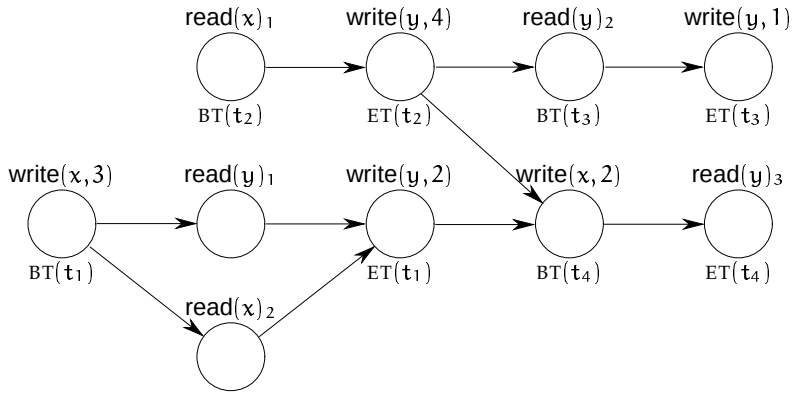


■

Locks may be implemented using synchronization, but have a more restricted use than general synchronization operations. For example, a lock may not be released before it is acquired. This added structure often makes it easier to reason about computations with locks than those with more general synchronization. We discuss locks in Chapter 7.

Transactional systems. A system may allow some operations to be grouped together into a *transaction*, which should be applied as a single larger operation. A transaction may be specified by marking the first operation as the beginning of the transaction and the last operation as its end; operations between the beginning and end are part of the transaction. Typically, the transactions are also given unique identifiers. We use the annotation set $\{\text{BT}(t) : t \in TI\} \cup \{\text{ET}(t) : t \in TI\}$, where TI is the set of transaction identifiers. The beginning of a transaction with identifier t is annotated $\text{BT}(t)$, and the end is annotated $\text{ET}(t)$. We discuss how to specify and reason about transactional systems in Chapter 8.

Example 3.13 A computation for a system with transactions.



■

Combinations. A system may understand more than one kind of constraint. We can model multiple kinds of constraints by combining the annotations used to specify each kind. However, in this thesis, we concentrate on examining each kind of constraint independently.

3.5 Getting Computations From Programs

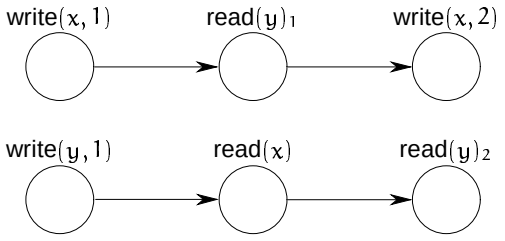
A programmer is ultimately interested in the behavior of a program running on a particular system. For this reason, it is important to understand the relationship between a program and the computations it generates. Although this relationship is difficult to characterize formally, it is a natural one in many cases. In this section, we illustrate this relationship by examining several small programs and the computations they generate. All the examples in this section assume a read/write memory data type.

Intuitively, a computation is an abstraction of a snapshot of an execution[1] of a program, as seen by the memory. That is, as a program executes, it issues instructions, some of which access the memory. At any given point in the execution, we can look at the memory accesses that have been made. Each memory access corresponds to a vertex in the computation. The edges in the computation correspond to "control" dependencies between these accesses, and the annotations usually correspond to synchronization information required by the memory. The dependencies and synchronization are usually determined by looking at the program, rather than the stream of instructions issued.

We first consider simple multiprocessor systems, in which each processor executes a sequential program. For "straight line code," where there are no unresolved control decisions, branching or looping, there is a unique computation that corresponds "naturally" to the program: the operations at each processor are totally ordered, and there are no dependencies between operations at different processors.

Example 3.14 Consider a system with two concurrent processes, two shared variables, and no mechanism for explicitly expressing synchronization. Here is a simple program and a computation that corresponds to it.

```
int x, y = 0

P1: write(x,1)
    read(y)
    write(x,2)

P2: write(y,1)
    read(x)
    read(y)
```

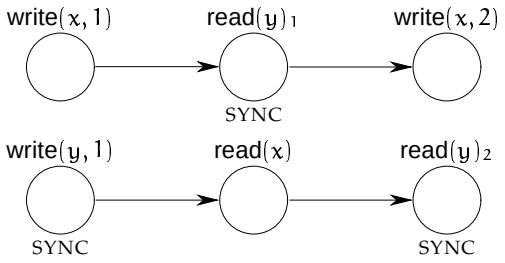


■

Example 3.15 A system may distinguish some variables as being *synchronization variables*. Operations on these variables are marked as synchronization operations. We only show the annotations for synchronization operations. Operations without annotations are implicitly annotated by NIL.

[1] We use this term informally here. It has a different, but related, formal meaning, when used in relation to automata, which we discuss in Chapter 9.

```
int x = 0
sync int y = 0

P1: write(x,1)
    read(y)
    write(x,2)

P2: write(y,1)
    read(x)
    read(y)
```

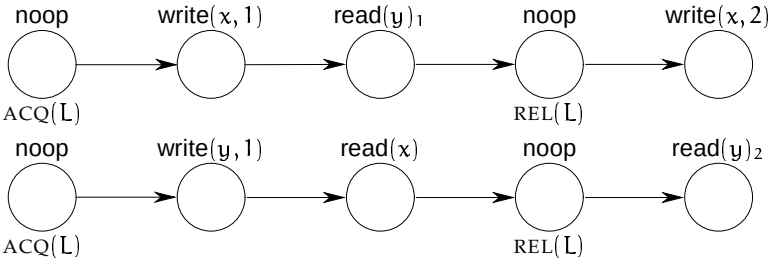


The memory may use the SYNC annotations to constrain how the operations may be applied. Such constraints are expressed by the memory model; the computation merely indicates which operations are synchronization operations. ■

Some systems provide explicit synchronization instructions that have no direct effect on the memory. To model these instructions, we assume the data type has a "no-op" operator, which always returns an acknowledgment and leaves the state unchanged (see page 25).

Example 3.16 Systems that provide locks can be modeled using explicit operations to *acquire* and *release* the locks.

```
lock L
int x,y = 0

P1: acquire(L)
    write(x,1)
    read(y)
    release(L)
    write(x,2)

P2: acquire(L)
    write(y,1)
    read(x)
    release(L)
    read(y)
```

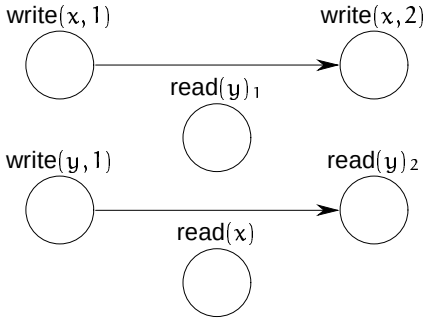


■

However, the "natural" computation for a straight line program is not always the appropriate one. For example, to improve performance, some systems explicitly allow processors to hide memory latency by reordering operations, prefetching memory reads and buffering memory writes. Such systems often provide *fence* instructions that allow the programmer to specify that the order of certain operations must be preserved. The relaxation in ordering requirements for the operations may be viewed as a weakening of the consistency guarantees of the system, but they may also be modeled as relaxations in the restrictions that the programmer expresses to the system. We discuss operation reordering in more detail in Section 6.6. Other systems allow their processors to maintain inconsistent caches. These systems cannot guarantee that the order of operations done on different locations is maintained, because a processor may have an old version of one of the locations.

Example 3.17 The computation of the program from Example 3.14 running on a system that preserves the order only between operations on the same location.

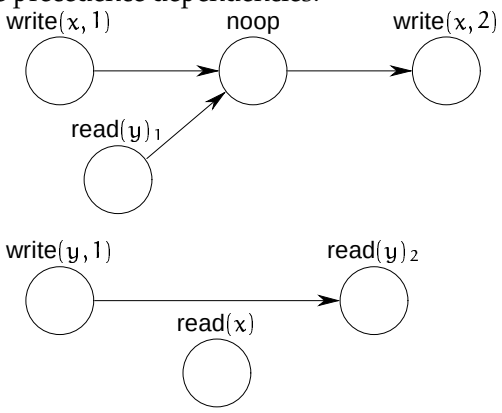
```
int x, y = 0

P1: write(x,1)
    read(y)
    write(x,2)

P2: write(y,1)
    read(x)
    read(y)
```



■

Example 3.18 There are many variations of the fence instruction. In this example, the system guarantees the order of all operations relative to the fence. No annotation is needed in the computation; the effect of the fence is expressed by the precedence dependencies.

```
int x,y = 0

P1: write(x,1)
    read(y)
    fence
    write(x,2)

P2: write(y,1)
    read(x)
    read(y)
```



■

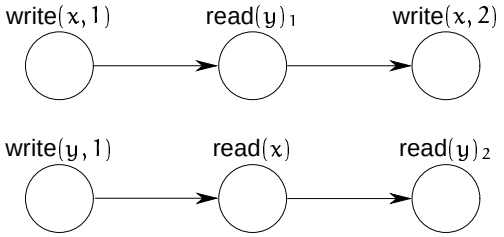
Because of nondeterministic choices that affect the way it executes, a program may not always generate the same computation, even if the system does not allow operations to be reordered. This nondeterminism may arise because of different inputs, or because different values were returned for operations invoked earlier, usually because of a race between two concurrent operations.[2]

Example 3.19 The computation generated by the following program depends on the value returned for the read(x) operation of P1.

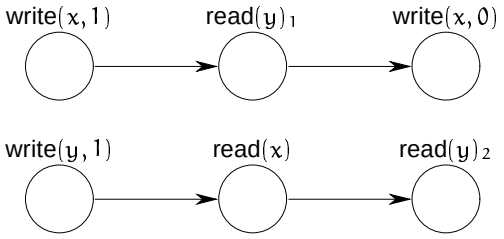
[2]It is also possible for the program to have an explicit nondeterministic choice. As far as we know, only logical programming languages and specification languages have such a construct.

```
int x,y = 0

P1: write(x,1)
    if read(y) = 0
      write(x,2)
    else
      write(x,0)

P2: write(y,1)
    read(x)
    read(y)
```

If the value returned for the read(y) of P1 is 0, then the computation generated is just as in Example 3.14:



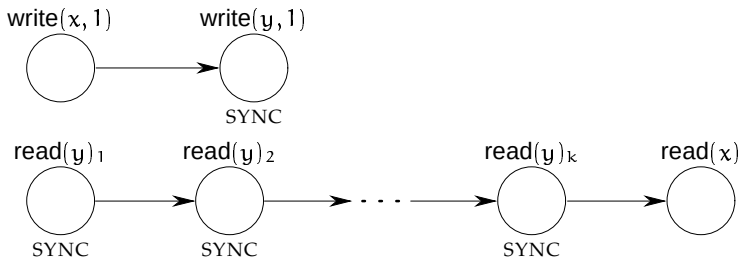
If the value returned for the read(y) of P1 is not 0, because the write(y,1) of P2 is applied before it, then the following computation is generated:



■

Example 3.20 This simple *producer-consumer* program uses a synchronized variable y, which P2 reads repeatedly until it equals 1. This one statement may generate several vertices in the computation. Also, although there is a data dependency from the write(y,1) of P1 and the final read(y) of P2, there is no edge in the computation, because data dependencies are not modeled in computations.

```
int x = 0
sync int y = 0

P1: write(x,1)
    write(y,1)

P2: repeat
      t1 <- read(y)
    until t1 = 1
    read(x)
```



■

A programming language may not have the expressive power to specify the permissible concurrency of some computations. Thus, a program may require one operation to precede another even though no logical dependence exists. In this case, we may not want the computation to include a precedence dependency between the two operations. This problem is common when programming in a sequential language for a multiprocessor. Determining whether two operations are logically or merely incidentally ordered is a major challenge in compiling sequential programs for multiprocessors, and we do not tackle it in this thesis.

Example 3.21 Consider the following sequential program, with local variables i, j, k, m, t0, t1 and t2, and a single shared array A, initialized to some random values.

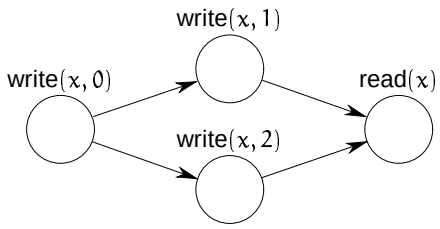
```
int A[*] = <random initial values>

count(int i,j,k):
  % count occurences of k in A[i..j]
  int m

  if j < i
    return 0
  m <- (i+j)/2   % integer division
  t0 <- read(A[m])
  t1 <- count(i,m-1,k)
  t2 <- count(m+1,j,k)
  if t0 = k
    return t1+t2+1
  else
    return t1+t2
```

Although this program specifies that read$(A[m])$ precedes the recursive call on the first half of the array, which precedes the recursive call on the second half of the array, these precedences are *not* logical dependencies. Thus, we might say that the computation corresponding to this program would not have edges between these operations. ■

Rather than require a compiler to determine the logical dependencies in a program, or relying on the processor to reorder operations to improve performance, some programming systems provide explicit control structures to indicate the logical dependence between operations.

Example 3.22 In this example, `spawn` indicates that the function called can be done concurrently with the operations that follow the `spawn`, until the next `sync` operation.

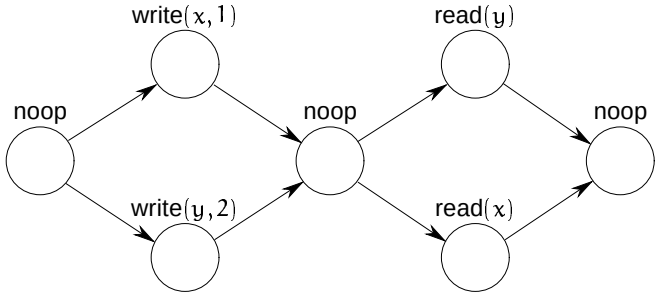
```
f: x <- 1

g: x <- 2

main: x <- 0
      spawn f
      spawn g
      sync
      read x
```



■

Example 3.23 This example uses the `cobegin` and `coend` construct to indicate which operations may be executed concurrently.

```
cobegin
  write(x,1)
  write(y,2)
coend;
cobegin
  read(y)
  read(x)
coend
```



■

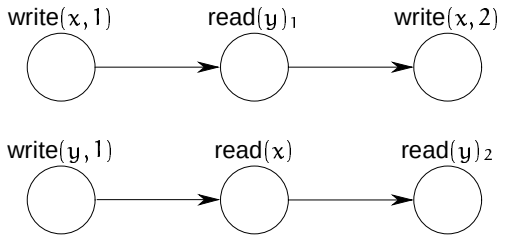
Again, no general method for determining the computation generated by a program is known. Finding one is a difficult task that deserves further investigation, which we do not undertake in this thesis.

3.6 Schedules

We want to know the result of executing a given computation—the values returned for its operations and the final state of the memory after the computation executes. This result depends on the memory model, which we discuss in later chapters. There is one possibility, however, that is easy to consider: when the operations are applied serially, that is, as if the computation were executed on a sequential machine. In this case, we only need to know the order in which the operations are applied. This order of operations is called a *schedule*. Operations should be scheduled after any operations that precede them in the computation; that is, the schedule should *respect* the precedence dependencies. The annotations may specify other constraints that schedules should respect.

Formally, a ***schedule*** α of C is any topological sort of C, that is, a serialization of the operations consistent with the precedence dependencies. We maintain the annotations with the schedule, writing $ann_\alpha(x) = ann_C(x)$ for the annotation of x. Because an annotation of an operation in a schedule is the same as its annotation in the computation, we often omit the subscript. We denote the set of schedules of C by $Sch(C)$.

Example 3.24 Here are some schedules of the following computation from Example 3.14:



$$\mathsf{write}(x,1), \mathsf{read}(y)_1, \mathsf{write}(x,2), \mathsf{write}(y,1), \mathsf{read}(x), \mathsf{read}(y)_2$$
$$\mathsf{write}(y,1), \mathsf{read}(x), \mathsf{read}(y)_2, \mathsf{write}(x,1), \mathsf{read}(y)_1, \mathsf{write}(x,2)$$
$$\mathsf{write}(x,1), \mathsf{write}(y,1), \mathsf{read}(y)_1, \mathsf{write}(x,2), \mathsf{read}(x), \mathsf{read}(y)_2$$ ■

Example 3.25 The following are *not* schedules of the computation from Example 3.24:

$$\mathsf{write}(y,1), \mathsf{read}(x), \mathsf{read}(y)_1, \mathsf{write}(x,1), \mathsf{read}(y)_2, \mathsf{write}(x,2)$$
$$\mathsf{write}(x,1), \mathsf{write}(y,1), \mathsf{read}(y)_1, \mathsf{write}(x,2), \mathsf{read}(y)_2, \mathsf{read}(x)$$
$$\mathsf{write}(x,1), \mathsf{read}(y)_1, \mathsf{write}(x,2), \mathsf{write}(x,3), \mathsf{write}(y,1), \mathsf{read}(x), \mathsf{read}(y)_2$$
$$\mathsf{write}(x,1), \mathsf{write}(y,1), \mathsf{read}(y)_1, \mathsf{write}(x,2), \mathsf{read}(x)$$
$$\mathsf{write}(x,1), \mathsf{write}(y,1), \mathsf{read}(y)_1, \mathsf{write}(x,2), \mathsf{read}(x), \mathsf{read}(y)_2, \mathsf{write}(x,2)$$

The first two sequences are not consistent with the precedence order. The third has an additional $\mathsf{write}(x,3)$ operation, while the fourth is missing the $\mathsf{read}(y)_2$ operation. The final sequence is not even a serialization, because the $\mathsf{write}(x,2)$ operation appears twice. ■

Because computations are dags, every computation has at least one schedule.

Lemma 3.1 Every computation has at least one schedule.

Proof: This lemma follows immediately from the basic graph theory result that every dag has a topological sort. ■

Given a schedule, the behavior of the memory is determined by its data type, which specifies its serial semantics. Using the theory developed in Chapter 2, we can reason about and determine the return values for the operations and the final state of the memory if the computation is executed according to that schedule.

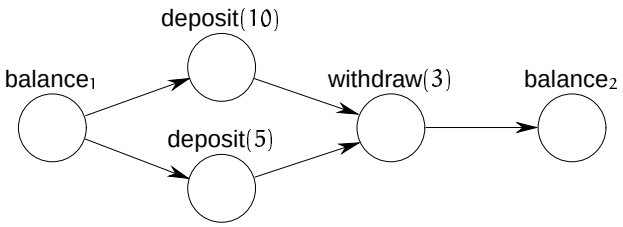
The annotations may specify *scheduling constraints*, which restrict the schedules that the system allows. We say that the schedules must *respect* the scheduling constraints. Like the well-formedness conditions discussed in Section 3.2, these restrictions vary with the system. For example, in a system with locks, a schedule should not release a lock before it is acquired, nor should it acquire a lock that was previously acquired but not yet released. We formally define scheduling constraints and what it means to respect them in the context of the particular memory models that we develop in later chapters.

3.7 Races and Determinacy

When the memory is implemented by a distributed system, the order in which operations are scheduled depends on unpredictable factors, such as communication latency, network congestion, and processor speed and load. Two operations comprise a *race*, and are said to *compete*, if they can occur in different orders, and the order in which they occur affects the result of the computation. Because the values returned for competing operations depend on these unpredictable factors, races make it harder to reason about the possible behaviors of the program running on the system. Thus, it is considered good practice to eliminate races from programs to the extent possible.

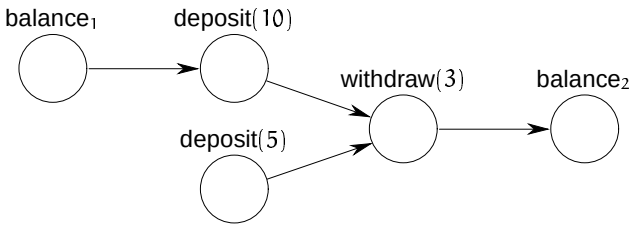
Formally, two operations ***compete*** in a computation if they conflict and are concurrent. A pair of competing operations is a ***race***, and a computation is ***(completely) race-free*** if it has no races, that is, if no operations compete.

Example 3.26 A completely race-free computation on $\mathcal{B}$:



■

Example 3.27 A computation on $\mathcal{B}$ that is *not* race-free:



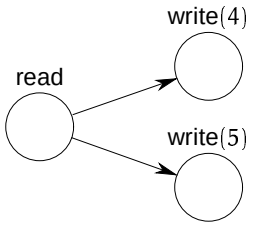
The balance$_1$ operation competes with the deposit(5) operation. ■

Race-free computations are easy to reason about because the values returned for operations are determined by the computation; they do not depend on the schedule. We say that computations with this property—that the return values are independent of the schedule—are *observationally determinate*.

As we show below, race-freedom also guarantees that the final state of the memory after executing the computation is independent of the schedule. This property, which we call *internal determinacy*, is useful when the computation being considered is a part of a larger computation. Computations that are both internally and observationally determinate are *strongly determinate*.

Formally, a computation is ***(observationally) determinate*** if all its schedules are observationally equivalent; it is ***internally determinate*** if all its schedules are internally equivalent; and it is ***strongly determinate*** if all its schedules are strongly equivalent.

Example 3.28 A determinate computation on $\mathcal{R}$ that is not strongly determinate.



■

Example 3.29 The computation from Example 3.26 is strongly determinate. ■

Example 3.30 The computation from Example 3.27 is internally, but not strongly, determinate. ■

A determinate computation is easy for a memory to execute, because it does not need to check the consistency of the values returned for different operations. The values may be computed using different schedules because every schedule yields the same value for every operation.

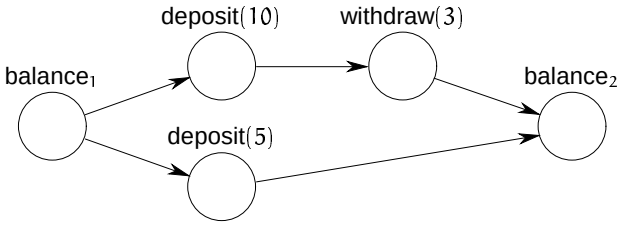
In general, it is difficult to prove that a computation is determinate because we need to check that all its schedules are observationally equivalent. However, as noted above, race-freedom is a sufficient condition for strong determinacy, which implies determinacy.

Lemma 3.2 If C is a completely race-free computation then C is strongly determinate.

Proof: Immediate from Theorem 2.16, with $\mathsf{X} = V_{\mathrm{C}}$ and the partial order $\preceq_{\mathrm{C}}$. ■

The converse of this theorem is not true in general; a computation with races may be determinate.

Example 3.31 The following computation is strongly determinate, but not completely race-free. In particular, the deposit(5) and the withdraw(3) operations compete.



■

Race-freedom is easier to check directly than determinacy, because we only need to examine the pairs of concurrent operations in the computation, rather than every schedule. Much research has been done to efficiently detect races in programs [6, 85, 89, 90, 37, 96, 24, 25].

Most systems have special *synchronization* operations that "protect" against races, so that competing operations that are separated by appropriate synchronization are not considered races. "Race-free" programs for these systems may not generate determinate computations. The system typically guarantees greater consistency for synchronization operations, so that the degree of nondeterminacy is reduced. The exact definition of what constitutes a race depends on the particular system.

3.8 Discussion

In this section, we discuss some of the reasons we use computations as the basis for the framework developed in this thesis. We also discuss the drawbacks of this approach.

We note again that computations are *not* programs. Ultimately, we want to reason about the behavior of programs. However, there are at least three reasons why programs are not suitable as the interface between the clients and the memory. First, a program is written in a particular programming language; we want this framework to be independent of the choice of programming language. Second, compilation creates a large gap between the program and what actually executes on the system; we wish to avoid many of the issues that arise because of compilation. Third, programs have control structures—such as loops and branches—which allow some statements to be executed multiple times while others are never executed. To reason about how a program executes, we must distinguish between the program's instructions and the *instantiations* of these instructions in an execution of the program, which the memory sees. Every instantiation corresponds to a different operation in the computation.

One significant point of confusion in much of the previous work on memory models is a result of failing to treat the last point—that instructions are different from their instantiations in a particular execution—with sufficient care. Following Lamport's definition of *sequential consistency* [68], many memory models used the notion of a *program order*, which

is "the order specified by the program" [1], or "the partial order on all memory operations that is consistent with the per-processor total order" [41, 97]. This order depends not only on the program but also on the particular execution of the program. The problem arises when this order is used to define a memory model, as the execution that results depends on the memory model, making the definition circular [100]. We sidestep this problem by having explicit precedence dependencies; that is, the program order is the precedence order of the computation.

A computation expresses the logical structure of a program's execution. In this thesis, we develop a *computation-centric framework* for modeling memory consistency guarantees, in which clients are characterized by the computations they generate. The values that may be returned by the memory are determined based only on the computation specified by the clients. This framework provides a foundation for reasoning rigorously about the semantics of concurrent access to a shared memory.

We believe the computation-centric framework is an adequate basis upon which to build a full theory of concurrent programming with shared memory. Although the values returned by a shared memory system may depend on factors not expressed in the computation—especially factors such as processor load and communication delays that affect the timing of operations—these factors are not available to the programmer, who thus cannot use them in reasoning about the behavior of a program.

To bridge the gap between computations and programs, there are three sets of issues that must be addressed: Language design, compilation, and the generation of computations from programs.

Most issues in the design of a programming language are orthogonal to the memory consistency guarantees. However, we want to ensure that the way we model consistency does not restrict the language design possibilities; it should not exclude desirable constructs for expressing and structuring concurrency. Maintaining this flexibility is important because there is no consensus yet on the "right" constructs for concurrent programming. Computations can express any logical constraint on how the memory may be accessed.

Compilation is also an orthogonal issue. Because computations can model a high level program executing on an abstract machine, the effects of compilation and the guarantees of the underlying system can—and usually should—be hidden from the programmer.

We discussed the generation of computations from programs in Section 3.5. The most difficult issue arises because the computation generated by a program may depend on the values returned by the memory for earlier operations, as we saw in Examples 3.19 and 3.20. Thus, what appears as a data dependency, which is not expressed in the computation, affects the control flow of the program, which is expressed in the computation. We discuss this problem, and how we can deal with it, in Chapter 9.

Instead of using computations, we could model a program as a state machine, which models the dynamic behavior of the system. Although this approach is general—it can capture the dynamic dependence on return values, for example—without additional structure, it is difficult to characterize programs by looking at their associated state machines. For example, it is hard to tell whether a state machine is race-free, a property that is critical to many of the results in this thesis. Nonetheless, in Chapter 9, we develop a simple theory using state machines that builds on the computation-centric framework.

Unlike much of the previous work [47, 8, 2, 100, and others], computations are not

based on processors or processes. Processor-centric memory models, which assume that the clients specify a total per-processor order on the operations with no cross-processor dependencies, restrict the forms of concurrency that can be expressed. These models are natural for low level descriptions of shared memory multiprocessors, but they cannot model higher level programming systems, such as Cilk [19], that do not have an explicit notion of a process. We discuss these models in Chapter 6.

In the literature, memory models are often formalized by identifying *legal histories*[3] of a system [68, 33, 53, 4, 43, 11, 8, 54]. In addition to synchronization information and the per-processor order, this approach uses the return values or a total order on system events (such as requests and responses from the memory) to derive a partial order analogous to the logical precedence order of a computation. Because the return values and the timing of some system events depend on the system rather than the clients, there is no independent description of the clients in this approach. In contrast, computations provide a clean split between the clients and the memory because they contain only the information that the clients specify to the memory, and neither the return values nor timing information.

Because annotations are defined for each operator, and operators are unique, we could incorporate them into the data type operator. However, the data type operator specifies how the data object changes when the operator is applied, whereas the annotation does not affect the data object at all. In particular, given a serialization of the operators to be applied, the result of applying that serialization does not depend on the annotations; it is completely specified by the operators. Thus, we separate the annotations from the data type. Annotations are used to constrain how the operators may be applied. That is, the annotations, and the precedence dependencies, restrict the possible serializations that may be used to determine the return value for each operator.

Because the well-formedness conditions and scheduling constraints depend on the annotations, which vary from system to system, it may be better to parameterize computations by a data type and an *annotation type*. In addition to the annotation set, an annotation type would specify the well-formedness conditions and the scheduling constraints implied by the annotations. Annotation types may allow a more general and uniform treatment of annotations. Also, well-formedness would be treated differently from other client restrictions, which correspond to programming disciplines rather limitations inherent to the system.

[3] We use this term broadly; there are many variants of this approach. Histories are often called *executions*, and correspond to the automaton executions we formally define in Chapter 9.

Chapter 4

The Computation-Centric Framework

A memory model specifies what values the memory may return for the operations that the clients request. When the interface between the clients and the memory consists of computations and return value functions, it is natural to describe a memory model as an association of computations and return value functions. We call such memory models *computation-centric*. In this chapter, we define computation-centric models and show how to reason about them.

In any execution of a memory system, the computation describes the clients' requests and the return value function describes the memory's response. Together, a computation and return value function comprise an *observation* of the system. A computation-centric memory model specifies the possible observations of a memory system.

A memory model may be either a description of the guarantees made by a memory or a specification of the guarantees assumed by the clients. A system *implements* a specification if the observations of the system are possible observations of the specification. Sometimes a memory system implements its specification only if the clients access the memory in a restricted fashion, for example, by avoiding races. With restricted clients, it is possible for a weakly consistent memory to implement a memory with stronger consistency guarantees. The main results in this thesis are of this form.

Some systems preprocess their computations and then execute them on an underlying memory. We introduce *computation transformations* to model this preprocessing. The resulting system may provide stronger consistency guarantees than the underlying memory by adding constraints on the way operations may be scheduled. For example, the system may force operations to be applied in a particular order by adding precedence dependencies to the computation. Conversely, the memory may weaken the consistency guarantees of the underlying memory by eliminating some of the dependencies specified by the clients, so that operations may be reordered. We use computation transformations to define several memory models in this thesis, particularly in Chapters 6 and 7.

Outline: In Section 4.1, we formally define observations and computation-centric memory models. Section 4.2 defines some properties computation-centric models may have.

The material in this chapter is based in part on earlier work done with Matteo Frigo and presented at the Symposium for Parallel Algorithms and Architectures in 1998 [40].

In Section 4.3, we define an implementation relation between computation-centric models. We show how to model restricted clients in Section 4.4, and we define computation transformations in Section 4.5. Finally, in Section 4.6, we discuss some of the strengths and weaknesses of computation-centric models.

Reading Guide: This chapter develops a computation-centric framework for defining and reasoning about memory consistency. It is dense with formal definitions and has fewer examples than previous chapters. The key definitions are those of observations, computation-centric memory models, client restrictions and computation transformations; many examples of these concepts appear in later chapters. It is possible to first read Section 4.1, skim Sections 4.3 and 4.4, and then skip to Chapter 5, referring back to this chapter as necessary. Computation transformations are not used until Section 5.5.

4.1 Computation-Centric Memory Models

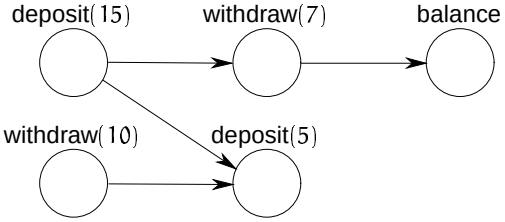
We characterize a memory system by the behavior that may be observed by the clients when a program is executed on the system. Given a computation, which specifies the operations requested by the clients and the logical constraints on how these operations may be applied, a *computation-centric memory model* specifies the values that may be returned for the operations. In this section, we formally define computation-centric memory models and discuss how they should be interpreted.

A computation-centric memory model is a postmortem characterization of a memory system. That is, in the computation-centric approach to modeling memory consistency, we look at an execution of a program after it has terminated rather than reason about how it unfolded over time. We can derive the computation from the execution, as described in Section 3.5. Because the execution has terminated, every operation has a return value. Together the computation and the return values comprise an *observation* of the system.

Because the system may not be a sequential machine, the return values of an observation need not be explained by any schedule of the computation. However, every memory should respect the precedence dependencies, so the return value for each operation should reflect the effects of the operations that precede it in the computation.

Formally, given a memory with data type $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$ and annotation set A, an ***observer function*** for a computation $C \in \mathfrak{C}_A^{\mathcal{D}}$ is a total return value function $\rho\colon V_C \to R$ on the operations of C such that for each operation $x \in V_C$, there is a schedule α of C that explains $\rho|_{\{x\}}$, that is, $\rho(x) = \mathit{retval}(x, \alpha)$. There may be a different schedule for each operation. An ***observation*** of the memory is a pair (C, ρ), where C is a computation and ρ is an observer function for C. We use $\mathfrak{O}_A^{\mathcal{D}}$ to denote the set of possible observations of any memory with $\mathcal{D}$ and A.

Example 4.1 Consider the following computation and return value function (combining the two partial return value functions from Example 2.18):



$$f(\text{deposit}(15)) = \text{ACK}$$
$$f(\text{withdraw}(7)) = \text{ERR}$$
$$f(\text{balance}) = 8$$
$$f(\text{withdraw}(10)) = 10$$
$$f(\text{deposit}(5)) = \text{ACK}$$

Although no schedule of the computation explains f, it is an observer function for the computation because each return value is explained by one of the schedules from Example 2.18. ■

Example 4.2 The following return value function is *not* an observer function for the computation in the previous example because there is no schedule of that computation that returns 12 for the balance operation.

$$g(\text{deposit}(15)) = \text{ACK}$$
$$g(\text{withdraw}(7)) = \text{ERR}$$
$$g(\text{balance}) = 12$$
$$g(\text{withdraw}(10)) = 10$$
$$g(\text{deposit}(5)) = \text{ACK}$$

■

An observation includes only the control structure of the clients' requests, specified by the computation, and the values returned by the memory. It does not specify data dependencies, as they are not observed directly by the clients; they must be inferred from the computation and return values. Neither does an observation indicate the order in which the values are returned to the clients. Clients that rely on one operation being applied before other must explicitly specify this requirement by a precedence dependency unless it is implied by the value returned for the earlier operation.

Restricting observations to observer functions, rather than allowing arbitrary return value functions, ensures that the value returned for each operation is "reasonable" and not made up arbitrarily by the memory. That is, each return value can be explained by some schedule. However, because each operation may be explained by a different schedule, the return values for different operations in an observation need not be consistent; there might not be any schedule that simultaneously explains the return values for two different operations. Consistency guarantees must be specified by the memory model.

A ***computation-centric memory model*** for a memory system is a set of observations of the system, including the empty computation with the null function as its observer function. If M is the model for a memory system, then $(C, \rho) \in M$ is a possible observation of that system. That is, if the clients specify the computation C, the memory may return $\rho(x)$ for each operation $x \in V_C$. The empty computation with the null function is included in every computation-centric model because it is the observation of any memory in which the clients do not request any operations.

Given a memory system, we can derive its computation-centric memory model as follows: For each possible execution of the system, determine the computation and the return values, which together comprise an observation of the system. The set of all these observations is the memory model. Because we do not have a formal model of the system, other than the computation-centric model, we cannot describe this process formally yet. In Chapter 9, we develop state machine models for memory systems, and in Section 9.6,

we show how to formally derive computation-centric models from the state machine descriptions.

The set of all computation-centric models for a memory with $\mathcal{D}$ and A is $\mathfrak{M}_A^{\mathcal{D}} = 2^{\mathfrak{O}_A^{\mathcal{D}}}$, the power set of $\mathfrak{O}_A^{\mathcal{D}}$; that is, $M \in \mathfrak{M}_A^{\mathcal{D}}$ if and only if $M \subseteq \mathfrak{O}_A^{\mathcal{D}}$. When $(C, \rho) \in M$, we say that M ***admits*** ρ for C, and that (C, ρ) is ***admissible***[1] according to M. Because clients specify the computation, a computation-centric memory model is often given by a rule relating computations and observer functions. We use $M[C] = \{\rho : (C, \rho) \in M\}$ to denote the set of observer functions that M admits for C.

Example 4.3 (Generic Memory) The *generic memory* model $GM = \mathfrak{O}_A^{\mathcal{D}}$ is the most general possible computation-centric memory model. It can also be written as follows:

$$\begin{aligned} GM &= \{(C, \rho) : \rho \text{ is an observer function for } C\} \\ &= \left\{(C, \rho) : \text{for each } x \in V_C, \text{ there is a schedule of } C \text{ that explains } \rho|_{\{x\}}\right\} \\ &= \{(C, \rho) : \forall x \in V_C, \exists \alpha \in Sch(C), \rho(x) = retval(x, \alpha)\} \end{aligned}$$

■

Example 4.4 (Sequential Consistency) A memory that preserves the appearance of a sequential machine is called *sequentially consistent*. Its computation-centric model is:

$$\begin{aligned} SC &= \{(C, \rho) : \text{there is a schedule } \alpha \text{ of } C \text{ such that for all } x \in V_C, \rho(x) = retval(x, \alpha)\} \\ &= \{(C, \rho) : \text{there is a schedule of } C \text{ that explains } \rho\} \end{aligned}$$

Sequential consistency is an important model, and we consider it in greater depth in Chapter 5. ■

Because observations record only the return values for the operations of a computation, computation-centric models are defined by these values, not how they were determined. Although we often define computation-centric models by constraining the schedules that explain the return values, the schedules actually used by a system described by a model need not satisfy the constraints specified by the model. For example, sequential consistency requires that a single schedule explain all the return values. However, a distributed system implementing sequential consistency may use different schedules to determine the return values of different operations—these schedules may not even be equivalent—as long as there is some schedule that explains all the return values.

A computation-centric model need not admit an observer function for every computation. In particular, a memory model typically does not admit any observer function for computations that are not well-formed; there is no appropriate observation of the system because the computation may not be specified.

On the other hand, computation-centric models do not imply any liveness properties. A system may deadlock or fail in some other way to completely execute a computation, even if the memory model admits an observer function for that computation. An observation is admissible if it may be observed in *any* completed execution of the system. Although liveness is an important issue—our model allows trivial implementations that never complete an execution—it is largely orthogonal to the consistency guarantees, which are safety properties. We do not study liveness in this thesis.

[1] This terminology is natural when we consider a memory model as a specification, as we discuss in Section 4.3, rather than a description of a memory system.

If every observation of one memory is a possible observation of another memory, then the first memory is at least as restrictive as the second; it will only return values that the second memory may also return.

Formally, a memory model M is ***stronger*** than another model M' if $M \subseteq M'$. We also say that M' is ***weaker*** than M. The "stronger" and "weaker" relations are partial orders. Note that the subset, not the superset, is said to be stronger, because the subset admits fewer observations of the memory.

Example 4.5 *SC* is stronger than *GM*. In fact, any computation-centric memory model is stronger than *GM*. ■

To compare our framework with memory models proposed in the literature, it is sometimes convenient to relax the requirement that the return value function of an observation be an observer function. A ***generalized computation-centric memory model*** is one in which an "observation" may pair a computation with an arbitrary (total) return value function for its operations; that is, it is a subset of $\{(C, \Phi) : \Phi \text{ is a return value function for } V_C\}$.

We may also observe the system in an intermediate state while a program is still executing. In this case, the computation consists of the operations requested so far. Because the system may not yet have returned a value for every operation requested, we allow the observer function to be partial. Formally, a ***partial observer function*** for a computation is a partial return value function on the operations in which each operation with a return value—the operations in its domain—is explained by some schedule of the computation. An ***intermediate observation*** is a computation paired with a partial observer function. Intermediate observations are useful for reasoning about the dynamic interaction between the clients and memory, which we discuss in Chapter 9.

4.2 Properties of Computation-Centric Models

Although the use of observer functions ensures that any observation is reasonable, the definition of computation-centric models allows many models that do not describe any real system. In this section, we define several properties that most real systems satisfy.

If two computations are identical except that one has fewer precedence dependencies than the other, then we expect that an observer function admissible for the computation with more dependencies is also admissible for the one with fewer dependencies. This property is called *monotonicity*. Formally, C is ***stricter*** than C' if $V_C = V_{C'}$, $ann_C = ann_{C'}$, and $E_{C'} \subseteq E_C$. A memory model $M \in \mathfrak{M}_A^{\mathcal{D}}$ is ***monotonic*** if whenever C is stricter than C', we have $M[C] \subseteq M[C']$.

Because the clients specify the computation and the memory model characterizes the memory, we expect that a memory model admits at least one observer function for each computation. This property is called *completeness*. Formally, a memory model $M \in \mathfrak{M}_A^{\mathcal{D}}$ is ***complete*** if for all $C \in \mathfrak{C}_A^{\mathcal{D}}$, there exists an observer function $\Phi\colon V_C \to R$ for C such that $(C, \Phi) \in M$, or equivalently, $M[C] \neq \emptyset$. This property implies that it is possible for the memory system to respond to to any computation requested by the clients.

Although an observation considers the entire computation and all the return values statically, a real memory system receives and responds to some operation requests before

others. Once it returns a value for an operation, it cannot take it back. Most memories guarantee not only completeness, which says that they can respond if the entire computation is known ahead of time, but also that they can respond as new operations are requested. A system could "get stuck" if the observer function for some initial computation cannot be extended as the computation is extended. A memory that cannot get stuck in this fashion is said to be *constructible*. Formally, a model M is ***constructible*** if whenever C' extends C and $\rho \in M[C]$, there is an extension ρ' of ρ such that $\rho' \in M[C']$.

Example 4.6 *GM* and *SC* from Examples 4.3 and 4.4 are monotonic, complete and constructible. ■

It is easy to see that constructibility implies completeness.

Lemma 4.1 Any constructible memory model is complete.

Proof: Immediate from the definition of constructibility, since every computation extends the empty computation, and every memory admits the null function for the empty computation. ■

As mentioned in Chapter 3, some systems may have well-formedness conditions that restrict the computations they can handle. For such systems, it is appropriate to consider only well-formed computations in the definition of the properties above. Formally, if *WF* is the set of well-formed computations for a memory, a model M is ***monotonic under*** *WF* if for all $C, C' \in WF$ such that C is stricter than C', we have $M[C] \subseteq M[C']$. Similarly, M is ***complete under*** *WF* if $M[C] \neq \emptyset$ for all $C \in WF$, and it is ***constructible under*** *WF* if for all $C, C' \in WF$ such that C' extends C, every $\rho \in M[C]$ has an extension $\rho' \in M[C']$.

We do not develop these concepts in greater depth, as it is not necessary for this thesis. See [40] and [39] for a more extensive treatment, particularly of constructibility.

4.3 Implementing Memory Models

In addition to describing a memory formally, a memory model can be used to specify the requirements for a memory system. Informally, a system *implements* a specification if the clients of the system cannot tell that they are not interacting with a system described by the specification. A system that implements a memory model can be viewed by the clients as a memory that provides the guarantees specified by the model, without regard to the actual structure of the underlying system. If the clients choose the computation assuming it will be executed on a memory modeled by a memory model, then it can be "safely executed" on any system that implements the memory model.

We do not have a general method for proving that a distributed system implements a computation-centric memory model: Any such method would depend on the formalism used to describe the system. However, when the proposed implementation is a memory described by a computation-centric model, there are some general techniques to prove that the system implements the specification memory model.

One memory system implements another system with the same interface if the clients, observing the first system execute a computation, cannot tell that it is not the second; that is, if every observation admissible according to the memory model of the first system is also admissible according to the memory model of the second. Formally, $M \in \mathfrak{M}_A^{\mathcal{D}}$ ***implements*** $M' \in \mathfrak{M}_A^{\mathcal{D}}$ if $M \subseteq M'$. This definition is identical to the definition of M being stronger

than M', which makes sense because any computation can be "safely executed" on a memory that provides stronger guarantees than the one it was intended to be executed on. Two memory models are ***equivalent*** if they implement each other.[2]

Example 4.7 *SC* implements *GM*, but *GM* does not implement *SC*. ■

We want to be able to prove that under certain conditions, a memory can be implemented using a weaker memory, or a memory with a different interface. These conditions may be restrictions on the clients, extra processing by the system, or both. We consider these possiblities in the next two sections.

4.4 Client Restrictions

Often, we are concerned about the behavior of the memory only when it is accessed in a restricted way, that is, by clients obeying a discipline of memory access. Under such conditions, some memories provide additional consistency guarantees; that is, they are indistinguishable from stronger memories to clients that obey the memory access discipline. The main results of this thesis, Theorems 5.4, 5.7, 6.13, 7.25, 7.31 and 8.16, consist of showing a result of this kind for specific memory models and access disciplines. In this section, we define *client restrictions*, which model memory access disciplines, and make precise what it means in the computation-centric framework for one memory to implement another *under a client restriction*.

In the computation-centric framework, clients are characterized by the computations they may specify. Thus, we model a restriction on the way memory may be accessed as a restriction on the computations; that is, the clients guarantee that the computation specified satisfies certain properties. We are interested only in the observations for those computations. The *memory model under a client restriction* includes only the observations with computations allowed by the restriction. This terminology is the same as the terminology used in Section 4.2 for the well-formedness conditions, reflecting that we treat well-formedness conditions exactly like other client restrictions.

Formally, a ***client restriction*** is a set $CR \subseteq \mathfrak{C}_A^{\mathcal{D}}$ of computations that includes the empty computation, where $C \in CR$ means that C may be specified by the clients. If $M \in \mathfrak{M}_A^{\mathcal{D}}$ then M ***under*** CR is $M|_{CR} = \{(C, \Phi) \in M : C \in CR\}$.

Example 4.8 For a read/write memory with memory barriers that do not read or write the memory, the annotation set is $A = \{\text{SYNC}\}$, and the client restriction, a well-formedness condition, is specified by

$$\left\{ C \subseteq_A \mathfrak{C}^{\mathcal{M}} \forall x \in \quad V_C,\ ann_C(x) = \text{SYNC} \implies x = \mathsf{noop} \right\}.$$

Recall that every annotation set implicitly includes NIL, so $ann_C(x) = \text{NIL}$ if $ann_C(x) \neq \text{SYNC}$. ■

[2] Two models are equivalent exactly when they are equal. We use the term equivalent to be consistent with later usage.

Example 4.9 (Serial Clients) The clients may access the memory serially; that is, they may coordinate so that there are no concurrent accesses to memory. In this case, the operations are totally ordered by the computation corresponding to their requests. Such clients satisfy the following client restriction:

$$\mathit{Serial} = \left\{ C \in \mathfrak{C}^{\mathcal{D}} : E_C \text{ is a total order} \right\}$$ ■

Example 4.10 (Race-Free Clients) It is often considered good practice to eliminate races from programs. A program is *race-free* if all the computations it generates are race-free. In the computation-centric framework, a client is race-free if it satisfies the following client restriction:

$$\mathit{RF} = \left\{ C \in \mathfrak{C}^{\mathcal{D}}_{A} : C \text{ is completely race-free} \right\}.$$

This restriction specifies ***complete race-freedom***. Memories often provide special operations that "protect" against races. We discuss such operations in the context of particular memory models. ■

We say that M ***implements*** M' ***under*** CR if $M|_{CR}$ implements $M'|_{CR}$, that is, if $M|_{CR} \subseteq M'|_{CR}$. Two memory models are ***equivalent under*** CR if they implement each other under CR. As mentioned earlier, the main results in this thesis have the form M implements M' under CR, where M' is stronger than M.

Example 4.11 Any memory implements sequential consistency under serial clients; that is, any memory model implements *SC* under *Serial*, where *SC* and *Serial* are defined in Examples 4.4 and 4.9. This statement is true because every computation in *Serial* has a unique schedule defined by its precedence dependencies. ■

The following lemma gives a simple alternate definition for the "implements under a client restriction" relation.

Lemma 4.2 M implements M' under CR if and only if $M[C] \subseteq M'[C]$ for all $C \in CR$.

Proof: Suppose M implements M' under CR, or equivalently, $M|_{CR} \subseteq M'|_{CR}$. If $C \in CR$ and $\rho \in M[C]$ then $(C, \rho) \in M|_{CR} \subseteq M'|_{CR}$, and thus, $\rho \in M'[C]$.

Suppose $M[C] \subseteq M'[C]$ for all $C \in CR$. If $(C, \rho) \in M|_{CR}$ then $C \in CR$ and $\rho \in M[C] \subseteq M'[C]$, so $(C, \rho) \in M'|_{CR}$. ■

The implementation relation is preserved under any client restriction.

Lemma 4.3 For any client restriction CR, if M implements M' then $M|_{CR}$ implements $M'|_{CR}$.

Proof: Since $M \subseteq M'$, we have $M|_{CR} = \{(C, \rho) \in M : C \in CR\} \subseteq \{(C, \rho) \in M' : C \in CR\} = M'|_{CR}$. ■

If a memory implements a stronger memory under some client restriction, then they are equivalent under that client restriction.

Lemma 4.4 If M is weaker than M' and implements M' under CR then M and M' are equivalent under CR.

Proof: Because M is weaker, M' implements M, and by the previous lemma, M' implements M under CR. Thus, M and M' are equivalent under CR. ■

Many client restrictions are *monotonic*; that is, they simply require that the clients include "enough" precedence dependencies. Formally, CR is ***monotonic*** if for all $\mathsf{C} \in CR$ and C' stricter than C, we have $\mathsf{C}' \in CR$.

Example 4.12 Both *Serial* and *RF* from Examples 4.9 and 4.10 are monotonic. ■

We can also compare two client restrictions. A client restriction CR is ***more restrictive*** than CR' if $CR \subseteq CR'$. A memory interacting with more restrictive clients provides stronger guarantees.

Lemma 4.5 If CR is more restrictive than CR' then $M|_{CR}$ implements $M|_{CR'}$.

Proof: Immediate from the definitions. ■

4.5 Computation Transformations

We can implement a memory model over a memory system with different guarantees by preprocessing the requests from the clients. In the computation-centric framework, preprocessing corresponds to changing the computation. For example, to get stronger consistency guarantees, the system may delay the application of some operations by adding precedence dependencies. Other systems allow operations to be reordered, by removing precedence dependencies, or change the annotations. In this section, we define *computation transformations*, which model the preprocessing done by the system. We use computation transformations to define many of the memory models in this thesis.

Informally, we want to implement an abstract memory model using a system with some underlying memory. The clients specify a computation for the abstract memory, which the system transforms into a computation for the underlying memory. This transformation may add or remove precedence dependencies or change the annotations; it need not be deterministic.

For example, a system with a weakly consistent memory can implement sequential consistency by serializing all the operations requested by the clients, adding precedence dependencies so that the operations of the computation submitted to the underlying memory are totally ordered. The serialization order chosen by the system may depend on unpredictable factors, such as network delay and processor load.

We allow the abstract memory and the underlying memory to have different annotation sets, giving them slightly different interfaces. For simplicity, we focus on the case where the memories have the same data type, and the system does not change the operations being requested; that is, the transformation may change the constraints on how the operations are applied but not the operations themselves. The system returns to the clients the values returned by the underlying memory.

Formally, an ***(operation-preserving) computation transformation*** from $\mathcal{D}$ and A to $\mathcal{D}$ and A' is a relation $\mathcal{T}$ from $\mathfrak{C}^{\mathcal{D}}_{\mathsf{A}}$ to $\mathfrak{C}^{\mathcal{D}}_{\mathsf{A}'}$ such that $(\mathsf{C}, \mathsf{C}') \in \mathcal{T}$ implies $V_{\mathsf{C}'} = V_{\mathsf{C}}$. We say that $\mathcal{T}$ ***transforms*** C into C' when $(\mathsf{C}, \mathsf{C}') \in \mathcal{T}$, and we denote the set of computations that $\mathcal{T}$

transforms C into by $\mathcal{T}[C] = \{C' : (C, C') \in \mathcal{T}\}$. When $\mathcal{T}[C]$ has only one element, we may write $\mathcal{T}(C)$ for that element.

Example 4.13 (Serializer) For any $\mathcal{D}$ and A, the following computation transformation serializes the operations: $\mathcal{T}_{ser} = \{(C, C') \in \mathfrak{C}^{\mathcal{D}}_A \times \mathfrak{C}^{\mathcal{D}} : V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge E_{C'} \text{ is a total order}\}$. ■

A memory model $M \in \mathfrak{M}^{\mathcal{D}}_{A'}$ ***using*** a computation transformation $\mathcal{T}$ from $\mathcal{D}$ and A to $\mathcal{D}$ and A' defines a new memory model $\mathcal{T}(M) = \{(C, \rho) : (C', \rho) \in M \text{ for some } C' \in \mathcal{T}[C]\}$. Note that the direction of the transformation is reversed for computations and memory: A transformation that takes abstract computations to real ones, takes a real memory to an abstract one. The abstract memory model admits an observer function for an abstract computation if the underlying memory model admits the observer function for the transformed computation.

Because a memory model defined by another memory model using a transformation is just a computation-centric model like any other, we can reason about it using the theory developed earlier in this chapter. For example, M using $\mathcal{T}$ implements M' using $\mathcal{T}'$ if $\mathcal{T}(M) \subseteq \mathcal{T}'(M')$. Similarly, we can combine computation transformations with client restrictions to say that M using $\mathcal{T}$ implements M' under CR if $\mathcal{T}(M)|_{CR} \subseteq M'|_{CR}$.

Example 4.14 Any memory model using $\mathcal{T}_{ser}$ from Example 4.13 implements sequential consistency; that is, $\mathcal{T}_{ser}(M) \subseteq SC$ for any memory model M. We can see this statement is true because, if $C' \in \mathcal{T}_{ser}[C]$ for any C, then $E_{C'}$ is a total order, so C' has only one schedule. ■

The implementation relation is preserved by transformations.

Lemma 4.6 For $M, M' \in \mathfrak{M}^{\mathcal{D}}_A$ and $\mathcal{T} : \mathfrak{C}^{\mathcal{D}}_A \to \mathfrak{C}^{\mathcal{D}}_A$, if $M \subseteq M'$ then $\mathcal{T}(M) \subseteq \mathcal{T}(M')$.

Proof: If $(C, \rho) \in \mathcal{T}(M)$ then $(C', \rho) \in M \subseteq M'$ for some $C' \in \mathcal{T}[C]$, so $(C, \rho) \in \mathcal{T}(M')$. ■

We can compare computation transformations in much the same way that we compare client restrictions. We say that a transformation $\mathcal{T}$ is ***more restrictive*** than $\mathcal{T}'$ if $\mathcal{T} \subseteq \mathcal{T}'$. Also, we may be interested in the behavior of the transformations assuming the clients obey a restriction. We say that a transformation $\mathcal{T}$ is ***more restrictive*** than $\mathcal{T}'$ under a client restriction CR if $\mathcal{T}[C] \subseteq \mathcal{T}'[C]$ for all $C \in CR$.

A memory using a transformation implements the memory resulting when a less restrictive transformation is used.

Lemma 4.7 If $\mathcal{T}$ is more restrictive than $\mathcal{T}'$ under CR then for any memory model M, $\mathcal{T}(M)$ implements $\mathcal{T}'(M)$ under CR.

Proof: If $(C, \rho) \in \mathcal{T}(M)|_{CR}$ then $C \in CR$ and $\rho \in M[C']$ for some $C' \in \mathcal{T}[C] \subseteq \mathcal{T}'[C]$. Thus, $(C, \rho) \in \mathcal{T}'(M)$. ■

A common way to strengthen the guarantees of a memory is to delay the application of some operations until other operations are known to be done throughout the system. We model this delay by adding precedence dependencies to the requested computation. The resulting set of possible computations typically meets a stricter client restriction under which the memory model guarantees a stronger consistency model. We use this technique to define *synchronization* for memory models in Section 5.5 and Chapters 6 and 7.

For a client restriction $CR \subseteq \mathfrak{C}_A^{\mathcal{D}}$, the computation transformation that ***enforces*** CR is $\mathcal{T}_{CR}^{c} = \{(C, C') : V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge C' \in CR \wedge ann_C = ann_{C'}\}$.

Example 4.15 $\mathcal{T}_{Serial}^{c} = \mathcal{T}_{ser}$, where *Serial* is defined in Example 4.9 and $\mathcal{T}_{ser}$ in Example 4.13. ■

Because a transformation that enforces a client restriction makes computations stricter, a monotonic memory using such a transformation implements the memory without the transformation.

Lemma 4.8 If $M \in \mathfrak{M}_A^{\mathcal{D}}$ is monotonic and $CR \subseteq \mathfrak{C}_A^{\mathcal{D}}$ then $\mathcal{T}_{CR}^{c}(M)$ implements M.

Proof: If $(C, \rho) \in \mathcal{T}_{CR}^{c}(M)$ then $(C', \rho) \in M$ for some $C' \in \mathcal{T}_{CR}^{c}[C]$. Since C' is stricter than C and M is monotonic, $\rho \in M[C'] \subseteq M[C]$, so $(C, \rho) \in M$. ■

The transformation that enforces a client restriction does not eliminate possible observations of a computation that meets the restriction.

Lemma 4.9 If $C \in CR$ then $M[C] \subseteq \mathcal{T}_{CR}^{c}(M)[C]$.

Proof: If $C \in CR$ then $C \in \mathcal{T}_{CR}^{c}[C]$, so $(C, \rho) \in M$ implies $(C, \rho) \in \mathcal{T}_{CR}^{c}(M)$. ■

To show that one memory implements another when both are using a transformation that enforces a client restriction, it suffices to show that the first memory using the transformation implements the second without the transformation.

Lemma 4.10 If $\mathcal{T}_{CR}^{c}(M) \subseteq M'$ then $\mathcal{T}_{CR}^{c}(M) \subseteq \mathcal{T}_{CR}^{c}(M')$.

Proof: If $(C, \rho) \in \mathcal{T}_{CR}^{c}(M)$ then $(C', \rho) \in M$ for some $C' \in \mathcal{T}_{CR}^{c}[C] \subseteq CR$. By Lemma 4.9, $M[C'] \subseteq \mathcal{T}_{CR}^{c}(M)[C']$. So $\rho \in M[C'] \subseteq \mathcal{T}_{CR}^{c}(M)[C'] \subseteq M'[C']$. Since $C' \in \mathcal{T}_{CR}^{c}[C]$, we have $\rho \in \mathcal{T}_{CR}^{c}(M')[C]$. ■

Two transformations that enforce different client restrictions can be combined into a single transformation enforcing both client restrictions, provided that the restrictions are monotonic.

Lemma 4.11 If CR is monotonic then $\mathcal{T}_{CR'}^{c} \circ \mathcal{T}_{CR}^{c} = \mathcal{T}_{CR \cap CR'}^{c}$.

Proof: If $C' \in (\mathcal{T}_{CR'}^{c} \circ \mathcal{T}_{CR}^{c})[C]$ then there exists $C'' \in \mathcal{T}_{CR}^{c}[C]$ such that $C' \in \mathcal{T}_{CR'}^{c}[C'']$, so $V_C = V_{C''} = V_{C'}$, $ann_C = ann_{C''} = ann_{C'}$ and $E_C \subseteq E_{C''} \subseteq E_{C'}$. Since CR is monotonic and C' is stricter than $C'' \in CR$, and since $C' \in range(\mathcal{T}_{CR'}^{c}) \subseteq CR'$, we have $C' \in CR \cap CR'$, so $C' \in \mathcal{T}_{CR \cap CR'}^{c}[C]$.

If $C' \in \mathcal{T}_{CR \cap CR'}^{c}[C]$, then $C' \in \mathcal{T}_{CR}^{c}[C]$, and $C' \in \mathcal{T}_{CR'}^{c}[C']$, so $C' \in (\mathcal{T}_{CR'}^{c} \circ \mathcal{T}_{CR}^{c})[C]$. ■

We can also use computation transformations to define weaker memory models by removing edges rather than adding them. Such a transformation is called a *reordering transformation*. Weakening the consistency guarantees of a memory model is useful when the model is used as a specification rather than an implementation: A relaxed memory model is defined using a reordering transformation with an underlying memory that has strong consistency guarantees. By removing edges from a computation, the reordering transformation relaxes some of the constraints specified by the clients in the computation by allowing the memory to ignore some of the precedence dependencies in the original computation. We use reordering transformations to define several relaxed processor-centric models in Chapter 6, including *weak ordering* [33] and *release consistency* [43].

A computation transformation may provide a conceptual model that suggests a particular implementation. However, as with memory models in general, the transformation may not reflect what the system being modeled actually does.

It is possible to define computation transformations that are not operation-preserving. Such a transformation must specify how to derive return values for operations in the original computation from an observation of the transformed computation running on the underlying memory. In the simplest case, the transformation only adds operations, and the return value for each operation is its return value in the underlying observation. In general, the return value for an operation may be determined by an arbitrary function of the return values for multiple operations of the transformed computation. In this thesis, we use only operation-preserving transformations; we defer the study of these more complicated cases.

4.6 Discussion

Using admissible observations to characterize memory consistency guarantees resembles the *legal histories* approach to specifying memory models [68, 33, 53, 4, 43, 11, 8, 54] in that it is based on a static postmortem description of system behavior. Thus, the two approaches share many strengths and weaknesses. However, by using computations and observer functions to make a clean split between the clients and the memory, we are able to define client restrictions and computation transformations, which we use heavily in later chapters to develop a theory of computation-centric memory models.

Because a computation is a static representation of the clients' requests, a computation-centric model is a static characterization of the memory's guarantees; there is no need to reason about how an execution unfolds. Because admissible observations have well-formed computations with total observer functions, there is no need to consider intermediate observations, in which some operations may not have return values. Thus, computation-centric models are simple to describe and reason about. We can use the tools developed in the previous chapter for computations to reason about computation-centric models.

This simplicity comes at a price: Because clients may use the values returned for earlier operations to determine what operations to request, the computation may depend on the memory model. We saw this kind of dynamic dependence in Examples 3.19 and 3.20. Computation-centric models do not capture the dynamic interaction between the clients and the memory.

Another shortcoming of computation-centric models is that they do not fit into a general theory of distributed systems. To show that a system implements a computation-centric memory model, we must rely on *ad hoc* methods.

In the literature, the alternative to the legal histories approach is to characterize shared memory systems directly as state machines [45, 69, 70, 36, 100].[3] The chief advantages of the state machine approach over the legal histories and computation-centric approaches are that a state machine captures the dynamic interaction between a system and its clients, and that formalizing programs and systems as state machines is typically a straightforward

[3] Although Shen, et al. [100] actually use a term rewriting system to describe their memory model, there is a trivial correspondence between state machines and term rewriting systems.

process, widely practiced throughout computer science. A state machine model is more likely to accurately represent a system than a static model. However, it has proven hard to determine the what properties a memory system guarantees by looking at its state machine representation.

Despite their shortcomings, computation-centric models do capture key characteristics needed to reason about the behavior of memory from the point of view of the clients. In the next four chapters, we use the computation-centric framework to formally specify several memory models from the literature and prove results about the guarantees they provide under various client restrictions. We also compare various models and investigate what properties are actually necessary for the results we prove.

In Chapter 9, we develop another framework based on state machine models for memories that preserve much of the flavor of computation-centric models by using precedence dependencies and annotations to specify client constraints. We use these models to prove that any safety property of the system deduced using the computation-centric framework is also a property of the state machine models of the system (Corollary 9.21). To formalize the state machine models, we use *I/O automata* [81], which support a rich general theory of distributed systems. Thus, this new framework for modeling memory consistency addresses our concerns about the computation-centric framework and has many of the advantages of both the state machine and computation-centric approaches.

The computation-centric models described here are generalized from those we introduced for read/write memory [40] to allow an arbitrary data type for the memory. One disadvantage of this generalization is that, when modeling a read/write memory, a write operation must specify the value written, and the observer function must specify the value read by each read operation. In our earlier work [40], the observer function specified the operation that wrote the value, which made the data flow of an execution explicit. With general data types, the data flow cannot be represented so simply because several operations may contribute to the data read by an operation.

There are several ways in which we could extend the computation-centric framework. It is already flexible enough to handle nondeterministic data types, if we extend our theory of data types in that way. But to handle operations that may block, we need to allow observations with partial return value functions. Permitting partial return value functions would also allow us to reason about computations corresponding to partial executions, in which some operations may not yet have return values. It would also be useful to have a notion of observational equivalence that ignored "internal" operations and return values. Finally, computation transformations could be generalized as described at the end of Section 4.5.

Chapter 5

Simple Memories

In the next four chapters, we look at specific memory models that capture the key features of many memories described in the literature. We show how to define these models formally, along with the client restrictions and computation transformations that we use to prove that one memory model implements another under certain conditions. These models and the results we prove about them are more than illustrations of how to specify and reason about memories in the computation-centric framework; they are important in their own right because they model memories and programming disciplines for real systems.

In this chapter, we consider a few simple but important memory models and client restrictions, showing how to model them in the computation-centric framework. We define *sequential consistency* [68] and *coherence* [47, 52], two of the most common consistency models. They belong to the class of *precedence-based memory models* [77], which can be expressed without annotations. We also show how to use annotations to capture the *synchronization* facilities provided by many memory systems [33, 104, 101, and many others].

We identify the class of *race-free clients* and prove that any memory appears sequentially consistent when accessed by race-free clients. Thus, programmers can safely assume sequential consistency when using weaker memories if their programs are race-free. This result is the main theorem of this chapter, and is the basis for the main theorems in Chapters 6 and 7, which prove that certain weakly consistent memories appear sequentially consistent to a restricted set of clients. We specialize this result to show that a coherent memory is sequentially consistent when no location is accessed concurrently.

The computation-centric memory models defined in this chapter are generalizations of the corresponding memory models usually defined in the literature. Most models in the literature are processor-centric; that is, the program order defined by the precedence dependencies is a total order for each processor. In the computation-centric versions, the program order may be an arbitrary partial order. Thus, the computation-centric framework makes fewer assumptions about the structure of the system, and is suitable for modeling parallel programming systems in which the mapping from computation to processor is not fixed [105].

In addition to specific definitions and results, this chapter illustrates a general approach

The notion of precedence-based memories and a version of Theorem 5.7 appeared in a paper presented at the International Workshop on Distributed Algorithms in 1997 [77].

to defining and reasoning about memory models in the computation-centric framework, which we develop further in later chapters.

Outline: In Section 5.1, we define the class of precedence-based memory models and discuss why it is an interesting class of models. Section 5.2 defines sequential consistency in the computation-centric framework, and Section 5.3 shows how avoiding races allows programmers to assume sequential consistency even when using a much weaker memory. In Section 5.4, we give a computation-centric definition for coherent memory and point out some confusion that has resulted from ambiguous informal definitions of coherence and incompatible formalizations of these definitions that have appeared in the literature. In Section 5.5, we discuss various kinds of synchronization mechanisms and show how to model some of them in the computation-centric framework.

Reading Guide: The main contributions of this chapter are the simple computation-centric definitions of sequential consistency, coherent memory, and race-free clients, and Theorem 5.4, which establishes that race-free clients cannot distinguish weaker memories from sequential consistency. Because of this theorem, programmers following a discipline that guarantees race-freedom can assume sequential consistency, regardless of the actual guarantees of the memory. The development of synchronization in this chapter illustrates how to specify non-precedence-based guarantees of a memory system; we derive specific results about systems with synchronization in later chapters.

5.1 Precedence-Based Memory Models

We begin our study of computation-centric models by focusing on a restricted class of memories, the *precedence-based memories*. A precedence-based memory restricts clients to specify constraints that can be expressed using only precedence dependencies; that is, the computation specified by the clients must not require annotations. In this section, we formally define precedence-based memory models and discuss their benefits and limitations.

A memory model M is ***precedence-based*** if, for all computations C and C' such that $V_{\mathsf{C}} = V_{\mathsf{C}'}$ and $E_{\mathsf{C}} = E_{\mathsf{C}'}$, we have $M[\mathsf{C}] = M[\mathsf{C}']$; that is, M does not distinguish computations that differ only in their annotations. Precedence-based models are easier to reason about because there is no need to consider the annotations. Our analysis of precedence-based models serves both as a guide for how to reason within the computation-centric framework, and as a tool in the analysis of non-precedence-based memories. For example, Theorems 6.13 and 7.25 are corollaries of Theorem 5.4 from this chapter.

Many of the memory models proposed in the literature are precedence-based, including sequential consistency [68], coherence [47, 52], *pipelined RAM* [74], *processor consistency* [47, 8], and *causal memory* [9]. These models are defined in the context of a fixed set of sequential processes accessing the shared memory; that is, they are processor-centric. In Sections 5.2 and 5.4, we define versions of sequential consistency and coherence, the two most important of these models, in the more general setting of computations. In Chapter 6, we examine the processor-centric setting in detail, and we define some of the other models in that context.

The ***generic memory*** from Example 4.3 is also precedence-based. It is the least restrictive memory model, admitting every observation. Its formal definition is:

$$GM = \{(C, \rho) : \forall x \in V_C, \exists \alpha \in Sch(C), \rho(x) = retval(x, \alpha)\}$$

Because generic memory is the weakest memory model, systems that can be built using it can be built using any memory, and properties it guarantees are guaranteed by every memory. Although it is simple, a generic memory is difficult to program because its guarantees are so weak. However, in Section 5.3, we show that programs that generate only completely race-free computations can use generic memory—and thus any memory—as though it were sequentially consistent.

Not all memories, however, are precedence-based. Sometimes we want the system to guarantee different degrees of consistency to different operations because it is sufficent for the correctness of a program for a few operations to be treated specially. Annotations are used to model the degree of consistency an operation requires. The resulting models are called *hybrid* models; non-hybrid models are *uniform* [87]. This hybrid approach was originated by Dubois, et al. in their definition of *weak ordering* [33]. It has also been used to define *hybrid consistency* [12], *release consistency* [43], *eventually-serializable data services* [36], and the *lazy replication* system of Ladin, et al. [66], among others, including most shared memory multiprocessors [104, 101, 84, 58].

A higher level reason to extend precedence-based memory models is that some constraints cannot easily be expressed by dependencies alone. Two important examples of these constraints are "exclusion" constraints and "atomicity" constraints. An exclusion constraint specifies that two sets of operations should not be overlapped; that is, all the operations of one set of operations should precede all the operations of the other, but the two sets may occur in either order. An atomicity constraint specifies that the effects of a set of operations is seen atomically, with no intervening operations. We show how to model these constraints in Chapters 7 and 8 respectively.

5.2 Sequential Consistency

Sequential consistency requires that the memory appear as though it were accessed sequentially [68], that is, as though the operations were handled by a single process. It is a natural and intuitive extension of the semantics of a sequential memory; it was adopted early, and is the only universally accepted memory model for concurrent programming [56]. In this section, we define sequential consistency in the computation-centric framework and discuss the advantages and disadvantages of this model.

The original definition of sequential consistency assumes that several independent processes are accessing the memory, and it requires the apparent serialization of the operations to be consistent with the order—called the *program order*—of the operations at each process. We model ***sequential consistency*** by requiring that a single schedule explain all the return values of a computation. Formally,

$$SC = \{(C, \rho) : \exists \alpha \in Sch(C), \alpha \text{ explains } \rho\}$$

Our definition of sequential consistency does not use the notion of a process. Instead, it relies on the precedence dependencies specified by the computation. We generalize the usual notion of sequential consistency by allowing the program order to be an arbitrary partial order on the operations, rather than a total order for each process. This generalization is especially useful for systems in which processes may be created and destroyed dynamically.

As with all memory models, this definition requires only that the behavior *appear* as though the operations are executed sequentially and consistently with the program order. A system may actually execute operations concurrently, or it may reorder them, as long as it preserves for the clients the appearance of a sequential system. In the computation-centric framework, clients see only the computation they specify and the values returned for the operations, so the appearance of a memory is represented by an observation. An observation is allowed by the sequential consistency model when it is allowed by a sequential system, that is, when all the return values are explained by a single schedule.

When designing and reasoning about their programs, programmers of early concurrent systems assumed their systems were sequentially consistent [28, 91, 22, 13]. Sequential consistency captures the intuition that the memory is a single shared resource being accessed by concurrent processes. It is a natural extension to the memory model of a uniprocessor, and is the memory model naturally guaranteed by a multiprogrammed uniprocessor.[1] For these reasons, sequential consistency is the basic memory consistency model assumed by multiprocessor programmers.

Unfortunately, it is expensive to guarantee sequential consistency in the distributed setting [74]. To ensure sequentially consistent behavior on systems with weaker memory models, programmers often adopt programming disciplines that restrict the computations generated to those for which the guarantees of the weaker memory imply sequential consistency. One such discipline, which we examine in the next section, is to avoid races.

Although it is a natural, widely used and readily understood extension of the semantics of sequential memory to systems that allow concurrent memory access, sequential consistency does not make concurrent programming easy. Much of the early work on multiprogramming, which assumed sequential consistency, was devoted to finding ways to structure concurrent programs [28, 21]; even mutual exclusion for two processes was elusive at first [30]. As Lampson says,

> Even [a sequentially consistent memory] is tricky to use when there are concurrent clients. Experience has shown that it's necessary to have wizards to package it so that ordinary programmers can use it safely. This packaging takes the form of rules for writing concurrent programs and procedures that encapsulate references to shared memory [70, Handout 30, p. 5].

Thus, it makes sense to provide "ordinary programmers" with models that are stronger than sequential consistency and easier to use safely. Such models often augment sequential consistency with high level features such as *locks* and *transactions*, which we discuss in Chapters 7 and 8 respectively.

[1] A multiprogrammed uniprocessor guarantees *linearizability* [53], which is a stronger property, but equivalent to sequential consistency when all communication between clients occurs through the shared memory. We discuss linearizability in Section 9.6.

5.3 Eliminating Races

When the clients request two conflicting operations concurrently, the result depends on the order in which the operations are applied to the memory. These operations comprise a *race*, and are said to *compete*. These terms are defined formally in Section 3.7. Races make the result of a program dependent on the schedule, which is determined by uncertain and unpredictable variables, such as communication latency and processor speed and load. This dependence makes programs with races difficult to reason about. Thus, races are often considered bad programming practice, and much research is devoted to detecting races [85, 6, 89, 90, 88, 96, 37, 25, 24, and many others].

Races are often considered bad programming practice because they make the result of a program dependent on the schedule, which is determined by uncertain and unpredictable variables, such as communication latency and processor speed and load. In this section, we show how a discipline of avoiding races allows a programmer to assume sequentially consistent semantics when using weaker memories.

The crucial property of race-free clients is that the computations they specify are determinate. We say that clients are ***safe*** if they specify only determinate computations,[2] and they are ***(completely) race-free*** if they specify only completely race-free computations. We characterize the clients formally with client restrictions:

$$\mathit{Safe} = \left\{ C \in \mathfrak{C}_A^{\mathcal{D}} : C \text{ is determinate} \right\}$$
$$\mathit{RF} = \left\{ C \in \mathfrak{C}_A^{\mathcal{D}} : C \text{ is completely race-free} \right\}$$

Because race-free computations are determinate, race-free clients are safe.

Lemma 5.1 *RF* is more restrictive than *Safe*.

Proof: Immediate from Lemma 3.2. ■

We show in Lemma 5.3 below that, for safe clients, all complete memory models for a given data type and annotation set are equivalent. The key to proving Lemma 5.3 is the following lemma, which says that every schedule of a determinate computation yields the same return values.

Lemma 5.2 If C is determinate then there is a unique observer function ρ for C.

Proof: Suppose ρ, ρ' are observer functions for C. For any $x \in V_C$, there are schedules α and α' of C such that $\rho(x) = \mathit{retval}(x, \alpha)$ and $\rho'(x) = \mathit{retval}(x, \alpha')$. Since C is determinate, $\mathit{retval}(x, \alpha) = \mathit{retval}(x, \alpha')$, so $\rho(x) = \rho'(x)$. Thus, $\rho = \rho'$. ■

For any client restriction that implies client safety, all complete memory models under the restriction are equivalent under the restriction.

Lemma 5.3 If $CR \subseteq \mathit{Safe}$ and $M, M' \in \mathfrak{M}_A^{\mathcal{D}}$ are complete under CR then M and M' are equivalent under CR.

[2] We do not say the clients are determinate because they may nondeterministically choose which computation to request. This choice would be specified by a model for the clients.

Proof: Let $C \in CR$. If $\rho \in M[C]$, then ρ is an observer function for C. Because M' is complete under CR, there is an observer function $\rho' \in M'[C]$. Since C is determinate, $\rho = \rho'$ by Lemma 5.2, so $\rho \in M'[C]$, and thus $M[C] \subseteq M'[C]$. Similarly, $M'[C] \subseteq M[C]$. Thus, $M[C] = M[C']$ for all $C \in CR$, so M and M' are equivalent under CR. ■

It follows immediately from these results that any memory model implements sequential consistency under complete race-freedom.

Theorem 5.4 Any memory model implements *SC* under *RF*.

Proof: Every memory model implements *GM*, so by Lemma 4.3, every memory model implements *GM* under *RF*. By Lemma 5.1 and Lemma 5.3, *GM* and *SC* are equivalent under *RF*, so every memory model implements *SC* under *RF*. ■

Theorem 5.4 is the first of several results in this thesis that support concrete disciplines that enable programmers to assume strong consistency guarantees when using weakly consistent memories. Based on this theorem, a programmer can confidently program any memory assuming sequentially consistent semantics, as long as the program is completely race-free.

Complete race-freedom is a very strict discipline; completely race-free programs are difficult to write without eliminating much of the possible concurrency, which would defeat the purpose of using multiple processors. Sometimes, the programmer may not care which way two competing operations are ordered, as long as they are ordered consistently.

Rather than requiring clients to be completely race-free, a system with a weakly consistent memory may provide a front end that resolves races. This shifts the responsibility of ensuring race-freedom from the clients onto the system, which orders competing operations. An advantage of this approach is that the system can order competing operations based on the current execution in a way that improves performance.

We model such a system using a computation transformation that enforces race-freedom:

$$\mathcal{T}^{c}_{RF} = \left\{ (C, C') \in \mathfrak{C}^{\mathcal{D}}_{A} : V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge C' \in RF \wedge ann_C = ann_{C'} \right\}.$$

Any system that eliminates races implements sequential consistency regardless of the underlying memory model.

Theorem 5.5 For any memory $M \in \mathfrak{M}^{\mathcal{D}}_{A}$, we have $\mathcal{T}^{c}_{RF}(M) \subseteq SC$.

Proof: If $(C, \rho) \in \mathcal{T}^{c}_{RF}(M)$ then $(C', \rho) \in M$ for some $C' \in \mathcal{T}^{c}_{RF}[C]$. By the definition of $\mathcal{T}^{c}_{RF}$, $C' \in RF$, so by Theorem 5.4, $(C', \rho) \in SC$. Because *SC* is monotonic and C' is stricter than C, we have $(C, \rho) \in SC$, as required. ■

One problem with this approach is that the system, not knowing the entire computation ahead of time, cannot tell which operations will compete. Thus, it needs to keep track of all the operations that are executed by distributed processes, which is exactly the difficulty that systems implementing sequential consistency face.

Many memory systems [104, 66, 15, 84, 101, and others] provide special operations that "protect" against races, so that competing operations separated by these operations are not considered races. The memory guarantees greater consistency and synchronization for these special operations. Programming with such memories can be difficult, because

the programmer must reason about various types of consistency, and must specify for each operation, what kind of consistency it has. In Section 5.5 and Chapters 6 and 7, we discuss how to model memories with these operations.

Adve and Gharachorloo advocate a *programmer-centric approach* to specifying memory consistency [2, 1, 41]. A programmer-centric memory model guarantees sequential consistency as long as the programmer properly identifies which operations may compete. Thus, to guarantee sequential consistency, the system does not need to keep track of all the operations—only those that the programmer identifies. The programmer-centric approach provides an intermediate level of programming complexity; the programmer only needs to reason about one type of consistency guarantee—sequential consistency—but needs to identify which operations compete. We discuss these models in more depth in Section 6.7.

5.4 Coherent Memory

Because sequentially consistent memory is expensive to implement, most modern multiprocessors provide weaker consistency guarantees. Although there is no consensus on the "right" consistency, almost all models proposed in the literature guarantee *coherence*, a very weak consistency guarantee. In this section, we give a formal model for coherent memory, and we define a discipline for accessing coherent memory so that one can assume sequential consistency. The notion of coherence arose informally and has been interpreted in different ways. At the end of this section, we discuss some of the confusion caused by this ambiguity.

The notion of coherence comes from caching protocols, where relatively small chunks of the memory are stored locally at a processor for faster access. Each operation accesses only one chunk, so this partitioning of memory defines *locations*. Coherence guarantees sequential consistency for each location separately; that is, looking only at the operations on any single location, a coherent memory appears sequentially consistent. This model has also been called *per-location sequential consistency*, *location consistency*, and *cache consistency* [77, 40, 39, 47].

Formally, suppose that $\{O_\ell\}_{\ell \in \mathcal{L}}$ is a location partition of $\mathcal{D}$. Coherence is modeled as follows:

$$\begin{aligned} Coh &= \{(C, \rho) : \forall \ell \in \mathcal{L}, \exists \alpha \in Sch(C), \forall x \in V_C, (x.loc = \ell \implies \rho(x) = retval(x, \alpha))\} \\ &= \{(C, \rho) : \forall \ell \in \mathcal{L}, \exists \alpha \in Sch(C), \alpha \text{ explains } \rho|_\ell\} \end{aligned}$$

For a memory with locations of fine granularity, such as those of most shared memory multiprocessors, coherence is a very weak guarantee. Most systems have special *synchronization* operations, for which they guarantee additional properties. We discuss how to handle such operations in Section 5.5 and later chapters.

For a memory system without synchronization operations, we can get stronger guarantees by restricting the clients. In particular, if clients never request concurrent accesses to the same location, then a coherent memory guarantees sequential consistency. We say that a computation ***separates locations*** $\mathcal{L}$ if it has no concurrent accesses to the same location $\ell \in \mathcal{L}$. Clients ***separate locations*** if they specify only computations that separate locations. Let $SepLocs = \{C : C \text{ separates locations}\}$.

Because operations performed on different locations are independent, clients that separate locations are completely race-free. Thus, such clients can assume sequentially consistent semantics when using coherent memory.

Lemma 5.6 *SepLocs* $\subseteq$ *RF*.

Proof: For $C \in$ *SepLocs*, if $x, y \in V_C$ are concurrent in C then they do not access the same location, so they are independent. Thus, $C \in$ *RF*. ■

Theorem 5.7 *Coh* implements *SC* under *SepLocs*.

Proof: Immediate from Theorem 5.4 and Lemma 5.6. ■

It seems strange to specify coherence in Theorem 5.7 when any memory implements sequential consistency under *SepLocs*. However, the literature often states the result in this way (e.g., [26]). Why require coherence? We conjecture that this stems from a confusion about reordering: Many processor-centric models, which we discuss in Chapter 6, are defined by allowing some operations to be reordered. To maintain coherence, a system cannot reorder operations on the same location. Reordering operations on the same location is generally proscribed because, as we discuss in Section 6.5, a system that reorders operations on the same location may admit "observations" with return value functions that are not observer functions. Because this condition is the same condition required for coherence, it seems to be generally accepted that a system should be coherent. But, as we show in Section 6.3, at least one common memory model, *release consistency*, is not coherent.

Theorem 5.7 is the basis for a simple discipline for programming coherent memories. A programmer who ensures that accesses to each location are explicitly ordered by precedence dependencies can assume any coherent memory is sequentially consistent. Since most multiprocessors guarantee coherence, ensuring race-freedom is a portable, though restrictive, programming style.

There is some ambiguity in the literature about the exact definition of coherence, which usually appears in the context of read/write memory. We use the definitions of Ahamad, et al., and Culler and Singh [8, 26], generalized to accommodate arbitrary data types with locations. Other researchers define coherence on a shared read/write memory multiprocessor by requiring that the writes to each location appear to occur in the same order at every processor [41, 52]. This property, called *write serialization*, is not identical to coherence, although it is often treated as such [26, 41]. We discuss the relationship between write serialization and coherence in more depth in Section 6.3.

Some researchers distinguish coherence from "consistency". For example, Hennessy and Patterson say that coherence "defines what values can be returned by a read," while consistency "determines when a written value will be returned by a read" [52, p. 656]. This definition of consistency ties memory semantics to timing issues; the possible observations depend on the time that operations are issued or completed, which cannot be determined from the program. They also say that "coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations" [52, p. 657]. This distinction, however, is not entirely well-defined: Since the operations requested depend on the values returned for other operations, the behavior of reads and writes at one location can affect

the behavior at another location. For these reasons, we consider any restriction on the possible observations of a system to be a consistency guarantee. Coherence is simply a weak consistency guarantee.

5.5 Synchronization

Concurrent systems are often described in terms of *threads*, where a thread is a sequential "flow of control".[3] Several threads may execute concurrently, communicating and interfering with each other. A system typically provides special *synchronization* primitives to limit the concurrency and prevent interference. In this section, we discuss some synchronization mechanisms and how to model them in the computation-centric framework. These ideas arise again in later chapters.

Barrier synchronization refers to bringing two or more threads to a single control point. This kind of synchronization is provided by the *join* operation in *fork/join* parallelism [27] and the *coend* statement in the *cobegin/coend* construct [29]. Barrier synchronization can be modeled by precedence dependencies, as illustrated in Examples 3.22 and 3.23. *Fence* and *memory barrier* operations of shared memory multiprocessors, which we discuss in Chapter 6, also provide barrier synchronization.

Synchronization also refers to a variety of mechanisms that support, or are supported by, barrier synchronization. These mechanisms include *semaphores*, *locks*, *critical sections*, and *synchronized variables*. Operations that provide synchronization in any of these forms are called *synchronization operations*. Rather than discuss each mechanism, we introduce some general principles for how to reason about synchronization in this section, and in Chapter 7, we consider locks and critical sections in detail.

Informally, synchronization fixes the order of the synchronization operations at run time. For barrier synchronization, the order is given explicitly by the clients. For other kinds of synchronization, this order is not specified, and the system can apply the operations in any order, but it must apply them consistently in that order. Synchronization usually also imposes constraints on the other operations. What these constraints are exactly define the different variants of synchronization.

We model synchronization using a computation transformation that is defined by a *synchronization predicate*. The synchronization predicate, which depends on the operations and the computation, indicates which operations must be synchronized relative to each other. The transformation adds precedence dependencies to the computation so that all such operations are ordered by the computation.

Formally, suppose $\Psi(x, y, C)$ indicates that x and y should be synchronized, where $x, y \in V_C$. A computation that is synchronized according to this predicate is one in which operations that satisfy the predicate are ordered by the computation. Such computations satisfy the following client restriction:

$$CR^s_\Psi = \{C : \forall x, y \in V_C, \Psi(x, y, C) \implies x \preceq_C y \vee y \preceq_C x\}.$$

[3] Because we only use this term informally, this description suffices for our discussion.

The ***synchronizing transformation*** enforces this client restriction. Formally,

$$\begin{aligned}\mathcal{T}^{s}_{\Psi} &= \mathcal{T}^{c}_{CR^{s}_{\Psi}} \\ &= \left\{ (C, C') : \begin{array}{l} V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge ann_C = ann_{C'} \\ \wedge\ \forall x, y \in V_C, (\Psi(x, y, C) \implies x \preceq_{C'} y \vee y \preceq_{C'} x) \end{array} \right\}\end{aligned}$$

We often indicate synchronization operations by the annotation SYNC.

Example 5.1 A memory with *weak synchronization* guarantees only that synchronization operations are seen in the same order by all operations; that is, two operations synchronize when both are annotated by SYNC. Formally, let $\Psi_W(x, y, C) \equiv ann_C(x) = \text{SYNC} \wedge ann_C(y) = \text{SYNC}$, so

$$\begin{aligned}CR^{s}_{\Psi_W} &= \{C : \forall x, y \in V_C, \Psi_W(x, y, C) \implies x \preceq_C y \vee y \preceq_C x\} \\ &= \{C : \forall x, y \in V_C, ann_C(x) = ann_C(y) = \text{SYNC}, \implies x \preceq_C y \vee y \preceq_C x\},\end{aligned}$$

and $WSync = \mathcal{T}^{s}_{\Psi_W}(GM)$. ■

Example 5.2 *Strong synchronization* guarantees that all operations are ordered consistently with respect to the synchronization operations; that is, two operations synchronize when either is annotated by SYNC. Formally, $\Psi_S(x, y, C) \equiv ann_C(x) = \text{SYNC} \vee ann_C(y) = \text{SYNC}$, which defines $CR^{s}_{\Psi_S}$ and $\mathcal{T}^{s}_{\Psi_S}$, and $SSync = \mathcal{T}^{s}_{\Psi_S}(GM)$. ■

Example 5.3 Sequential consistency requires all operations to be ordered consistently; that is, every pair of operations must be synchronized. Thus, $SC = \mathcal{T}^{s}_{True}(GM)$. ■

A stronger synchronization predicate means that fewer operations must be synchronized, which results in a weaker memory model.

Lemma 5.8 If $\Psi \implies \Psi'$ then $\mathcal{T}^{s}_{\Psi'}(M) \subseteq \mathcal{T}^{s}_{\Psi}(M)$ for any memory model M.

Proof: The definitions of $\mathcal{T}^{s}_{\Psi}$ and $\mathcal{T}^{s}_{\Psi'}$ immediately imply that $\mathcal{T}^{s}_{\Psi'}$ is more restrictive than $\mathcal{T}^{s}_{\Psi}$. So by Lemma 4.7, $\mathcal{T}^{s}_{\Psi'}(M) \subseteq \mathcal{T}^{s}_{\Psi}(M)$. ■

Client restrictions corresponding to synchronization are always monotonic.

Lemma 5.9 For any synchronization predicate Ψ, CR^{s}_{Ψ} is monotonic.

Proof: Immediate from the definition of CR^{s}_{Ψ}. ■

Two synchronizing transformations can be combined into one transformation that synchronizes the operations of both the original transformations.

Lemma 5.10 $\mathcal{T}^{s}_{\Psi} \circ \mathcal{T}^{s}_{\Psi'} = \mathcal{T}^{s}_{\Psi \vee \Psi'}$.

Proof: By definition,

$$\begin{aligned}
CR^{s}_{\Psi\vee\Psi'} &= \{C : \forall x, y \in V_C, (\Psi \vee \Psi')(x, y, C) \implies x \preceq_C y \vee y \preceq_C x\} \\
&= \left\{ C : \begin{array}{r} \forall x, y \in V_C, (\Psi(x, y, C) \implies x \preceq_C y \vee y \preceq_C x) \\ \wedge\ (\Psi'(x, y, C) \implies x \preceq_C y \vee y \preceq_C x) \end{array} \right\} \\
&= \{C : \forall x, y \in V_C, \Psi(x, y, C) \implies x \preceq_C y \vee y \preceq_C x\} \\
&\quad \cap \{C : \forall x, y \in V_C, \Psi'(x, y, C) \implies x \preceq_C y \vee y \preceq_C x\} \\
&= CR^{s}_{\Psi} \cap CR^{s}_{\Psi'}
\end{aligned}$$

By Lemma 5.9, $CR^{s}_{\Psi'}$ is monotonic, so $\mathcal{T}^{s}_{\Psi} \circ \mathcal{T}^{s}_{\Psi'} = \mathcal{T}^{c}_{CR^{s}_{\Psi}} \circ \mathcal{T}^{c}_{CR^{s}_{\Psi'}} = \mathcal{T}^{c}_{CR^{s}_{\Psi} \cap CR^{s}_{\Psi'}} = \mathcal{T}^{c}_{CR^{s}_{\Psi\vee\Psi'}} = \mathcal{T}^{s}_{\Psi\vee\Psi'}$, by the definitions and Lemma 4.11. ■

Synchronization can be viewed as precedence dependencies that are determined dynamically at run time. Why then do we use annotations to model synchronization, instead of precedence dependencies? There are three sets of issues that make explicit precedence dependencies difficult or inappropriate: language, semantics and modeling.

The simplest reason to prefer synchronization over explicit dependencies is that the language in which programs are written may not have the facilities to indicate the dependencies between operations. For example, concurrent programs are often collections of sequential threads that are run concurrently on separate processors and communicate only through the shared memory. To order operations at different processors, one thread must write a variable that another thread reads, and the latter thread must be able to determine that the value it receives was written by the appropriate operation of the first thread. For simple programs such as the one from Example 3.20, making such a determination may be easy, but in general, it may be quite subtle. In addition, because the threads use the shared memory to coordinate, the ordering guarantees on these operations depend on the guarantees of the memory.

Even if we have a rich language in which to write concurrent programs, we may not care what order some operations are executed, as long as they are ordered consistently. Ordering such operations explicitly in the program does not capture the semantics intended by the programmer and may result in inefficiencies in the program by unnecessarily delaying the execution of some operations.

In the literature, a common way to get a fixed order for synchronization operations without requiring the order to be fixed by the program is to use the execution order of the operations [33, 1, 41, 55, and many others]. The problem with this approach is that the order in which the operations are executed is determined at run time and depends on factors that the programmer cannot predict. Using the execution order to determine precedence dependencies widens the gap between computations and programs and muddies the clean split between the memory system and its clients that computations provide. One of the chief motivations for developing the computation-centric framework was to provide this clean split, and the lack of it, we believe, is a major cause of confusion and difficulty in reasoning about memory models in the literature. Choosing precedence dependencies to model only the dependencies explicit in the program, not the incidental ordering of operations in an execution, is one of the central modeling decisions we made for computations. Although computations are derived from executions and programs may generate many

different computations depending on the values returned by the memory, the computation generated in any execution should depend only on the control decisions that are explicit in the program. Given the values returned for each operation, a programmer should be able to determine the computation generated.

A more abstract way to model synchronization in computations is encode into the annotations the values returned for previous synchronization operations. Instead of maintaining a consistent order for the synchronization operations, the system guarantees that the return value of a synchronization operation is explained by some schedule that also explains all the return values specified in the annotation. This approach, which we call *value synchronization*, is attractive because it closely models the information available to the programmer. We have not developed any theory about this approach, but we believe it would be a fruitful direction for future study.

Chapter 6

Processor-Centric Memories

Much of the prior work on modeling shared memory focuses on *processor-centric memories*, which are accessed by a fixed set of processors, each requesting operations sequentially. These memories are usually coherent read/write memories with some synchronization. All inter-processor communication is mediated by the memory, so there are no explicit control dependencies across processors. Processor-centric memories form a natural class of memories to consider when studying shared memory multiprocessors; their memory is typically of this kind. In this chapter, we discuss processor-centric memory models that have been proposed in the literature and how they fit into the computation-centric framework.

Many processor-centric models were developed to model specific architectures that use a variety of techniques, both in hardware and in the compiler, to improve the performance of the memory. These architectures may reorder operations issued by the clients, and they may keep multiple, possibly inconsistent, copies of some memory locations. The models give an abstract description of the machine implementing the memory, and indicate which operations may be reordered. These models expose hardware features, such as *write buffers* and *caches*, to the programmer. These features do not necessarily represent real hardware, but provide a conceptual model to reason about the behavior of the memory. We show how to model reordering, write buffers and caches in the computation-centric framework, and present computation-centric versions of several processor-centric models that have been proposed in the literature.

Most multiprocessors implement coherence by synchronizing write operations to the same location. We show that this technique, called *write serialization*, is not always sufficient to guarantee coherence, and we give conditions under which it is sufficient.

The *system-centric approach* to specifying memory models described above is most useful for system implementors, as it makes explicit what techniques may be used to reduce or hide memory latency. However, system-centric models are complicated; even experts find it difficult to reason about them [56].

There is an alternative *programmer-centric approach* to specifying memory consistency guarantees [1, 41]: A memory model is defined by the class of programs guaranteed to execute on the memory as on sequentially consistent memory. When using a system that implements a programmer-centric model, a programmer can assume sequential consistency as long as the program is in the class defined by the programmer-centric model.

We adapt this approach to the computation-centric framework, defining the class of *data-race-free programs*. We show that the system-centric *weak ordering* memory model [33, 4] implements the programmer-centric model defined by data-race-free programs.

Outline: Section 6.1 describes several characteristics common to many processor-centric models, and Section 6.2 shows how to express these models in the computation-centric framework, giving computation-centric versions of many memory models proposed in the literature. Section 6.3 discusses write serialization. In Section 6.4, we compare the guarantees of various processor-centric models for "ordinary" reads and writes—those with the minimal synchronization allowed by the model. Section 6.5 points out a danger in the way we model the reordering of operations and shows how we avoid it. In Section 6.6, we consider two ways in which reordering can be interpreted by a programmer. Section 6.7 shows how to incorporate programmer-centric models into the our framework and proves that weak ordering implements that data-race-free model. Finally, in Section 6.8, we discuss the strengths and weaknesses of processor-centric models and why we advocate the more flexible model of concurrency afforded by the computation-centric framework.

Reading Guide: This chapter is a digression from the main work in this thesis; it may be skipped with little loss in continuity. We believe that the processor-centric view is a flawed way to specify memories for programmers, except at the lowest level, when programmers are exposed to the underlying hardware. However, because much of the work on modeling memory consistency adopts the processor-centric view, it is important to understand how this view relates to the computation-centric framework we propose. In particular, Section 6.2 gives computation-centric versions of many system-centric models for processor-centric memories. Section 6.7 shows how to specify a programmer-centric model, the *data-race-free* memory model, and proves that a broad class of system-centric models implement the data-race-free model.

6.1 Characteristics of Processor-Centric Models

By definition, a processor-centric memory is accessed by a fixed set of processors. Although the processors may be nonblocking, they issue instructions sequentially; that is, each processor specifies a sequence of operations to the memory. The most important implication of this requirement is that thare are are no control dependencies between operations requested by different processors. Most models of processor-centric memories share several other assumptions or characteristics. In this section, we informally discuss these characteristics. This discussion applies only to *system-centric models* of processor-centric systems, not to *programmer-centric models*, which we discuss in Section 6.7.

Processor-centric models are typically described as weakenings, or relaxations, of sequential consistency [2, 52]. The guarantees of sequential consistency are weakened by allowing some operations to be reordered, and by presenting processors with a view of memory in which operations are not atomic. Reordering relaxes the ordering constraints expressed by the sequence of operations requested by each processor. The nonatomic view

relaxes the consistency constraint that requires that the values returned by operations from all the processors be explained by a single schedule.

The data type of a processor-centric memory is usually a read/write memory with many locations, so that each operation accesses only a small piece of the data. The ordering constraints are based on the types of the operations. For example, a memory may allow reads to overtake writes, but not vice versa, or it may allow writes to different locations to be reordered.

Sometimes the memory has simple read-modify-write operations, such as test-and-set, that access a single location. In these cases, the models are typically described in terms of reads and writes, where read-modify-write operations must satisfy the constraints of both reads and writes.

The operations of a processor-centric memory may fail to be atomic because the memory maintains multiple copies of some locations. This lack of atomicity may be exposed to the processors when the copies are allowed to be inconsistent. A processor-centric memory model gives a conceptual mechanism to describe this view. Two common mechanisms are *write buffers with read forwarding*[1] and *caches*, or local copies of the memory. A write buffer with read forwarding allows a read to complete before prior writes to the same location by *forwarding* the value of the last such write in the buffer. A cache allows both reads and writes to complete by accessing a local copy of the memory location, which may not reflect all the writes completed by other processors. This description is intended for reasoning about the memory, and may not describe the actual implementation of the memory.

A memory without caches has a single schedule that "explains" the values returned by the operations of all the processors. With read forwarding, a read may complete—that is, appear in the schedule—before the write whose value it reads.

A memory with caches has a separate schedule for each processor, corresponding to the order in which the operations are reflected in the cache of the processor. In the literature, being reflected in the cache of a processor is called being *performed at* that processor [33, 2]. In addition to the ordering constraints, the schedules typically must satisfy a consistency requirement. For example, most processor-centric memories are coherent.

Many processor-centric memories provide special operations to enforce ordering between other operations and to limit the inconsistency between the schedules of different processors. These operations may be explicit *fence* or *memory barrier* instructions, or they may be ordinary instructions that inhibit reordering in the processor, as conditional branches do in many processors.

6.2 Processor-Centrism in the Computation-Centric Framework

In this section, we show how to model processor-centric memories in our computation-centric framework. Specifically, we show how to express a processor-centric model as a computation-centric model that has the same semantics and captures much of the same flavor. We give examples of processor-centric models from the literature.

[1] Write buffers without read forwarding do not present a nonatomic view of the memory; they simply allow operations to be reordered before buffered writes.

Let $\mathcal{P}$ be the fixed set of processors accessing memory. Because the memory knows which processor requested each operation, we assume there is a function $p\colon O \to \mathcal{P}$, such that $p(x)$ is the processor that requested x.[2] For a processor $p \in \mathcal{P}$ and a set X of operations, we denote the set of operations requested by p by $X|_p = \{x \in X : p(x) = p\}$. Similarly, $\alpha|_p = \alpha|_{elems(\alpha)|_p}$ is the ***projection*** of α onto processor p, and $\rho|_p = \rho|_{domain(\rho)|_p}$.

The data type of the memory is $\mathcal{M}$ with address, or location, set $\mathcal{L}$ (see Example 2.2), extended with the noop operator defined on page 25. The operations are implicitly tagged with identifiers that distinguish different invocations of the same operator, as discussed in Section 2.5. Because noop is independent of all operations, including itself, each noop operation is considered to be performed on a different location from every other operation, including other noop operations. As mentioned in Section 2.9, for a location $\ell \in \mathcal{L}$, we write $X|_\ell = \{x \in X : x.loc = \ell\}$ and $\alpha|_\ell = \alpha|_{X|_\ell}$. We denote the set of write operations by $Wr = \{\mathsf{write}(\ell, k) : \ell \in \mathcal{L}, k \in \mathbb{Z}\}$, and the set of read operations by $Rd = \{\mathsf{read}(\ell) : \ell \in \mathcal{L}\}$. For $x \in Wr$, we denote by $x.v$ the value written by x, that is, if $x = \mathsf{write}(\ell, k)$, then $x.v = k$.

Because each processor requests operations sequentially and there are no dependencies across processors, two operations are ordered by the precedence dependencies if and only if they are requested by the same processor. We model this requirement as a well-formedness condition on the computations. Formally, the set of well-formed computations for a processor-centric model with annotation set A is

$$WF_P = \left\{C \in \mathfrak{C}_A^{\mathcal{M}} : p(x) = p(y) \iff (x, y) \in E_C \vee (y, x) \in E_C \vee x = y\right\}.$$

For the rest of this section, we assume that all computations are well-formed; all the memory models defined in this section are implicitly restricted to well-formed computations.

Reordering. A processor-centric model specifies which types of operations may be reordered, and describes an underlying memory on which the operations are executed. We express reordering in the computation-centric framework by giving a model for the underlying memory and a transformation that takes a processor-centric computation to a computation of this memory model. The transformation eliminates precedence dependencies between operations that may be reordered and changes the annotations to be appropriate for the underlying memory model.

Formally, a processor-centric memory model $M \in \mathfrak{M}_A^{\mathcal{M}}$ is described by an underlying memory model $M' \in \mathfrak{M}_{A'}^{\mathcal{M}}$ and a *reordering transformation*[3] $\mathcal{T}_M^r\colon \mathfrak{C}_A^{\mathcal{M}} \to \mathfrak{C}_{A'}^{\mathcal{M}}$ such that $M = \mathcal{T}_M^r(M') = \left\{(C, \rho) : (\mathcal{T}_M^r(C), \rho) \in M'\right\}$. We define a reordering transformation using a predicate $Preserve_M(x, y, a_x, a_y)$, which indicates whether M preserves the program order between x and y when x precedes y and they are annotated by a_x and a_y respectively. The transformed computation retains only those edges that satisfy this predicate; that is, $E_{\mathcal{T}_M^r(C)} = \{(x, y) \in E_C : Preserve_M(x, y, ann_C(x), ann_C(y))\}$.

Some memories present an atomic view of the memory to the processors, and do not allow read forwarding. The underlying model of these memories is sequential consistency, and reordering is sufficient to describe the relaxations they allow. For such a memory

[2] Alternatively, this information could be given in the annotations, as in Examples 3.1 and 3.11.

[3] A reordering transformation is a special kind of computation transformation, as defined in Section 4.5.

model M with annotations A, we have $\mathcal{T}^{\mathrm{r}}_M \colon \mathfrak{C}^{\mathcal{M}}_A \to \mathfrak{C}^{\mathcal{M}}$, so $ann_{\mathcal{T}^{\mathrm{r}}_M(C)}(x) = \text{NIL}$ for all x, and $M = \mathcal{T}^{\mathrm{r}}_M(SC)$.

Example 6.1 (IBM/370) The IBM/370 model [58] allows reads to be reordered before writes to different locations. Other than this relaxation, the IBM/370 guarantees sequential consistency. It also has instructions that act as *fences*; they are never reordered. We specify fences with the annotation FNC; other operations have no annotation. The reordering transformation $\mathcal{T}^{\mathrm{r}}_{IBM} \colon \mathfrak{C}^{\mathcal{M}}_{\{\text{FNC}\}} \to \mathfrak{C}^{\mathcal{M}}$ is defined using

$$Preserve_{IBM}(x, y, a_x, a_y) \equiv x \in Rd \vee y \in Wr \vee x.loc = y.loc \vee a_x = \text{FNC} \vee a_y = \text{FNC}.$$

We can also represent the $Preserve_{IBM}$ predicate as a table that indicates, for any two operations, whether the order between them must be preserved.

$Preserve_{IBM}$ x, a_x \ y, a_y	read(ℓ') NIL	write($\ell', \cdot$) NIL	noop FNC
read(ℓ), NIL	$\surd$	$\surd$	$\surd$
write($\ell, \cdot$), NIL	$\bullet$	$\surd$	$\surd$
noop, FNC	$\surd$	$\surd$	$\surd$

The mark "$\surd$" indicates that the order between operations must always be preserved; "$\bullet$" indicates the order needs to be preserved only if they access the same location.

The memory model is $IBM = \mathcal{T}^{\mathrm{r}}_{IBM}(SC) = \{(C, \rho) : (\mathcal{T}^{\mathrm{r}}_{IBM}(C), \rho) \in SC\}$. ■

Example 6.2 (Alpha) The underlying memory model of an Alpha processor [101] is also sequential consistency, but the Alpha allows almost all operations to be reordered. In particular, any read and write operations may be reordered unless they are performed on the same location. The Alpha provides two fence operations to enforce ordering constraints. The *memory barrier*, indicated by the annotation MB, is never reordered; the *write memory barrier* indicated by the annotation WMB, only enforces the ordering between writes.

Formally, $Alpha = \mathcal{T}^{\mathrm{r}}_{Alpha}(SC)$, where we define $\mathcal{T}^{\mathrm{r}}_{Alpha} \colon \mathfrak{C}^{\mathcal{M}}_{\{\text{MB}, \text{WMB}\}} \to \mathfrak{C}^{\mathcal{M}}$ using

$$\begin{aligned} Preserve_{Alpha}(x, y, a_x, a_y) \equiv\ & x.loc = y.loc \vee a_x = \text{MB} \vee a_y = \text{MB} \\ & \vee (a_x = \text{WMB} \wedge y \in Wr) \vee (a_y = \text{WMB} \wedge x \in Wr), \end{aligned}$$

or in table form,

$Preserve_{Alpha}$ x, a_x \ y, a_y	read(ℓ') NIL	write($\ell', \cdot$) NIL	noop MB	noop WMB
read(ℓ), NIL	$\bullet$	$\bullet$	$\surd$	-
write($\ell, \cdot$), NIL	$\bullet$	$\bullet$	$\surd$	$\surd$
noop, MB	$\surd$	$\surd$	$\surd$	$\surd$
noop, WMB	-	$\surd$	$\surd$	-

As in Example 6.1, the order is preserved when indicated by "$\surd$", or by "$\bullet$" and the operations access the same location. The mark "-" indicates that the operations may always be reordered. ■

There is a danger, when using reordering transformations to define memory models, that some return value functions associated with computations in the result will not be observer functions for their associated computation. That is, the result of using a reordering

transformation may not be a memory model. In Section 6.5, we discuss this danger, and why it is not a problem for the memory models defined here.

We can view the difference between sequential consistency and the IBM/370 and Alpha models less as a difference in the consistency models, and more as a difference in the way computations are derived from executions. Specifically, instead of saying that the computation defines a total order for each processor, we could say that it only includes dependencies between operations that may not be reordered, as we saw in Example 3.17. We discuss this view further in Section 6.6.

Read Forwarding. To model a write buffer with read forwarding, we eliminate the precedence dependency of a read on prior writes to the same location, and we annotate the read with the write immediately preceding it in the program order. If the read is scheduled before the write that annotates it, then it returns the value written by that write; that is, the value is *forwarded* to the read.

To specify memory models with this optimization, we define two functions. First, for a well-formed computation $C \in WF_P$ and a read operation $x \in Rd \cap V_C$, we use $lw_C(x)$ to denote the last write by the same processor to the same location. If no write to the same location precedes x in C, then $lw_C(x) = \text{NIL}$. The function lw_C is well-defined on the write operations of computations in WF_P. For convenience, we extend lw_C to V_C, where $lw_C(x) = \text{NIL}$ for $x \in V_C - Rd$.

Second, for a valid operator sequence α of $\mathcal{M}$, and $x, y \in elems(\alpha)$ such that $x \in Rd$ and $y \in Wr$, we define

$$bufretval(x, \alpha, y) = \begin{cases} retval(x, \alpha) & \text{if } y \leq_\alpha x \\ y.v & \text{otherwise.} \end{cases}$$

The intuition behind this definition is that y precedes x in the original computation, but not in the transformed computation because y may be buffered. However, if x is ordered before y, then the value that y would write is forwarded to x from the write buffer. For convenience, we define $bufretval(x, \alpha, \text{NIL}) = retval(x, \alpha)$; that is, an operation without a write whose value may be forwarded to it always returns the value determined by the schedule. Given a partial return value function ρ, we say that a schedule α ***explains*** ρ ***with read forwarding*** if $\rho(x) = bufretval(x, \alpha, ann_\alpha(x))$[4] for all $x \in domain(\rho)$.

For a memory with read forwarding but no caches, the underlying memory model is similar to sequential consistency, except that it uses *bufretval* instead of *retval* to determine the return values. A read operation in a computation of this memory is annotated by the write operation whose value may be forwarded to it, which is the last write operation to the same location by the same processor.

Formally, for a memory $M \in \mathfrak{M}_A^{\mathcal{M}}$ with read forwarding but no caches, the model of

[4]Recall that schedules of a computation are annotated, with each operation having the same annotation in the schedule as in the computation.

the underlying read forwarding memory, which has annotation set Wr, is

$$\begin{aligned} RdFd &= \left\{(C,\rho) \in \mathfrak{O}^{\mathcal{M}}_{Wr} : \exists \alpha \in Sch(C), \forall x \in V_C, \rho(x) = bufretval(x,\alpha,ann_C(x))\right\} \\ &= \{(C,\rho) : \exists \alpha \in Sch(C) \text{ that explains } \rho \text{ with read forwarding}\}. \end{aligned}$$

The reordering transformation $\mathcal{T}^{r}_{M}\colon \mathfrak{C}^{\mathcal{M}}_{A} \to \mathfrak{C}^{\mathcal{M}}_{Wr}$ determines the edges of the transformed computation as before, so $E_{\mathcal{T}^{r}_{M}(C)} = \{(x,y) \in E_C : Preserve_M(x,y,ann_C(x),ann_C(y))\}$, where $Preserve_M$ indicates whether the program order between two operations is preserved. The order between reads and prior writes on the same location should no longer be preserved. The annotation function of the transformed computation is $ann_{\mathcal{T}^{r}_{M}(C)} = lw_C$.

The SPARC family of processors uses this type of model.

Example 6.3 (Total Store Ordering) The *total store ordering (TSO)* model of the SPARC Version 8 [104] is similar to the IBM/370 model from Example 6.1, except that TSO has read forwarding and no fence operation.

Formally, we define $\mathcal{T}^{r}_{TSO}\colon \mathfrak{C}^{\mathcal{M}} \to \mathfrak{C}^{\mathcal{M}}_{Wr}$ using $Preserve_{TSO}(x,y,a_x,a_y) \equiv x \in Rd \vee y \in Wr$, and $ann_{\mathcal{T}^{r}_{TSO}(C)} = lw_C$. The table for $Preserve_{TSO}$ is

$Preserve_{TSO}$ y	read(ℓ')	write($\ell',\cdot$)
$x, a_x \quad a_y$	NIL	NIL
read(ℓ), NIL	$\surd$	$\surd$
write($\ell,\cdot$), NIL	$\circ$	$\surd$

The "$\circ$" symbol indicates that the operations may be reordered, but if the read overtakes a write to the same location, it is forwarded the value written by the last write by the same processor to that location, as captured by its annotation.

The memory model is $TSO = \mathcal{T}^{r}_{TSO}(RdFd) = \left\{(C,\rho) : (\mathcal{T}^{r}_{TSO}(C), \rho \in RdFd\right\}$. ■

Example 6.4 (SPARC V8 Partial Store Ordering) The *partial store ordering (PSO)* model relaxes total store ordering by allowing writes to overtake writes to different locations. It provides a *store barrier*, analogous to the write memory barrier of the Alpha from Example 6.2, to enforce ordering between writes to different locations. The store barrier is a noop annotated with WMB.

This memory model is $PSO = \mathcal{T}^{r}_{PSO}(RdFd)$, where $\mathcal{T}^{r}_{PSO}\colon \mathfrak{C}^{\mathcal{M}}_{\{WMB\}} \to \mathfrak{C}^{\mathcal{M}}_{Wr}$ is defined using

$$\begin{aligned} Preserve_{PSO}(x,y,a_x,a_y) \equiv\ & x \in Rd \vee (x \in Wr \wedge y \in Wr \wedge x.loc = y.loc) \\ & \vee (x \in Wr \wedge a_y = \text{WMB}) \vee (a_x = \text{WMB} \wedge y \in Wr), \end{aligned}$$

or in table form, using the symbology defined in the previous examples,

$Preserve_{PSO}$ y	read(ℓ')	write($\ell',\cdot$)	noop
$x, a_x \quad a_y$	NIL	NIL	WMB
read(ℓ), NIL	$\surd$	$\surd$	$\surd$
write($\ell,\cdot$), NIL	$\circ$	$\bullet$	$\surd$
noop, WMB	-	$\surd$	-

As defined here, the store barriers may be reordered; prohibiting the reordering of store barriers does not change the memory model. If a store barrier is placed between every pair of writes, *PSO* is exactly like *TSO*. ■

Example 6.5 (SPARC V9 Relaxed Memory Ordering) SPARC Version 9 [107] has an even more relaxed memory model, similar to the Alpha but with read forwarding. The *relaxed memory ordering (RMO)* model allows all read and write operations to be reordered except that writes cannot overtake reads or writes to the same location. To enforce desired ordering, it provides four different memory barriers, read-read, read-write, write-read, and write-write, where, for example, a read-write barrier cannot overtake reads or be overtaken by writes. We represent these barriers by noop operations annotated with FNC_{rr}, FNC_{rw}, FNC_{wr} and FNC_{ww} respectively.

Whether the barriers may be reordered with respect to each other is ambiguous. According the manual, they must be applied in program order, but other formal definitions allow them to be reordered [41, 100]. If the barriers may not be reordered, a read-read barrier followed by a write-write one also acts as a read-write barrier. Because of this implicit barrier, it seems more natural to allow barriers to be reordered.

Formally, we present the $Preserve_{RMO}$ predicate in a table, using the symbology defined in previous examples, and "?" to indicate the uncertainty of whether barriers may be reordered. $\text{FNC}_{\text{r}*}$ indicates an annotation of either FNC_{rr} or FNC_{rw}; $\text{FNC}_{\text{w}*}$, $\text{FNC}_{*\text{r}}$ and $\text{FNC}_{*\text{w}}$ are defined similarly.

$Preserve_{RMO}$ y	read(ℓ')	write($\ell', \cdot$)	noop	noop
x, a_x a_y	NIL	NIL	$\text{FNC}_{\text{r}*}$	$\text{FNC}_{\text{w}*}$
read(ℓ), NIL	-	•	√	-
write($\ell, \cdot$), NIL	∘	•	-	√
noop, $\text{FNC}_{*\text{r}}$	√	-	?	?
noop, $\text{FNC}_{*\text{w}}$	-	√	?	?

Once $Preserve_{RMO}$ is defined appropriately, this model is defined exactly as *TSO* and *PSO* above. ■

Example 6.6 (Commit-Reconcile) The *Commit-Reconcile (CR)* model [100] uses caches with explicit commit and reconcile operations, and background communication between each cache and the main memory. All reads and writes of a processor are done on the local cache. A commit operation ensures that a write has been propagated to main memory. A reconcile operation ensures that the cached value is not stale. Each location may be committed or reconciled independently. The commit and reconcile operations for a location ℓ are represented by noop operations annotated by $\text{COM}(\ell)$ and $\text{REC}(\ell)$ respectively. Thus, the annotation set is $A_{\text{CR}} = \{\text{COM}(\ell) : \ell \in \mathcal{L}\} \cup \{\text{REC}(\ell) : \ell \in \mathcal{L}\}$.

Although the Commit-Reconcile model is described in terms of caches, it can also be specified as a system with read forwarding. Informally, a write operation is "buffered" until its value, or the value written by a later operation to the same location, is sent back to main memory. A read operation moves earlier, to the last time the local cache and the main memory were made consistent. If a read operation is ordered before a previous write operation to the same location, it is "forwarded" the value last written to that location, which is in the cache.

Formally, $CR = \mathcal{T}^{\text{r}}_{CR}(RdFd)$, where $\mathcal{T}^{\text{r}}_{CR} \colon \mathfrak{C}^{\mathcal{M}}_{A_{\text{CR}}} \to \mathfrak{C}^{\mathcal{M}}_{Wr}$ is defined using the $Preserve_{CR}$ predicate described by the following table:

$Preserve_{CR}$ y	read(ℓ')	write($\ell', \cdot$)	noop	noop
x, a_x a_y	NIL	NIL	$\text{COM}(\ell')$	$\text{REC}(\ell')$
read(ℓ), NIL	•	√	√	√
write($\ell, \cdot$), NIL	∘	•	•	-
noop, $\text{COM}(\ell)$	-	√	√	√
noop, $\text{REC}(\ell)$	•	√	√	√

■

Example 6.7 (Commit-Reconcile & Fences) The *Commit-Reconcile & Fences (CRF)* model [100] takes the Commit-Reconcile model and, like the relaxed memory ordering model, allows almost all operations to be reordered. Like RMO, it uses fences to enforce the ordering when desired; unlike RMO,

the fences in CRF are fine-grained: A fence specifies two locations, the "pre-location" and "post-location", and maintains the order only between reads and writes to those locations. We extend the annotation sets of *RMO* and *CR* to accommodate these operations. For example, we represent a read-write fence with pre-location ℓ and post-location ℓ' by a noop operation with annotation $\text{FNC}_{\text{rw}}(\ell, \ell')$. The resulting annotation set is denoted by A_{CRF}.

Because a read operation effectively occurs as early as the previous reconcile operation, the write-read and read-read fences maintain their order relative to subsequent reconcile operations, rather than read operations. Similarly, a write effectively occurs as late as the next commit operation, so the write-read and write-write fences maintain their order relative to prior commit operations.

Formally, $CRF = \mathcal{T}^{r}_{CRF}(RdFd)$, where $\mathcal{T}^{r}_{CRF} \colon \mathfrak{C}^{\mathcal{M}}_{A_{\text{CRF}}} \to \mathfrak{C}^{\mathcal{M}}_{Wr}$ is defined using the $Preserve_{CRF}$ predicate described by the following table:[5]

$Preserve_{CRF}$ y	read(ℓ')	write$(\ell', \cdot)$	noop	noop	noop	noop
x, a_x a_y	NIL	NIL	$\text{COM}(\ell')$	$\text{REC}(\ell')$	$\text{FNC}_{r*}(\ell', \cdot)$	$\text{FNC}_{w*}(\ell', \cdot)$
read(ℓ), NIL	-	•	-	-	•	-
write$(\ell, \cdot)$, NIL	∘	•	•	-	-	-
noop, $\text{COM}(\ell)$	-	-	-	-	-	•
noop, $\text{REC}(\ell)$	•	-	-	-	-	-
noop, $\text{FNC}_{*r}(\cdot, \ell)$	-	-	-	•	-	-
noop, $\text{FNC}_{*w}(\cdot, \ell)$	-	•	-	-	-	-

■

Caching. The final models we discuss in this section expose the programmer to caching, where each processor maintains a local copy of the memory. Every operation is applied to each local copy, but the processors may schedule the operations differently. Often there is a global consistency requirement on the schedules, that some operations are ordered consistently by all processors. As usual, the local copies of the memory and the schedules at each processor are abstract: The system must appear as though each processor maintains a local copy on which it applies the operations according to the schedule, but there need not be any direct representation of this value in the implementation.

Formally, we model a basic caching memory by:

$$Cache = \{(C, \rho) : \forall p \in \mathcal{P}, \exists \alpha \in Sch(C), \alpha \text{ explains } \rho|_p\}$$

The global consistency requirement is modeled using synchronizing transformations. A synchronizing transformation adds edges to a computation between operations that synchronize, as specified by a synchronization predicate. The formal definitions of synchronizing transformations and synchronization predicates are given in Section 5.5.

[5]The model defined here is slightly weaker than the original model in a way analogous to the way RMO has been relaxed from its original definition (see Example 6.5): A sequence of operations can inhibit reordering in the original model in a way that none of the operations in the sequence inhibit by themselves. To get the original model, first use $\mathcal{T}^{r}_{CRF}$, except that read operations do not overtake write operations to the same location. Then take the transitive closure of the resulting computation, and execute it on *CR*. In the model defined here, the transitive closure is not taken between the explicit reordering in *CRF* (which is inhibited by the fences) and the reordering in *CR* due to caching.

Example 6.8 (Pipelined RAM) The *pipelined RAM (pRAM)* model [74] does not reorder any operations and has no global consistency requirement. Thus, $pRAM = Cache$. ■

Most multiprocessors guarantee coherence by requiring all processors to schedule the writes to a single location consistently; the writes to that location appear in the same order in the schedules of each of the processors. This property is called *write serialization* in the literature [26, 52]. We investigate the relationship between write serialization and coherence in Section 6.3.

Formally, we use a synchronizing transformation defined in Section 5.5 to model write serialization. The synchronization predicate is

$$\begin{aligned}\Psi_{\text{ws}}(x, y, C) &\equiv x, y \in Wr \wedge x.loc = y.loc \\ &\equiv \exists \ell \in \mathcal{L}, x, y \in Wr|_\ell.\end{aligned}$$

Example 6.9 (Processor Consistency) *Processor consistency* is an intermediate model between coherence and sequential consistency [47]. Like the pipelined RAM model, it allows each processor to schedule the operations differently and does not allow any reordering. However, unlike pipelined RAM, processor consistency requires write serializaton. Thus, $PC = \mathcal{T}^{\text{s}}_{\Psi_{\text{ws}}}(Cache)$.

The original definition was informal, and different formal interpretations were developed [12, 8]. We follow the interpretation of Ahamad, et al. A rather different definition was used in the DASH system developed at Stanford [43, 41]. ■

Memories with caches can also reorder operations, which we model as we did for memories without caches, using reordering transformations.

Example 6.10 (Weak Ordering) In *weak ordering* [33, 32], memory operations may be reordered unless they access the same variable or are designated as *synchronization operations*, indicated by the annotation SYNC. The schedules at each processor must have the same order for all synchronization operations and for other operations with respect to synchronization operations.[6] Finally, the memory is implicitly assumed to maintain write serialization.

Formally, let $WO = \mathcal{T}^{\text{r}}_{WO}(\mathcal{T}^{\text{s}}_{\Psi_{\text{S}}}(\mathcal{T}^{\text{s}}_{\Psi_{\text{ws}}}(Cache))) = \mathcal{T}^{\text{r}}_{WO}(\mathcal{T}^{\text{s}}_{\Psi_{\text{S}}}(PC))$, where Ψ_{S} is the strong synchronization predicate defined in Example 5.2, and $\mathcal{T}^{\text{r}}_{WO}\colon \mathfrak{C}^{\mathcal{M}}_{\{\text{SYNC}\}} \to \mathfrak{C}^{\mathcal{M}}_{\{\text{SYNC}\}}$ is defined using

$$Preserve_{WO}(x, y, a_x, a_y) \equiv a_x = \text{SYNC} \vee a_y = \text{SYNC} \vee x.loc = y.loc,$$

or in table form, using the symbology defined in Example 6.1,

$Preserve_{WO}$ y / x, a_x a_y	read(ℓ') NIL	write($\ell', \cdot$) NIL	read(ℓ') SYNC	write($\ell', \cdot$) SYNC
read(ℓ), NIL	•	•	√	√
write($\ell, \cdot$), NIL	•	•	√	√
read(ℓ), SYNC	√	√	√	√
write($\ell, \cdot$), SYNC	√	√	√	√

■

For a memory with caching and read forwarding, the underlying memory model combines *Cache* and *RdFd*. Its annotation set is *Wr*, and each processor has a schedule that

[6] Adve notes that it is sufficient to maintain the order between conflicting operations [1].

explains, with read forwarding, the values returned for its operations.

$$RdFdCache = \{(C, \rho) : \forall p \in \mathcal{P}, \exists \alpha \in Sch(C), \alpha \text{ explains } \rho|_p \text{ with read forwarding}\}$$

Global consistency requirements and reordering are modeled as before.

Example 6.11 (Release Consistency) *Release consistency* was developed to model the guarantees of the Stanford DASH machine [71, 43]. As with weak ordering, computations of this memory distinguish synchronization operations with the annotation SYNC. Any operation may be labeled as a synchronization operation; other operations are *ordinary*. A synchronization read operation is called an *acquire*, and a synchronization write operation is called a *release*.

A release consistent memory guarantees sequential consistency for synchronization operations; it does not reorder them. Ordinary operations may be reordered, except that writes may not overtake operations to the same location. In addition, ordinary operations may not overtake acquires, nor may they be overtaken by releases.

Formally, $RC = \mathcal{T}^{r}_{RC}(\mathcal{T}^{s}_{\Psi_W \vee \Psi_{ws}}(RdFdCache))$, where $\mathcal{T}^{r}_{RC}$: $\mathfrak{C}^{\mathcal{M}}_{A} \to \mathfrak{C}^{\mathcal{M}}_{Wr}$ defines the edges using

$$Preserve_{RC}(x, y, a_x, a_y) \equiv a_x = a_y = \text{SYNC} \vee (x \in Rd \wedge a_x = \text{SYNC}) \vee (y \in Wr \wedge a_y = \text{SYNC}) \vee (y \in Wr \wedge x.loc = y.loc),$$

and the annotations using lw_C, and Ψ_W is the weak synchronization predicate from Example 5.1. The table describing $Preserve_{RC}$ is

$Preserve_{RC}$ x, a_x \ y, a_y	read(ℓ') NIL	write($\ell', \cdot$) NIL	read(ℓ') SYNC	write($\ell', \cdot$) SYNC
read(ℓ), NIL	-	•	-	√
write($\ell, \cdot$), NIL	∘	•	∘	√
read(ℓ), SYNC	√	√	√	√
write($\ell, \cdot$), SYNC	∘	•	√	√

The original definition guarantees only processor consistency for the synchronization operations, using a different notion of processor consistency than the one defined in Example 6.9. It also provides another label NSYNC for operations that compete with other operations but are not used for synchronization. For simplicity, we omit this extra label in our definition.[7]

Although release consistency serializes writes—Gharachorloo calls this requirement the *coherence requirement* [41]—it is not coherent. We demonstrate and discuss this distinction in Section 6.3. ■

6.3 Write Serialization and Coherence

In the literature, write serialization is frequently used synonymously with coherence, or is assumed to guarantee coherence [41, 52, 26]. As long as a processor-centric memory preserves the order between operations to the same location, this assumption is justified. However, if operations to the same location may be reordered, a write serializing memory

[7] This omission does eliminate some optimization opportunities, which motivate the *data-race-free-1* and *properly-labeled-2* programmer-centric models [1, 41]. We can extend our definition to handle this additional distinction: NSYNC operations are treated as SYNC operations except that they may be reordered with respect to the ordinary operations annotated by NIL.

might not be coherent. In particular, release consistency does not imply coherence. In this section, we examine the relationship between write serialization and coherence.

Write serialization without reordering—that is, processor consistency—implies coherence.

Lemma 6.1 *PC* implements *Coh* under WF_P.

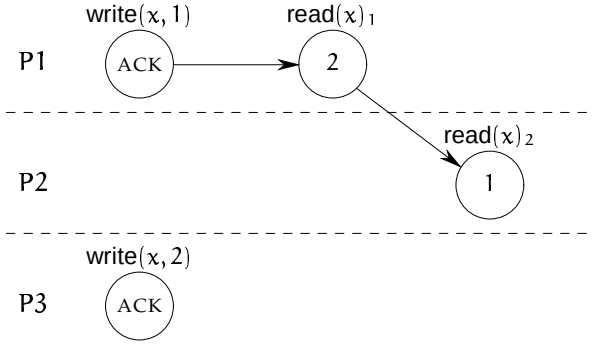
Proof: If $(C, \rho) \in PC = \mathcal{T}^s_{\Psi_{ws}}(Cache)$, then there exists $C' \in \mathcal{T}^s_{\Psi_{ws}}[C]$ such that $(C', \rho) \in Cache$. Let $\beta_{p,\ell}$ be the serialization at processor p of all write and local read operations to ℓ; that is, $\beta_{p,\ell} = \alpha_p|_{(Wr|_\ell \cup Rd|_\ell|_p)}$. By Lemma 2.21, $retval(x, \alpha_p) = retval(x, \beta_{p,\ell})$ for $x \in elems(\beta_{p,\ell})$.

Let $\prec_\ell = TC(\bigcup_{p\in\mathcal{P}} <_{\beta_{p,\ell}})$. We know that $\prec_\ell$ is a partial order on $V_C|_\ell$ because $x <_{\gamma|_{Wr|_\ell}} y$ implies $x <_{\beta_{p,\ell}} y$ for all $p \in \mathcal{P}$, and read operations are only ordered by the sequence of the processor that requested them. We show that $\prec_\ell$ and $\prec_C$ are consistent. Assume for contradiction that they are not consistent. Then there is a cycle $x_1 \prec_\ell x_2 \prec_C \cdots \prec_\ell x_{2k} \prec_C x_{2k+1} = x_1$ of $\prec_\ell \cup \prec_C$. Because $x_{2i-1} \prec_\ell x_{2i}$ for all $i = 1, \ldots, k$, we have $x_i \in V_C|_\ell$ for all $i = 1, \ldots, 2k$. Since $C \in WF_P$ and $x_{2i} \prec_C x_{2i+1}$ for all $i = 1, \ldots, k$, we have $p(x_{2i}) = p(x_{2i+1})$. Thus, $x_{2i} \prec_\ell x_{2i+1}$ for all $i = 1, \ldots, k$, so $\prec_\ell$ has a cycle, which contradicts the fact that $\prec_\ell$ is a partial order.

Let $C_\ell = (V_C, E_C \cup \prec_\ell)$ and $\alpha_\ell \in Sch(C_\ell) \subseteq Sch(C)$. By Lemma 2.21, for $x \in V_C|_\ell$, $retval(x, \alpha_\ell) = retval(x, \alpha_\ell|_\ell) = retval(x, \beta_{p(x),\ell}) = retval(x, \alpha_{p(x)}) = \rho(x)$. Because this is true for all $\ell \in \mathcal{L}$, we have $(C, \rho) \in Coh$. ■

The previous lemma does not hold for computations in which there are dependencies between operations at different processors.

Example 6.12 Consider the following observation, in which the return value for each operation is given in the center of the circle. It is not coherent because any coherent memory must return 2 for $\mathsf{read}(x)_2$ whenever the value returned for $\mathsf{read}(x)_1$ is 2. However, *PC* admits this observation.



■

Even if the underlying memory is sequentially consistent, a system that may arbitrarily reorder operations on different locations is at best coherent. Formally, let $\mathcal{T}^r_{Coh}$: $\mathfrak{C}^{\mathcal{D}}_A \to \mathfrak{C}^{\mathcal{D}}_{A'}$, be the reordering transformation defined using $Preserve_{Coh}(x, y, a_x, a_y) \equiv x.loc = y.loc$, which allows operations on different locations to be reordered. A coherent memory implements sequential consistency when this transformation is used.

Lemma 6.2 *Coh* implements $\mathcal{T}^r_{Coh}(SC)$.

Proof: Let $\{\ell_1, \ell_2, \ldots, \ell_n\} = \{x.loc : x \in V_C\}$. If $(C, \rho) \in Coh$ then for all $i = 1, \ldots, n$, there exists $\alpha_i \in Sch(C)$ that explains $\rho|_{\ell_i}$. Let $\beta = \alpha_1|_{\ell_1} \cdot \alpha_2|_{\ell_2} \cdot \cdots \cdot \alpha_n|_{\ell_n}$. By Corollary 2.22, $\beta|_\ell = \alpha_\ell|_\ell$ explains

$\rho|_\ell$ for all $\ell \in \mathcal{L}$, so β explains ρ. Also, $\beta \in Sch(\mathcal{T}^{r}_{Coh}(C))$ since if $(x,y) \in E_{\mathcal{T}^{r}_{Coh}(C)}$ then $(x,y) \in E_C$ and $x.loc = y.loc$, so $x <_{\alpha_{x.loc}|_{x.loc}} y$, and thus, $x <_\beta y$. So $(\mathcal{T}^{r}_{Coh}(C), \rho) \in SC$, and $(C, \rho) \in \mathcal{T}^{r}_{Coh}(SC)$. ■

Consider a write serializing memory that may arbitrarily reorder operations on different locations. Formally, let $WS = \mathcal{T}^{r}_{Coh}(\mathcal{T}^{s}_{\Psi_{ws}}(Cache)) = \mathcal{T}^{r}_{Coh}(PC)$. In the processor-centric view, this memory is exactly coherent memory.

Theorem 6.3 *WS* is equivalent to *Coh* under WF_P.

Proof: If $(C, \rho) \in WS = \mathcal{T}^{r}_{Coh}(\mathcal{T}^{s}_{\Psi_{ws}}(Cache))$ for $C \in WF_P$ then $(\mathcal{T}^{r}_{Coh}(C), \rho) \in \mathcal{T}^{s}_{\Psi_{ws}}(Cache) \subseteq Coh$, by Lemma 6.1. So for each $\ell \in \mathcal{L}$, there exists $\alpha \in Sch(\mathcal{T}^{r}_{Coh}(C))$ that explains $\rho|_\ell$. Suppose $x.loc = y.loc = \ell$. Since $C \in WF_P$, $x \prec_C y$ implies $(x,y) \in E_C$, so $(x,y) \in E_{\mathcal{T}^{r}_{Coh}(C)}$, and $x <_{\alpha|_\ell} y$. Thus, $\alpha|_\ell$ is consistent with $\prec_C$, and so there exists $\beta \in Sch(C)$ with $\beta|_\ell = \alpha|_\ell$, which by Corollary 2.22, explains $\rho|_\ell$. So $(C, \rho) \in Coh$.

By Lemma 6.2, *Coh* implements $\mathcal{T}^{r}_{Coh}(SC)$. By Lemma 4.6, $\mathcal{T}^{r}_{Coh}(SC)$ implements $\mathcal{T}^{r}_{Coh}(\mathcal{T}^{s}_{\Psi_{ws}}(Cache))$ since *SC* implements $\mathcal{T}^{s}_{\Psi_{ws}}(Cache)$. Thus, *Coh* implements $\mathcal{T}^{r}_{Coh}(\mathcal{T}^{s}_{\Psi_{ws}}(Cache)) = WS$. ■

Using a different formalism, Higham, Kawash and Verwaal prove similar results [55], including the following corollary, which says that coherence is the same as sequential consistency if operations on different locations can be reordered.

Corollary 6.4 $\mathcal{T}^{r}_{Coh}(SC)$ is equivalent to *Coh* under WF_P.

Proof: By Lemma 4.3 and 6.2, *Coh* implements $\mathcal{T}^{r}_{Coh}(SC)$ under WF_P. By Lemma 4.6, $\mathcal{T}^{r}_{Coh}(SC)$ implements $\mathcal{T}^{r}_{Coh}(PC) = WS$, which, by Theorem 6.3, is equivalent to *Coh* under WF_P. ■

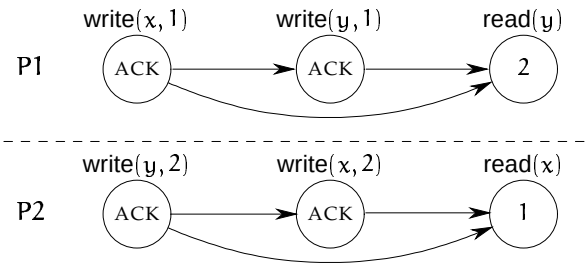
We can also derive immediately that weak ordering is coherent.

Corollary 6.5 *WO* implements *Coh* under WF_P.[8]

Proof: Immediate from Theorem 6.3 because $WO = \mathcal{T}^{r}_{WO}(\mathcal{T}^{s}_{\Psi_s}(PC)) \subseteq \mathcal{T}^{r}_{WO}(PC) \subseteq \mathcal{T}^{r}_{Coh}(PC) = WS$. ■

Theorem 6.3 establishes the close relationship between write serialization and coherence. However, requiring a caching memory to serialize writes is not the same as insisting that the memory be coherent. Serializing writes may inhibit observations of the original system that were admissible according to coherence.

Example 6.13 Ahamad, et al. [8] point out that although processor consistency is pipelined RAM with write serialization, $pRAM \cap Coh$ is not equivalent to *PC*. The following observation, for example, is admissible according to both *pRAM* and *Coh*, but not according to *PC*:

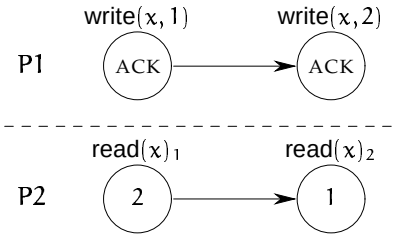


[8] In fact, $WO = \mathcal{T}^{r}_{WO}(SC)$, but proving this fact would be a further digression on what is already a digression.

Of course, *PC* implements both $pRAM = Cache$ and $\mathcal{T}^{r}_{Coh}(\mathcal{T}^{s}_{\Psi_{ws}}(Cache))$, which is equivalent to *Coh* under WF_P, so *PC* is strictly stronger than $pRAM \cap Coh$ under WF_P. ■

Conversely, a system that reorders operations on the same location may not guarantee coherence even if it serializes writes. In particular, although release consistency has a "coherence requirement" [41], it is not coherent. To our knowledge, this fact has not been noted in the literature.

Example 6.14 The following observation is release consistent but not coherent.



This observation is also admitted by *RMO* and *CRF*, so they too are not coherent. ■

Although release consistency admits incoherent observations, the systems it was designed to model probably never exhibited them, as there is no reason for a read operation to return a value already known to be stale. Thus, these systems may indeed have been coherent.

The extra flexibility provided by allowing incoherent observations is helpful primarily for the compiler: Requiring operations on the same location to occur in order inhibits the reordering of any operations to variables that may be aliased to the same location.[9] Proving that variables are never aliased is difficult—often it is not even true—so this requirement significantly restricts the possible compiler transformations.

6.4 Comparisons Between Processor-Centric Models

Strictly speaking, most of the memory models defined in Section 6.2 are incomparable because they have different mechanisms for synchronization. However, the bulk of the instructions issued by the clients to the memory are typically "ordinary" reads and writes, with the minimal synchronization allowed by the model. We compare stripped-down versions of these models, which only handle ordinary operations. We also compare them with processor-centric versions of sequential consistency and coherence. The relationships between the stripped-down, or *baseline*, versions of all these models are summarized in Figure 6-1. This section formally defines the baseline models and sketches proofs for these relationships. It can be skipped without loss of continuity.

[9] I am indebted to Xiaowei Shen for this observation [98]. It was also made by Pugh in his analysis of the Java memory model [93].

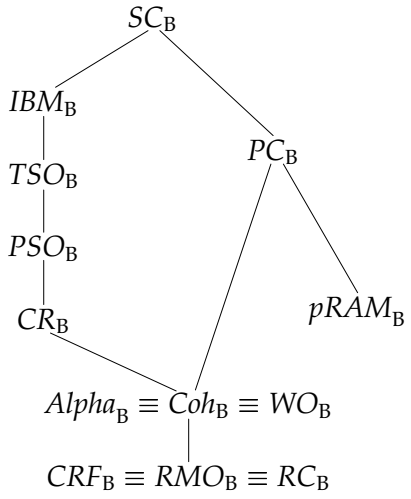


Figure 6-1: Haase diagram of the relative strength of the baseline processor-centric models discussed in this chapter. Stronger models are on top.

The ***baseline processor-centric version*** of a memory model M, denoted M_B, restricts the well-formed computations to operations without annotations. Formally,

$$M_B = \{(C, \rho) \in M : C \in WF_P \wedge \forall x \in V_C, ann_C(x) = \text{NIL}\}$$

A sequentially consistent memory synchronizes even the ordinary operations; a generic memory provides no synchronization at all. The models defined in Section 6.2 lie between these two extremes, ranging from the IBM/370, which is almost sequentially consistent, to the relaxed memory ordering and release consistency models, which are very weak. The memory models compared in this section are characterized in Figure 6-2 by the reordering each model allows and the guarantees of its underlying memory. These characterizations are derived directly from the definitions of these models, and from the characterization of coherence in Corollary 6.4. The relative strength of many of these models follows directly from these characterizations, and results we proved earlier.

Lemma 6.6 The following relationships hold:

- $SC_B \subseteq IBM_B \subseteq Alpha_B \equiv Coh_B \equiv WO_B \subseteq RC_B$.
- $SC_B \subseteq PC_B \subseteq pRAM_B$.
- $IBM_B \subseteq TSO_B \subseteq PSO_B \subseteq CR_B \subseteq RMO_B \equiv CRF_B \subseteq RC_B$.
- $PC_B \subseteq Coh_B \subseteq RMO_B$.

Proof Sketch: $PC_B \subseteq Coh_B$ follows from Lemma 6.1 and $WO_B \subseteq Coh_B$ follows from Corollary 6.5. All the other relationships follow straightforwardly from the tables in Figure 6-2, and the facts that $SC \subseteq \mathcal{T}^s_{\Psi_{ws}}(Cache) \subseteq \mathcal{T}^s_{\Psi_{ws}}(RdFdCache)$, $SC \subseteq \mathcal{T}^s_{\Psi_{ws}}(Cache) \subseteq Cache$, and $SC \subseteq RdFd \subseteq RdFdCache$. ■

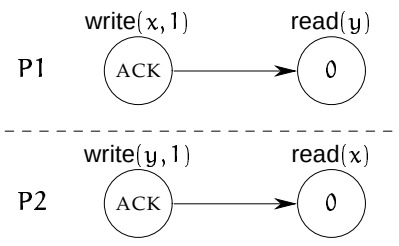
$Preserve_M$	read(ℓ')	write($\ell', \cdot$)	read(ℓ')	write($\ell', \cdot$)	read(ℓ')	write($\ell', \cdot$)
M (M')	SC_B (SC)		TSO_B ($RdFd$)		$pRAM_B$ ($Cache$)	
read(ℓ)	√	√	√	√	√	√
write($\ell, \cdot$)	√	√	∘	√	√	√
M (M')	IBM_B (SC)		PSO_B ($RdFd$)		PC_B ($\mathcal{T}^{s}_{\Psi_{ws}}(Cache)$)	
read(ℓ)	√	√	√	√	√	√
write($\ell, \cdot$)	•	√	∘	•	√	√
M (M')	$Alpha_B$ (SC)		RMO_B ($RdFd$)		WO_B ($\mathcal{T}^{s}_{\Psi_{ws}}(Cache)$)	
read(ℓ)	•	•	-	•	•	•
write($\ell, \cdot$)	•	•	∘	•	•	•
M (M')	Coh_B (SC)		CR_B ($RdFd$)		RC_B ($\mathcal{T}^{s}_{\Psi_{ws}}(RdFdCache)$)	
read(ℓ)	•	•	•	√	-	•
write($\ell, \cdot$)	•	•	∘	•	∘	•
M (M')			CRF_B ($RdFd$)			
read(ℓ)			-	•		
write($\ell, \cdot$)			∘	•		

Symbology:
√ = order between operations is always preserved
• = order is preserved when the operations access the same location
∘ = order is not preserved, but if the read overtakes a write to the same location, it is forwarded the value written
- = order is not preserved

Figure 6-2: Tables defining the baseline processor-centric models. The underlying memory for each model is in parentheses. Each box contains the relevant part of the reordering tables used to define the full versions of the memory models in Section 6.2.

Except for $CRF_B \subseteq RC_B$, the relationships in Lemma 6.6 are strict. ¿From Example 6.13, we know that PC_B is strictly stronger than Coh_B and $pRAM_B$. ¿From Example 6.14, we know that Coh_B is strictly stronger than RC_B, RMO_B and CRF_B, and also that CR_B is strictly stronger than CRF_B since that observation is not admissible according to CR. The following examples show that SC_B is strictly stronger than both IBM_B and PC_B, and that IBM_B, TSO_B, PSO_B and CR_B are all different.

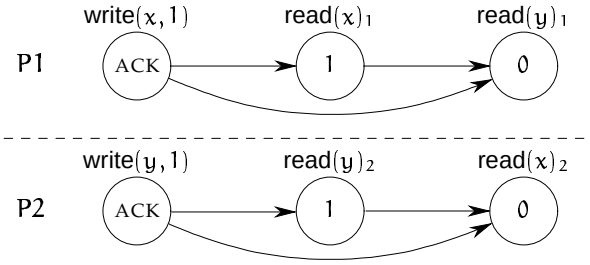
Example 6.15 The following observation is admissible according to PC_B and IBM_B but not SC_B.



In any sequentially consistent observation, at least one of the read operations must return 1. PC_B admits this observation because each processor can execute its local operations first. IBM_B admits

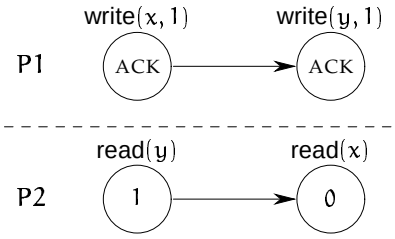
it because the read operations may be reordered before the write operations. Thus, SC_B is strictly stronger than both PC_B and IBM_B. ■

Example 6.16 The following observation is admissible according to TSO_B and PC_B but not IBM_B.



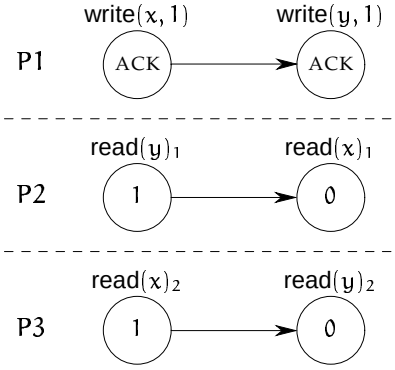
This observation is exactly the same as the observation from Example 6.15 except that x and y are read immediately after they are written. It is not admitted by SC_B and admitted by PC_B for the same reasons. It is not admitted by IBM_B, because the intervening read operations inhibit reordering. However, TSO_B allows the reordering, but forwards the values written to the read operations, so it does admit this observation. Thus, IBM_B is strictly stronger than TSO_B (and Coh_B, because $PC_B \subseteq Coh_B$), and PC_B is not stronger than IBM_B. ■

Example 6.17 The following observation is admissible according to PSO_B but not TSO_B.



TSO_B does not admit this observation because it does not allow any of the operations to be reordered. PSO_B however, may reorder the two write operations because they are on different locations, so it does admit this observation. Thus, TSO_B is strictly stronger than PSO_B. ■

Example 6.18 The following observation is admissible according to CR_B but not PSO_B.



This observation, which is the same as the observation from Example 6.17 except for additional operations for P3, is not admitted by PSO_B because the return values for the operations of P2 imply that write$(y,1)$ is scheduled before write$(x,1)$ while those of P3 imply that write$(x,1)$ is scheduled before write$(y,1)$. CR_B may reorder the read operations, as they are on different locations, so it admits this observation. Thus, PSO_B is strictly stronger than CR_B. ■

Without synchronization, release consistency and CRF are equivalent.

Lemma 6.7 $RC_B \subseteq CRF_B$ (which implies $RC_B \equiv CRF_B$).

Proof Sketch: For any observation of RC_B, there is a local schedule for each processor that explains the values returned for the read operations of that processor with read forwarding. The order of the write operations on each location is the same in all the local schedules. Construct a global schedule with the same order for the write operations on each location, and each read operation inserted according to the local schedule of its processor. This construction yields a schedule because in the reordered transformation, there are no precedence dependencies between operations on different locations or from different processors. This schedule explains all the return values with read forwarding because each read operation gets the same value as in its local schedule. Thus, $RC_B \subseteq CRF_B$. ■

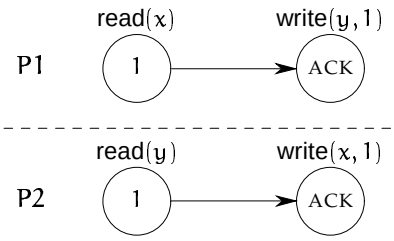
We can also show that *TSO*, *PSO* and *CR* are coherent. It is sufficient to show that *CR* is coherent, since *TSO* and *PSO* are stronger than *CR*.

Lemma 6.8 $CR_B \subseteq Coh_B$.

Proof Sketch: For any observation $(C,\rho) \in CR$, there is some schedule $\alpha \in Sch(\mathcal{T}^r_{CR}(C))$ such that α explains ρ with read forwarding. Except for those read operations that are forwarded values from write operations they overtake, α explains ρ (without read forwarding). Move those read operations backwards until after the write whose value they are forwarded. This construction yields a schedule of $\mathcal{T}^r_{Coh}(C)$ because the write operation preceded the read operation in C. So $(\mathcal{T}^r_{Coh}(C),\rho) \in SC$, and $(C,\rho) \in Coh$. ■

The preceding lemmas and examples establish all the implementation and equivalence relationships in Figure 6-1. To complete our analysis, we verify that models for which we give no relationship are incomparable. Specifically, we use the following examples to show that PC_B and $pRAM_B$ are incomparable with IBM_B, TSO_B, PSO_B and CR_B, and that $pRAM_B$ is incomparable with RC_B.

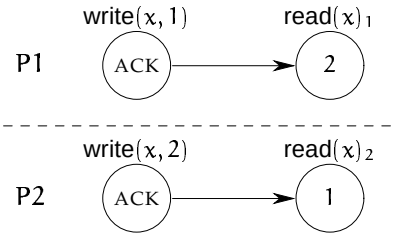
Example 6.19 The following observation is admissible according to PC_B (and Coh_B) but not CR_B.



This observation is the reverse of the one from Example 6.15. PC_B admits it because P1 may schedule the operations of P2 before its own while P2 schedules the operations of P1 before its own; that is, they execute the remote operations first. It is not admissible according to CR_B because none of

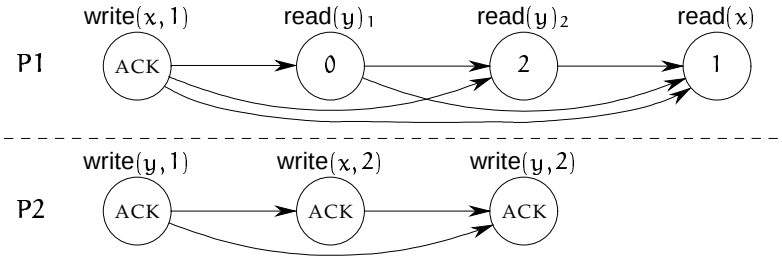
its operations may be reordered by CR_B, and there is no single schedule that explains the return values. Thus, PC_B is not stronger than CR_B, nor is it stronger than PSO_B, TSO_B or IBM_B, which are all stronger than CR_B. This also shows that CR_B is strictly stronger than Coh_B. ■

Example 6.20 The following observation is admissible according to $pRAM_B$ but not RC_B.



$pRAM_B$ admits this observation because each processor may schedule the remote operations before its local ones. RC_B does not admit it because it requires both processors to schedule the write operations consistently; the read operations may not overtake the write operations because they return values written by remote operations. Thus, $pRAM_B$ is not stronger than RC_B. ■

Example 6.21 The following observation, adapted from an example due to Kawash [62], is admissible according to IBM_B but not $pRAM_B$.



This is not admissible according to $pRAM_B$ because no schedule explains the return values of P1. It is admissible according to IBM_B because the read$(y)_1$ can be reordered before write$(x, 1)$, and all the operations of P2 can be executed after read$(y)_1$ and before any of the other operations of P1. Thus, IBM_B is not stronger than $pRAM_B$, nor is it stronger than PC_B. This also implies than none of TSO_B, PSO_B, CR_B, Coh_B and RC_B are stronger than either $pRAM_B$ or PC_B. ■

6.5 A Possible Pitfall with Reordering

Often we define a memory model by giving a schedule that explains the return value of each operation. Such a definition guarantees that the return value function the memory model associates with any computation is an observer function, as required by the definition of memory models. However, a reordering transformation eliminates some of the precedence dependencies, allowing schedules of the transformed computation that are not schedules of the original computation. In this section, we show why the return value functions associated by the memory models defined in this chapter are observer functions, even though the serializations used in their definitions to explain their return values may not be schedules.

As a concrete example of the potential problem, consider a memory that may reorder reads before writes to the same location. Suppose a processor issues two writes and then a read to the same location. If the system reorders the read before the second write, then the read returns the value written by the first write, which is not possible in any schedule of the original computation. We get a similar result if we reorder the two write operations.

This problem does not arise in systems that do not reorder operations to the same location. The return value for each operation in the schedule of the transformed computation is explained by some schedule of the original computation.

Lemma 6.9 Suppose $\{O_\ell\}_{\ell\in\mathcal{L}}$ is a location partition for $\mathcal{D}$, $\ell \in \mathcal{L}$, $C \in \mathfrak{C}_A^{\mathcal{D}}$ and $C' \in \mathfrak{C}_{A'}^{\mathcal{D}}$, such that $V_C = V_{C'}$ and $x \prec_C y$ implies $x \prec_{C'} y$ whenever $x.loc = y.loc = \ell$. For any $\alpha' \in Sch(C')$ and $x \in V_C|_\ell$, there exists $\alpha \in Sch(C)$ such that $retval(x, \alpha) = retval(x, \alpha')$.

Proof: Since $\alpha' \in Sch(C')$, $\alpha'|_\ell$ is consistent with $\prec_{C'}$. By Lemma 2.1, $\alpha'|_\ell$ is consistent with $\prec_C$ since $\prec_C|_\ell \subseteq \prec_{C'}|_\ell$. Thus, there is some topological sort α of $<_{\alpha'|_\ell} \cup \prec_C$, so by Corollary 2.22, $retval(x, \alpha) = retval(x, \alpha|_\ell) = retval(x, \alpha'|_\ell) = retval(x, \alpha')$. ■

Of the memories defined in the previous section, those without read forwarding preserve the order between operations on the same location. It follows from the previous lemma, that they admit only observer functions for well-formed computations.

Lemma 6.10 Suppose $\{O_\ell\}_{\ell\in\mathcal{L}}$ is a location partition for $\mathcal{D}$ and $M' \in \mathfrak{M}_{A'}^{\mathcal{D}}$. If $M = \mathcal{T}^r(M')$, where $\mathcal{T}^r\colon \mathfrak{C}_A^{\mathcal{D}} \to \mathfrak{C}_{A'}^{\mathcal{D}}$ is defined by *Preserve* and $x.loc = y.loc \implies Preserve(x, y, a_x, a_y)$, then $M|_{WF_P} \in \mathfrak{M}_A^{\mathcal{D}}$.

Proof: If $(C, \rho) \in M|_{WF_P}$ then $(\mathcal{T}^r(C), \rho) \in M'$. Because $C \in WF_P$, for all $x, y \in V_C$, if $x \prec_C y$ then $(x, y) \in E_C$, and if $x.loc = y.loc$, then $Preserve(x, y, ann(x), ann(y))$, so $(x, y) \in E_{\mathcal{T}^r(C)}$. Because $(\mathcal{T}^r(C), \rho) \in M' \in \mathfrak{M}_{A'}^{\mathcal{D}}$, for all $x \in V_C$, there is some schedule $\alpha' \in Sch(\mathcal{T}^r(C))$ such that $\rho(x) = retval(x, \alpha')$. By Lemma 6.9, there exists $\alpha \in Sch(C)$ such that $retval(x, \alpha) = retval(x, \alpha')$. Thus, for all $x \in V_C$, $\rho(x) = retval(x, \alpha)$ for some $\alpha \in Sch(C)$, as required. ■

For the models with read forwarding, reads may overtake writes to the same location, so the argument above does not hold. However, if a read overtakes a write to the same location, the memory forwards the value written by the last write requested by the same processor. This value is explained by any schedule that does not interleave the operations of different processors.

The lemmas above hold for a memory with any location-partitioned data type, not just read/write memory. For read/write memory, we could also allow two reads to the same location to be reordered. We might guess that the results can be generalized to data types without location partitions. However, at least the simplest generalization, reordering only independent operations, does not work.

Example 6.22 Consider the following program for a read/write memory with an additional swap operation, which swaps the values of two locations:

```
int x,y = 0

P1: write(x,1)      P2: swap(x,y)
    write(y,2)
    a <- read(y)
```

If the operations may not be reordered, then no schedule allows an observation with `a = 0`: If the swap occurs between the write$(y, 2)$ and the read(y), then we get `a = 1`; otherwise, `a = 2`. However, if we can reorder the write$(x, 1)$ and write$(y, 2)$ operations, then the schedule

$$\text{write}(y, 2), \text{swap}(x, y), \text{write}(x, 1), \text{read}(y)$$

yields `a = 0`. ■

6.6 Interpreting Reordering in Processor-Centric Models

Thus far, we have followed most of the literature in viewing reordering as a way to describe the weak guarantees of a memory system. In this section, we introduce an alternative view, which considers reordering as a relaxation of the program order. That is, reordering is a way to express concurrency using a sequential language. By moving reordering from memory to the clients, this view simplifies the memory semantics.

In the conventional view, reordering operations is a formal mechanism to describe the semantics of weak consistency guarantees. The program order is a total order at each processor and the system must respect that order. To achieve better performance, the values returned for different operations, especially operations of different processors, may not be consistent. The relaxation in the guarantees may be due to explicit reordering in the system, by the compiler or the hardware, or it may be due to other mechanisms such as caching. For example, the Commit-Reconcile model from Example 6.6 does not explicitly reorder operations, and it has caches. Nonetheless, we model it with reordering (and read forwarding) and without caching. Similarly, Corollary 6.4 characterizes coherence in terms of reordering.

The alternative view of reordering exposes the programmer directly to the additional concurrency by relaxing the program order. Operations that may be reordered are viewed by the programmer as logically concurrent. The computation we associate with a program reflects this view by eliminating the precedence dependencies between operations that may be reordered, as we saw in Example 3.17. Programs that generate the same computations are logically equivalent.

Example 6.23 On a system that preserves the order of operations only if they are on the same location, the following programs are logically equivalent; they generate the same computation.

```
int x, y = 0          int x, y = 0          int x, y = 0

P1: write(x,1)        P1: read(y)           P1: write(x,1)
    read(y)               write(x,1)            write(x,2)
    write(x,2)            write(x,2)            read(y)

P2: write(y,1)        P2: write(y,1)        P2: read(x)
    read(x)               read(y)               write(y,1)
    read(y)               read(x)               read(y)
```

These all generate the computation from Example 3.17. ■

These two views trade complexity in program structure for complexity in the consistency guarantees. The conventional view keeps the program structure simple: There are a fixed number of concurrent threads—one per processor—throughout an execution, with no explicit control dependencies between threads. Synchronization is implemented through the memory, which complicates the memory semantics. The relaxed-program-order view, on the other hand, allows concurrency at a single processor but provides a simpler memory model. In this view, the interface between the clients and the memory lies after the reordering transformation has been applied, so the memory model for a system is the model of the underlying memory.

One advantage of the relaxed-program-order view is that several memory models have the same underlying memory model. For example, the IBM/370 and Alpha processors both have a sequentially consistent underlying memory. A programmer who views reordering as a relaxation of the program order would therefore consider those systems sequentially consistent. Similarly, *TSO*, *PSO*, *RMO*, *CR* and *CRF* would all be viewed as providing the same memory consistency guarantees, that of the *RdFd* model, with different program semantics.

Because reordering is done on the clients side of the clients-memory interface, we no longer have the danger discussed in Section 6.5, where the system may not implement the generic memory model. However, we may wish to require that programs running on only one processor behave like sequential programs, despite the reordering allowed by the system. Informally, this requirement allows a system to run programs written for sequential machines without modification. This property is guaranteed by any system that preserves the order between operations on the same location; that is, any observation admitted by such a system for a serial computation is also sequentially consistent.

Lemma 6.11 Suppose $\{O_\ell\}_{\ell\in\mathcal{L}}$ is a location partition for $\mathcal{D}$, $M \in \mathfrak{M}_A^{\mathcal{D}}$, and $\mathcal{T}^r$ is a reordering transformation defined by *Preserve* such that $x.loc = y.loc \implies Preserve(x, y, a_x, a_y)$. If $C \in \mathfrak{C}_A^{\mathcal{D}}$ is serial[10] and $(\mathcal{T}^r(C), \rho) \in M$ then $(C, \rho) \in SC$.

Proof: By Lemma 6.10, $\mathcal{T}^r(M)$ is a memory model. Since C is completely race-free and $(C, \rho) \in \mathcal{T}^r(M)$, by Theorem 5.4, $(C, \rho) \in SC$. ■

The conditions of Lemma 6.10 and Lemma 6.11 are the same, but the motivation is different: Lemma 6.10 says that every value returned by the system for a processor-centric computation can be explained by some schedule of the computation; that is, that the system is a "real" memory system. Lemma 6.11 says that it executes sequential programs correctly.

The two views of reordering are not mutually exclusive. We may view some of the reordering allowed by a model as relaxing the program order while other reordering may be hidden from the programmer, and viewed merely as a formal mechanism to describe the consistency guarantees. For example, the original Commit-Reconcile & Fences model definition [100] gives explicit reordering rules for operations executed on an underlying Commit-Reconcile memory. It is natural for a programmer to take the relaxed-program-order view for the explicit reordering and the conventional view for the reordering that

[10]This lemma extends trivially to any determinate computation. We state it for serial computations because they model the sequential programs that motivate the lemma.

defines the Commit-Reconcile model. The programmer may even prefer to reason about the Commit-Reconcile guarantees using an operational model with explicit caches rather than the reordering with read forwarding model from Example 6.6.

When few operations may be reordered, the conventional view may be more appropriate. However, when most operations may be reordered and the programmer must use explicit fences to force operations to be executed in a particular order, the relaxed-program-order view seems more natural. Because the programmer must already consider out-of-order execution, it is natural to expose the reordering explicitly. In this view, the sequentiality of the operations of a program is an artifact of the language; like class definitions in a C++ program, the operations must be written in some order, but the programmer does not attach significance to this order.

The relaxed-program-order view of reordering also emphasizes the difference between fences and synchronization operations: A fence inhibits reordering; it affects the program order of a computation. It is not necessary to indicate a fence with an annotation because its effect is already reflected in the precedence dependencies. Synchronization operations, on the other hand, must be indicated by the annotations[11] because the order in which synchronized operations are executed is not fixed by the program order, but determined by the system. The system may appear to execute the operations in any order, as long as this order is consistent for all operations.

This distinction is often blurred in processor-centric models with explicit synchronization operations. For example, in weak ordering and release consistency, ordering is enforced between operations on different locations only by synchronization operations. To be useful, synchronization operations must preserve some precedence dependencies—that is, they must also be fences—but fences need not synchronize. The Commit-Reconcile & Fence model, for example, has "pure" fences; synchronization is done, in the operational model with explicit caches,[12] by the commit and reconcile operations.

6.7 Programmer-Centric Models

The system-centric models defined in Section 6.2 present a programmer with complicated memory semantics that can be difficult to use in reasoning about the behavior of programs [56]. To reduce the complexity of such reasoning, some researchers advocate a *programmer-centric approach* to specifying memory models [1, 41], in which a memory model is characterized by the set of programs that execute using the model as they do using sequential consistency. In this section, we show how to adapt this approach to the computation-centric framework. We define a class of *data-race-free programs* and prove that such programs cannot distinguish weak ordering from sequential consistency.

A *programmer-centric model* is

> specified as a contract between the system and the programmer where the programmer provides some information about the program to the system, and the system uses the

[11] For write serialization, the need for synchronization is indicated by the type of operation rather than explicitly in the annotation.

[12] There are no real synchronization operations in the reordering with read forwarding definition because only one schedule is used to explain all the return values; all the operations are synchronized.

information to provide both sequential consistency and high performance [1, p. 5].

The information provided by the programmer indicates the synchronization necessary to guarantee sequentially consistent behavior. The programmer must specify "sufficient synchronization" so that there are no "data races" in any sequentially consistent execution of the program, that is, in any execution of the program on a sequentially consistent system. Informally, a sequentially consistent execution corresponds to a schedule,[13] and a data race consists of two unsynchronized competing operations that are adjacent in the schedule.

We use annotations to provide the synchronization information and a client restriction to characterize the computations with sufficient synchronization. A programmer-centric memory model is defined by the client restriction that characterizes "sufficient synchronization" for that model. A memory system modeled by M implements a programmer-centric model characterized by CR if $M|_{CR}$ implements sequential consistency.

One difficulty with this approach is determining which schedules are sequentially consistent executions of a program. Although a computation may be generated by a program, some of its schedules may not be sequentially consistent executions of the program because the program may specify different computations depending on the values returned for earlier operations. Thus, there is a data dependency that affects the control structure of the program. The system enforces this dependency by synchronization. In particular, when a synchronization read returns the value written by a synchronization write, the write must be "performed at" every processor. Thus, the write precedes the read in the local schedule of every processor.[14]

To model the control dependencies between synchronization operations at different processors, we relax the well-formedness condition for processor-centric memories. In addition to a total order at each processor, the clients may specify precedence dependencies between conflicting synchronization operations. In particular, we allow a synchronization read to have a precedence dependency on a conflicting synchronization write at a different processor. For the read/write memories we consider, these are the only possible dependencies across processors. Formally,

$$WF_{\mathrm{P}+} = \left\{ C \in \mathfrak{C}_A^{\mathcal{M}} : \begin{array}{r} (p(x) = p(y) \implies (x,y) \in E_C \vee (y,x) \in E_C \vee x = y) \\ \wedge\ ((x,y) \in E_C \wedge p(x) \neq p(y) \\ \implies ann_C(x) = ann_C(y) = \mathrm{SYNC} \wedge x \in Wr \wedge y \in Rd|_{x.loc}) \end{array} \right\}$$

Edges between operations issued by the same processor are ***program order dependencies***, and edges between operations from different processors are ***cross-processor dependencies***.

We now define the *data-race-free* programmer-centric model; that is, we define the set of data-race-free computations. A system implements the data-race-free model if it guarantees sequential consistency for data-race-free computations. Our definition is a slight variation of the *data-race-free-0 (DRF0)* model of Adve [1], and the *properly-labeled-1 (PL1)* model of Gharachorloo [41].

[13] In the literature and for the dynamic memory models of Chapter 9, an execution also specifies the values returned for the operations. We omit the values for simplicity, since they can be derived from the schedule.

[14] This condition is weaker than the condition that every processor applies the write before any processor reads it, but it suffices for our purposes.

A computation is ***data-race-free*** if it is well-formed for a programmer-centric memory and it is (completely) race-free. Formally, $DRF = RF \cap WF_{P+}$.

Weak ordering preserves the per-location order of data-race-free computations.

Lemma 6.12 For $C \in WF_{P+}$, if $x \prec_C y$ and $x.loc = y.loc$ then $x \prec_{\mathcal{T}^r_{WO}(C)} y$.

Proof: If $p(x) = p(y)$ then $(x, y) \in E_C$ and $Preserve_{WO}(x, y, ann_C(x), ann_C(y))$, so $(x, y) \in E_{\mathcal{T}^r_{WO}(C)}$. Thus, $x \prec_{\mathcal{T}^r_{WO}(C)} y$. If $p(x) \neq p(y)$ let $E_{po} = \{(z, z') \in E_C : p(z) = p(z')\}$, and $E_{cp} = E_C - E_{po} = \{(z, z') \in E_C : p(z) \neq p(z')\}$. Because $C \in WF_{P+}$, both E_{po} and E_{cp} are partial orders. Thus, there exist $z_1, \ldots, z_{2n} \in V_C$ such that either $x = z_1$ or $(x, z_1) \in E_{po}$, either $y = z_{2n}$ or $(z_{2n}, y) \in E_{po}$, $(z_{2i-1}, z_{2i}) \in E_{cp}$ for $i = 1, \ldots, n$, and $(z_{2i}, z_{2i+1}) \in E_{po}$ for $i = 1, \ldots, n-1$. Because $C \in WF_{P+}$ and $(z_{2i-1}, z_{2i}) \in E_{cp}$, we have $ann_C(z_i) = \text{SYNC}$ for $i = 1, \ldots, 2n$, so none of these precedence dependenices are eliminated by $\mathcal{T}^r_{WO}(C)$. Thus, $x \prec_{\mathcal{T}^r_{WO}(C)} y$, as required. ■

We now prove the main result of this section: The data-race-free programmer-centric model is implemented by weak ordering; that is, weak ordering implements sequential consistency under data-race-free clients.

Theorem 6.13 *WO* implements *SC* under *DRF*.

Proof: We only need to prove that *WO* is a memory model under *DRF*, since *DRF* is more restrictive than *RF*, and every memory model implements *SC* under *RF*, by Theorem 5.4.

Suppose $\rho \in WO[C]$ for $C \in DRF$. Then $(\mathcal{T}^r_{WO}(C), \rho) \in \mathcal{T}^s_{\Psi_s \vee \Psi_{ws}}(Cache)$. $\mathcal{T}^s_{\Psi_s \vee \Psi_{ws}}(Cache)$ is obviously a memory model, so for each $x \in V_C$, there exists $\alpha \in Sch(\mathcal{T}^r_{WO}(C))$ with $\rho(x) = retval(x, \alpha)$. By Lemmas 6.9 and 6.12, there exists $\alpha' \in Sch(C)$ with $retval(x, \alpha') = retval(x, \alpha) = \rho(x)$. ■

Lemma 6.12 is the key to the proof that weak ordering implements sequential consistency under data-race-free clients. However, weak ordering provides stronger guarantees than necessary to prove Lemma 6.12. As long as a memory system does not allow synchronization reads to overtake any operations nor synchronization writes to be overtaken, does not reorder operations to the same location, and guarantees that all processors order synchronization reads after the synchronization writes whose values they read, the system will appear sequentially consistent to data-race-free clients.

Formally, let $\mathcal{T}^r_{DRF}$ be defined by

$$\begin{aligned} Preserve_{DRF}(x, y, a_x, a_y) \;\equiv\; & p(x) = p(y) \wedge ((x \in Rd \wedge a_x = \text{SYNC}) \\ & \qquad \vee (y \in Wr \wedge a_y = \text{SYNC}) \\ & \qquad \vee x.loc = y.loc) \\ & \vee (x \in Wr \wedge y \in Rd \wedge a_x = a_y = \text{SYNC} \wedge x.loc = y.loc) \end{aligned}$$

This transformation preserves the per-location order.

Lemma 6.14 For $C \in WF_{P+}$, if $x \prec_C y$ and $x.loc = y.loc$ then $x \prec_{\mathcal{T}^r_{DRF}(C)} y$

Proof: This proof is analogous to the proof of Lemma 6.12. The only difference is that when $p(x) \neq p(y)$, we require $z_1, \ldots, z_n \in V_C$ such that either $x = z_1$ or $(x, z_1) \in E_{po}$, either $y = z_{2n}$ or $(z_{2n}, y) \in E_{po}$, $(z_{2i-1}, z_{2i}) \in E_{cp}$ for $i = 1, \ldots, n$, $(z_{2i}, z_{2i+1}) \in E_{po}$ for $i = 1, \ldots, n-1$, *and also*, $z_{2i-1} \in Wr$, $z_{2i} \in Rd$, and $(z_{2i-1}).loc = z_{2i}.loc$ for $i = 1, \ldots, n$. These properties all hold since C is well-formed, so $\mathcal{T}^r_{DRF}$ does not eliminate the precedence dependencies. ■

Any memory using this transformation implements the data-race-free model.

Theorem 6.15 $\mathcal{T}^{\mathrm{r}}_{DRF}(GM)$ implements *SC* under *DRF*.

Proof: This proof is identical to the proof of Theorem 6.13, except that it uses Lemma 6.14 instead of Lemma 6.12. ■

Theorem 6.15 is a significant generalization of Theorem 6.13. It states very weak conditions that a system must satisfy to implement the data-race-free model. In particular, the system does not need to be coherent.[15] The system may not even implement generic memory for computations that are not data-race-free, as long as it respects the precedence dependencies preserved by $\mathcal{T}^{\mathrm{r}}_{DRF}$.

One apparent anomaly with Theorem 6.15 is that it allows the underlying memory model to be any memory model. In particular, it does not require the memory to order synchronization reads consistently at each processor after the synchronization write whose value they read. However, this requirement is captured by the computation: A data-race-free computation explicitly encodes data dependencies involving synchronization operations as precedence dependencies, and $\mathcal{T}^{\mathrm{r}}_{DRF}$ preserves these dependencies.

6.8 Discussion

The processor-centric approach to modeling memory systems has two factors in its favor. First, it extends the familiar sequential model of computing is a simple way, and thus, it is easy for programmers to understand. Second, shared memory multiprocessors generally satisfy the processor-centric assumption—that there is a fixed set of sequential processors accessing a shared read/write memory—and thus, the processor-centric models can closely match the guarantees of the actual system. These factors make processor-centric models appropriate for analyzing the guarantees of shared memory multiprocessors when programmers are exposed to the hardware. However, processor-centric models cannot capture the richer concurrent structure provided by high level concurrent programming languages, in which threads can be created and destroyed dynamically and concurrent threads may have a variety of mechanisms for synchronizing.

One approach to adapting processor-centric models for high level multithreaded languages is to introduce a *virtual processor* for each thread. The Java memory model, for example, is specified in this way [49]. Because a processor-centric model has a fixed set of processors and does not allow precedence dependencies across processors, we model the threads as all existing from the beginning of the program, and the first operation of a thread has an implicit data dependency on the operation that creates it. Thus, what appears as a control dependency in the program—the creation of a thread by its parent—is modeled as a data dependency. We consider the reinterpretation of control dependencies as data dependencies to be a significant drawback to this approach. The virtual processor approach also increases the gap between the model and the actual system, which was

[15] Many researchers seem to have assumed that coherence was necessary, treating it as a minimal condition that should be guaranteed by all multiprocessor systems [72, 52]. The original works on weak ordering and processor consistency implicitly assume coherence, or some similar condition [33, 47, 8], and release consistency specifically includes a "coherence requirement" [41]. However, as we saw in Example 6.14, the formal definition of release consistency does *not* guarantee coherence.

one of the main advantages of processor-centric models. Finally, virtual processors are still not general enough to express, in a natural fashion, the concurrent structure of some programming systems, including Cilk [105].

The restricted concurrency model of processor-centric models is one of the main reasons we advocate abandoning them, except for low-level descriptions of shared memory multiprocessors. Because there is no consensus yet on the "right way" to structure concurrent programs, it is especially important that we do not restrict the possible ways to do this by our choice of modeling techniques. There is, however, consensus that more structure is needed, because concurrent programs are difficult to reason about correctly [70]. Fortunately, as we have shown, we can cast processor-centric models into the computation-centric framework, so that we can leverage the processor-centric results in the literature.

Another disadvantage of processor-centric models is that they encourage programmers to view as privileged, the order in which operations are listed in the program, even if the system may reorder the operations. Similarly, system designers view the order in which operations are issued as special,[16] even though compilation may make this order completely different from the order of the corresponding program instructions.

[16]For example, many processors that reorder operations require that the operations are "retired" in the order they were issued [98].

Chapter 7

Locks

Locks eliminate *data races*, or unsynchronized access to the memory, by enforcing *exclusion constraints*; that is, by preventing some operations from executing concurrently. Locks are among the most common devices for structuring concurrent programs; most systems provide support for implementing them. Many other structuring devices, including critical sections and monitors, are implemented using locks. In this chapter, we show how to model locks in the computation-centric framework. We introduce very weak consistency conditions for a memory system with locks, and we show that the discipline of using locks to eliminate all data races guarantees sequential consistency on systems that meet these conditions.

Because threads accessing shared data concurrently may interfere with each other, programming concurrent systems, even sequentially consistent ones, is challenging [70, 56]. Typically, a lock is used to *protect* shared data. Before the data can be accessed, the lock must be *acquired* and *held*. When the data is no longer needed, the lock is *released*. We focus on the simplest and most common kind of lock, the *mutual exclusion (mutex) lock*. Informally, only one thread can hold a mutex lock at any time. If a thread tries to acquire a lock that is held by another thread, the first thread blocks until the lock is released. A system that ensures that at most one thread holds a lock at any time is said to *respect locking*.

The operations of a thread beginning with the acquire and ending with the release form a *critical section* or a *locked section*. However, in our framework, there is no explicit notion of threads. Instead, because control flow is represented by the edges of a computation, we say that operations between an acquire and its matching release in the computation order hold the lock and form a locked section.

Accessing data without using locks leads to data races, which result in behaviors that depend on timing conditions that the programmer cannot predict. Thus, data races are usually considered bugs [16, 70, 96, 25]. Protecting the data with locks leads to well-structured programs that are easier to reason about. Furthermore, when data races are eliminated using locks, even memories with weak consistency guarantees appear to be strongly consistent [70]. We formally define a new memory model called *weak sequential locking* that is weaker than many similar models in the literature but still has this property.

A greatly abbreviated version of the material in this chapter was presented at the Symposium for Parallel Algorithms and Architectures in 2001 [78].

Some systems provide locks with weaker exclusion constraints than mutex locks. For example, the *shared/exclusive lock* is a common variant that provides a shared and an exclusive mode. Several threads can hold a lock in shared mode concurrently as long as no thread holds the lock in exclusive mode. Although we focus on mutex locks in this thesis, the results in this chapter can be extended to other kinds of locks.

Outline: Section 7.1 defines several graph-theoretic notions used in this and later chapters. Section 7.2 formally defines computations with locks and what it means to acquire, release and hold locks. In Section 7.3, we characterize well-formed computations with locks, and in Section 7.4, we define schedules that respect locking. Section 7.5 defines several memory models with locking, including sequential consistency with locking and weak sequential locking. In Section 7.6, we define data races in the presence of locks and prove that weak sequential locking implements sequential consistency with locking for data-race-free computations, and in Section 7.7, we contrast these results with the results about programmer-centric models in Section 6.7. In Section 7.8, we specialize our discussion and results to the important case in which the memory has locations, each with an associated lock. Section 7.9 shows how to extend the results of this chapter to shared/exclusive locks. Section 7.10 discusses the advantages of modeling locks directly in the computation and points out directions for further study.

Reading Guide: In a sense, this chapter, and particularly Theorem 7.25, which says that weak sequential locking implements sequential consistency with locking for data-race-free programs, is the center of this thesis. The earlier definitions and results, except those in Chapter 6, build up to this theorem. It is also the key to using weakly consistent memory to implement sequentially consistent transactions in the next chapter.

It may be best to skip Section 7.1 initially, because the definitions and lemmas there are motivated by their use in later sections and are chiefly important for understanding the proofs of Sections 7.3 and 7.4. Sections 7.2 to 7.6 form the core of this chapter and should be read in sequence. Like the rest of the thesis, these sections may be read informally first, as later parts depend only on the results mentioned, and not the proofs of those results. The later sections are independent from each other and may be read in any order.

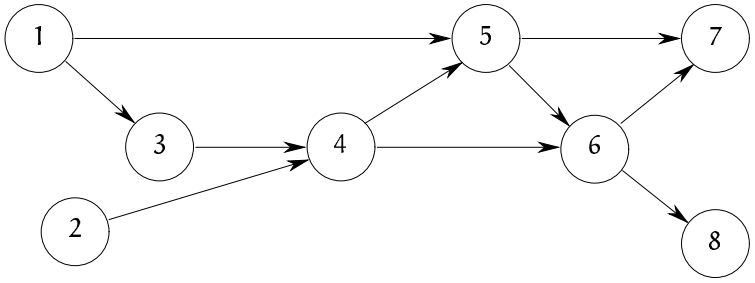
7.1 Preliminary Graph Theory: Regions, Guards and Sections

In this section, we introduce several graph-theoretic concepts that are useful for defining locks on computations. In particular, we define an *enclosure* of a dag and prove several properties about enclosures that are used throughout this chapter and the next. The definition of an enclosure of a dag is the hidden workhorse of this chapter. Though it seems straightforward, a considerable fraction of the intellectual effort of this thesis involved fine-tuning this definition so that it simply and accurately captured the conditions that make the statements and proofs in this chapter understandable. The motivation for the definitions and lemmas in this section appears in Section 7.3, where they are first used. They are collected here because they rely only on graph theory, and not on any properties of computations. This section may be skipped initially and used as a reference as needed.

A ***region*** of a dag G is any subset of V_G. We associate regions with the subgraphs induced on them, and apply terminology for these subgraphs to the regions. For example, S is a ***prefix*** of G if $G|_S$ is a prefix of G. We use interval notation to denote regions between two vertices. For example, the ***region from*** u ***to*** v in G is $[u,v]_G = \{w \in V_G : u \preceq w \preceq v\}$. Similarly, $[u,v)_G = \{w \in V_G : u \preceq w \prec v\}$. We may omit the subscript when the dag is clear from context. A region S is ***convex*** if $[u,v] \subseteq S$ for all $u,v \in S$. A ***serial region*** of G is a convex region that is totally ordered by $\preceq_G$.

A *guard* of a region is a root of the region that is included in every convex path into the region. Similarly, a *rear guard* is an inverse root included in every convex path out of the region. Formally, a ***(front) guard*** of a region S in a dag G is a vertex $u \in S$ such that for all $v \in S$ and $w \notin S$, $u \preceq v$ and $w \prec v \implies w \prec u$; a ***rear guard*** of S is a vertex $u \in S$ such that for all $v \in S$ and $w \notin S$, $v \preceq u$ and $v \prec w \implies u \prec w$.

Example 7.1 In the following dag, vertex 4 is a guard for the region $\{4,5,6,7\}$ and vertex 5 is a rear guard for the region $\{3,4,5\}$.



■

Lemma 7.1 A region has at most one guard and at most one rear guard.

Proof: If u and u' are guards for S then $u,u' \in S$, so $u \preceq u'$ and $u' \preceq u$. Thus, $u = u'$. Similarly, if u and u' are rear guards for S then $u,u' \in S$, so $u' \preceq u$ and $u \preceq u'$. Thus, $u = u'$. ■

If a region S has a guard, we denote it by $gd_G(S)$; if it has a rear guard, we denote it by $rgd_G(S)$. As usual, we may omit the subscript when the dag is clear from context. An ***enclosure*** is a region with both a front guard and a rear guard.

Example 7.2 The enclosures of the graph from Example 7.1 are $\{1,3\}$, $\{4,5,6\}$, $\{4,5\}$, $\{5,6\}$, and all the regions with a single vertex. ■

Lemma 7.2 Every singleton subset of V_G is an enclosure of G.

Proof: Immediate from the definitions of enclosure, guard, and rear guard. ■

An enclosure is the region from its guard to its rear guard.

Lemma 7.3 If S is an enclosure then $S = [gd(S), rgd(S)]$.

Proof: If $u \in S$ then by the definition of guard and rear guard, $gd(S) \preceq u \preceq rgd(S)$.

Suppose $u \notin S$. If $u \not\preceq rgd(S)$ then $u \notin [gd(S), rgd(S)]$. If $u \preceq rgd(S)$ then, since $rgd(S) \in S$, by the definition of guard, $u \preceq gd(S)$, and $u \notin [gd(S), rgd(S)]$. ■

If any element of one enclosure precedes any element of another enclosure that is disjoint from it, then every element of the first enclosure precedes every element of the second enclosure. In particular, the rear guard of the first enclosure precedes the guard of the second enclosure.

Lemma 7.4 If S and S' are disjoint enclosures of G with $u \prec v$ for some $u \in S$ and $v \in S'$, then $rgd(S) \prec gd(S')$.

Proof: Since S and S' are disjoint, $v \notin S$, so by the definition of a rear guard, $rgd(S) \prec v$. Since $rgd(S) \notin S'$, we have by the definition of a guard, $rgd(S) \prec gd(S')$. ■

Any vertex that is ordered by a topological sort of a dag between the guard and rear guard of an enclosure is either in the enclosure or is not ordered by the dag with respect to any vertex of the enclosure.

Lemma 7.5 If α is a topological sort of a dag G and S is an enclosure of G then for all $u \notin S$ such that $gd(S) <_\alpha u <_\alpha rgd(S)$, u is not ordered by G with respect to any vertex in S.

Proof: Since $\alpha \in Sch(C)$, if $gd(S) <_\alpha u <_\alpha rgd(S)$ for $u \notin S$, then $u \not\preceq gd(S)$ and $rgd(S) \not\preceq u$. Thus, for $v \in S$, $u \not\preceq v$ and $v \not\preceq u$. ■

The intersection and union of two intersecting enclosures are also enclosures, and their guards and rear guards are the guards and rear guards of the original enclosures.

Lemma 7.6 If S and S' are enclosures with a nonempty intersection, then $S \cap S'$ is an enclosure with $gd(S \cap S') \in \{gd(S), gd(S')\}$ and $rgd(S \cap S') \in \{rgd(S), rgd(S')\}$, and $S \cup S'$ is an enclosure with $gd(S \cup S') \in \{gd(S), gd(S')\}$ and $rgd(S \cup S') \in \{rgd(S), rgd(S')\}$.

Proof: Suppose $gd(S) \in S'$. For all $v \in S \cap S' \subseteq S$, $gd(S) \preceq v$ and for all $w \notin S \cap S'$, either $w \notin S$ or $w \notin S'$, so if $w \prec v$, either $w \prec gd(S)$ or $w \prec gd(S') \preceq gd(S)$. Thus, $gd(S)$ is a guard of $S \cap S'$.

For all $v \in S \cup S'$, either $v \in S'$ or $v \in S$. In the first case, $gd(S') \preceq v$ and for all $w \notin S \cup S' \supseteq S'$, if $w \prec v$ then $w \prec gd(S')$. In the second case, $gd(S') \preceq gd(S) \preceq v$, and for all $w \notin S \cup S' \supseteq S$, if $w \prec v$ then $w \prec gd(S)$, and since $gd(S) \in S'$, $w \prec gd(S')$. Thus, $gd(S')$ is a guard of $S \cup S'$.

If $gd(S) \notin S'$, let $u \in S \cap S'$. Then $gd(S) \prec u \in S'$, so $gd(S) \prec gd(S')$. Since $gd(S') \preceq u$ and $[gd(S), u] \subseteq S$, we have $gd(S') \in S$. By symmetry with the previous case, $gd(S')$ is a guard of $S \cap S'$ and $gd(S)$ is a guard of $S \cup S'$.

The proof that $rgd(S \cap S')$ and $rgd(S \cup S')$ are either $rgd(S)$ or $rgd(S')$ is similar. ■

An important, and perhaps surprising, property of enclosures is that it is possible to find a topological sort of any dag in which every enclosure is a contiguous subsequence. This property holds even if the enclosures are overlapping.

Lemma 7.7 For any dag G, there is a topological sort α such that every enclosure of G appears contiguously in α.

Proof: We prove this lemma by strong induction on the number of vertices in G. If every enclosure is either V_G or a singleton set, then any topological sort of G satisfies this lemma. Otherwise, let S be any enclosure that is neither V_G nor a singleton set.

Define G' so that $V_{G'} = V_G - S \cup \{gd(S)\}$ and $E_{G'} = \prec_G|_{V_{G'}}$. We have $|V_{G|_S}| = |S| < |V_G|$, and, because $|S| > 1$, $|V_{G'}| < |V_G|$. Thus, by the inductive hypothesis, there are topological sorts α for G' and β for $G|_S$ in which every enclosure appears contiguously. Since $gd(S) \in V_{G'}$, we can

decompose α into $\alpha_1 \cdot gd(S) \cdot \alpha_2$. Let $\gamma = \alpha_1 \cdot \beta \cdot \alpha_2$. We show that γ is a topological sort of G in which every enclosure appears contiguously.

To see that γ is a topological sort of G, first note that it is a serialization of V_G because β is a serialization of S and $\alpha_1 \cdot \alpha_2$ is a serialization of $V_{G'} - \{gd(S)\} = V_G - S$. Suppose $(u, v) \in E_G$. If $u, v \in S$ then $u <_\beta v$, so $u <_\gamma v$. If $u, v \notin S$ then $u, v \in V_{G'}$ and $(u, v) \in E_{G'}$, so $u <_\alpha v$ and $u <_\gamma v$. If $u \notin S$ and $v \in S$ then, since S is an enclosure of G, $u \prec_G gd_G(S) \preceq_G v$, so $u <_\alpha gd_G(S) \leq_\beta v$, and thus, $u <_\gamma v$. If $u \in S$ and $v \notin S$ then $gd_G(S) \prec_G u \prec_G v$, so $gd_G(S) <_\alpha v$ and $v \in \mathit{elems}(\alpha_2)$. Since $u \in S = \mathit{elems}(\beta)$, $u <_\gamma v$.

Suppose S' is an enclosure of G. If $gd(S'), rgd(S') \in S$ then $S' \subseteq S$, so S' is an enclosure of $G|_S$ and it appears contiguously in β. Thus, S' appears contiguously in γ. If S and S' are disjoint then S' is an enclosure of G' because $u \prec_{G'} v \iff u \prec_G v$ for all $u, v \in V_{G'}$. Thus, S' appears contiguously in α, and since $gd_G(S) \notin S'$, S' appears contiguously in either α_1 or α_2. In either case, S' appears contiguously in γ. If $gd(S'), rgd(S') \notin S$ with S and S' not disjoint, then by Lemma 7.6, $S \subseteq S'$. So $S' - S \cup \{gd(S)\}$ appears contiguously in α. Because $\mathit{elems}(\beta) = S \subseteq S'$ includes $gd(S)$, S' appears contiguously in γ. If $gd(S') \notin S$ and $rgd(S') \in S$ then since $gd(S') \preceq_G rgd(S')$ and S is an enclosure, we have $gd(S') \prec_G gd(S) \preceq_G rgd(S') \preceq_G rgd(S)$. By Lemma 7.6, $S \cap S' = [gd(S), rgd(S')]$ is an enclosure, so it appears contiguously in β, and because it includes $gd(S)$, it is a prefix of β. Also, $S' - S \cup \{gd(S)\}$ is an enclosure of G' because $u \prec_{G'} v \iff u \prec_G v$ for all $u, v \in V_{G'}$, so it appears contiguously in α, and in fact, is a suffix of $\alpha_1 \cdot gd(S)$. Thus, S' appears contiguously in γ. Similarly, if $gd(S') \in S$ and $rgd(S') \notin S$, we have that S' appears contiguously in γ. ■

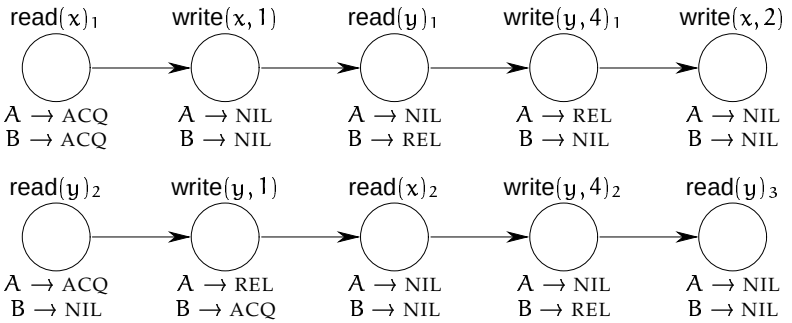
7.2 Computations with Locks

In the traditional setting, a thread requests a lock from the system, and waits until the lock is *acquired*. Once acquired, the lock is *held* by the thread until the thread *releases* it. The operations executed by the thread while a lock is held—that is, between an acquisition of a lock and its subsequent release—comprise a *locked section* of the computation. In this section, we show how to model the acquisition and release of locks in the computation-centric framework, where there is no explicit notion of threads or time. We also formally define the locked sections of a computation with locks.

We model locks as being independent of operations on the data type. A system is parameterized by its set L of locks. Each lock is treated independently, and may be acquired or released by any operation, which we specify using annotations as follows: ACQ indicates that the lock is to be acquired; REL indicates that the lock is to be released; A/R indicates that the lock is to be acquired and then immediately released; and NIL indicates that the lock is neither acquired nor released.

Formally, a ***computation with (mutex) locks*** L is a computation with annotation set $A_L = \{\lambda \colon L \to \mathit{LockOps}\}$, where $\mathit{LockOps} = \{\text{ACQ}, \text{REL}, \text{A/R}, \text{NIL}\}$. For $x \in V_C$ and $l \in L$, we use the notation $\Lambda_C(x, l) = \mathit{ann}_C(x)(l)$. An operation x ***acquires*** a lock $l \in L$ in C if $\Lambda_C(x, l) \in \{\text{ACQ}, \text{A/R}\}$, and it ***releases*** l if $\Lambda_C(x, l) \in \{\text{REL}, \text{A/R}\}$. We denote the set of all operations that acquire l by $\mathit{Acqs}_C(l) = \{x \in V_C : \Lambda_C(x, l) = \text{ACQ} \vee \Lambda_C(x, l) = \text{A/R}\}$, and the set of all operations that release l by $\mathit{Rels}_C(l) = \{x \in V_C : \Lambda_C(x, l) = \text{REL} \vee \Lambda_C(x, l) = \text{A/R}\}$. We omit the subscript when the computation is clear from context.

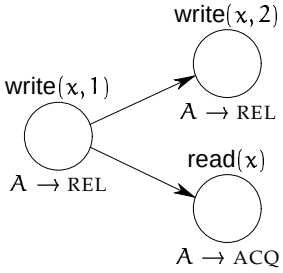
Example 7.3 With the annotation set A_L, the computation from Example 3.12 might be better represented as follows:



We often omit the entries for locks that are neither acquired nor released. ■

Our definition allows clients to specify any pattern of acquiring and releasing locks, even if the resulting computation is incompatible with our intuitive notion of locking. For example, an operation may specify that it releases a lock that has not been acquired. In Section 7.3, we discuss a well-formedness condition that rules out such computations.

Example 7.4 Although the following computation on $\mathcal{M}$ with locks $\{A\}$ can be specified, it is not compatible with the notion that a lock must be acquired before it can be released.



■

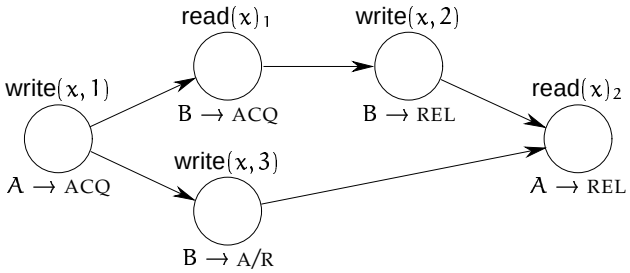
The operation that acquires a lock, and any operation that follows it in the partial order defined by the computation, *holds* the lock until it is released. That is, an operation holds a lock if it acquires the lock, or is preceded by an operation that acquires the lock with no intervening operation that releases the lock. The operation that releases the lock holds the lock.

Formally, an operation y ***holds*** $l \in L$ ***acquired by*** $x \in Acqs(l)$ in C if $[x, y) \cap Rels(l) = \emptyset$; that is, if there is no $z \in Rels(l)$ such that $x \preceq z \prec y$. We also say that x ***acquires*** l ***for*** y in C. The set $Holds(l, x)$ of operations for which x acquires l is called a ***locked section*** for l, or an l-***section***, of C. We say that y ***holds*** l, or that l is ***held by*** y, if some operation acquires l for y. We denote the set of operations that hold l by $Holds_C(l) = \bigcup_{x \in Acqs(l)} Holds_C(l, x)$. As usual, we omit the subscript when the computation is clear from context.

Example 7.5 In the computation from Example 7.3, $\mathsf{read}(x)_1$ acquires A for itself, $\mathsf{write}(x,1)$, $\mathsf{read}(y)_1$ and $\mathsf{write}(y,4)_1$, and it acquires B for itself, $\mathsf{write}(x,1)$ and $\mathsf{read}(y)_1$. $\mathsf{read}(y)_2$ acquires A for itself and $\mathsf{write}(y,1)$, and $\mathsf{write}(y,1)$ acquires B for itself, $\mathsf{read}(x)_2$ and $\mathsf{write}(y,4)_2$. Thus, the A-sections are $\{\mathsf{read}(y)_2, \mathsf{write}(y,1)\}$ and $\{\mathsf{read}(x)_1, \mathsf{write}(x,1), \mathsf{read}(y)_1, \mathsf{write}(y,4)_1\}$, and the B-sections are $\{\mathsf{read}(x)_1, \mathsf{write}(x,1), \mathsf{read}(y)_1\}$ and $\{\mathsf{write}(y,1), \mathsf{read}(x)_2, \mathsf{write}(y,4)_2\}$. ■

Example 7.6 In the computation from Example 7.4, $\mathsf{read}(x)$ acquires A for itself. No other operation holds the lock. Thus, the only locked section is the A-section $\{\mathsf{read}(x)\}$. ■

Example 7.7 In the following computation on $\mathcal{M}$ with locks $\{A, B\}$, $\mathsf{write}(x, 1)$ acquires A for every operation, $\mathsf{read}(x)_1$ acquires B for itself and $\mathsf{write}(x, 2)$, and $\mathsf{write}(x, 3)$ acquires B for itself. Thus, it has one A-section with all the operations, and two B-sections, $\{\mathsf{read}(x)_1, \mathsf{write}(x, 2)\}$ and $\{\mathsf{write}(x, 3)\}$.



■

7.3 Well-Formedness

Although the annotations allow the clients to specify any pattern of acquiring and releasing locks, clients are expected to use the locks in a restricted way. For example, an operation should not release a lock that it does not hold, nor should it acquire a lock that it already holds. In this section, we define a well-formedness condition for clients specifying a computation with locks, which formally expresses the properties we expect of computations specified by the clients. There are other possible well-formedness conditions, as we discuss at the end of this section.

Informally, we impose four restrictions on how the clients may acquire and release locks. First, the clients may not release a lock unless it is held. Second, they may not acquire a lock that is already held. Third, acquisitions and releases of locks must come in matching pairs; that is, for each operation that acquires a lock, there must be a unique corresponding operation that releases it. Fourth, the acquisition of a lock must "guard" its locked section in the following sense: If x acquires a lock for y then any operation that precedes y and does not hold the lock acquired by x must also precede x.

Formally, a computation with locks L is ***well-formed*** if for all $l \in L$, the following conditions are satisfied:

- $\mathit{Rels}(l) \subseteq \mathit{Holds}(l)$; that is, only operations that hold l release it.
- For each $x \in \mathit{Acqs}(l)$,
  - $\mathit{Holds}(l, x) \cap \mathit{Acqs}(l) = \{x\}$; that is, no operation that holds l acquired by x, other than x itself, acquires l.
  - there exists $y \in \mathit{Rels}(l)$ such that $\mathit{Holds}(l, x) = [x, y]$; that is, there is a unique release operation corresponding to x.
  - For $y \in \mathit{Holds}(l, x)$ and $z \notin \mathit{Holds}(l, x)$, if $z \prec y$ then $z \prec x$; that is, x guards $\mathit{Holds}(l, x)$.

We denote the set of well-formed computations by WF_{L}.

Example 7.8 The computations from Examples 7.3 and 7.7 are well-formed; the computation from Example 7.4 is not. ■

In a well-formed computation, an operation cannot hold the same lock acquired by two different operations. Thus, locked sections for the same lock are disjoint; that is, the set of operations that acquire a lock by one operation is disjoint from the set of operations that acquire the same lock by a different operation.

Lemma 7.8 If C is a well-formed computation with locks L, and $x, x' \in \mathit{Acqs}(l)$ for some $l \in L$ such that $x \neq x'$, then $\mathit{Holds}(l, x) \cap \mathit{Holds}(l, x') = \emptyset$.

Proof: Let $S = \mathit{Holds}(l, x)$ and $S' = \mathit{Holds}(l, x')$. Because C is well-formed, $x \notin \mathit{Holds}(l, x')$ and $x' \notin \mathit{Holds}(l, x)$. Suppose, for contradiction, $y \in S \cap S'$. By the definition of $\mathit{Holds}(l, x)$, $x \preceq y$. Because C is well-formed and $x \notin \mathit{Holds}(l, x')$, $x \prec x'$. By symmetry, $x' \prec x$, which is a contradiction. ■

The previous lemma means that $\{\mathit{Holds}(l, x)\}_{x \in \mathit{Acqs}(l)}$ is a partition of $\mathit{Holds}(l)$, so every operation x that holds a lock l is in a unique l-section. When the computation is clear from context, we denote the l-section that contains $x \in \mathit{Holds}(l)$ by $S_l(x)$. Formally, $S_l(x) = \mathit{Holds}(l, y)$ such that $y \in \mathit{Acqs}(l)$ and $x \in \mathit{Holds}(l, y)$.

The well-formedness condition requires a locked section to have a final operation that releases the lock. No other operation in the locked section releases the lock.

Lemma 7.9 For any l-section S of a well-formed computation with locks, $|S \cap \mathit{Rels}(l)| = 1$.

Proof: Let $x \in \mathit{Acqs}(l)$ be such that $S = \mathit{Holds}(l, x)$. By well-formedness, there exists $y \in \mathit{Rels}(l)$ such that $S = [x, y]$. Because y holds l acquired by x, $[x, y) \cap \mathit{Rels}(l) = \emptyset$. Thus, $S \cap \mathit{Rels}(l) = [x, y] \cap \mathit{Rels}(l) = ([x, y) \cap \mathit{Rels}(l)) \cup (\{y\} \cap \mathit{Rels}(l)) = \{y\}$. ■

Just as the operation that acquires a lock guards its locked section, the operation that releases the lock is a "rear guard" for the section in that every operation outside the section that is preceded by any operation in the section is preceded by the release operation.

Lemma 7.10 In a well-formed computation with locks L, for $l \in L$, $x \in \mathit{Acqs}(l)$, $y \in \mathit{Holds}(l, x) \cap \mathit{Rels}(l)$, $z \in \mathit{Holds}(l, x)$ and $z' \notin \mathit{Holds}(l, x)$, if $z \prec z'$ then $z \preceq y \prec z'$.

Proof: By well-formedness and Lemma 7.9, $\mathit{Holds}(l, x) = [x, y]$, so $x \preceq z \preceq y$.

Since $z' \notin \mathit{Holds}(l, x)$ and $x \preceq z \prec z'$, there exists $y' \in \mathit{Rels}(l)$ such that $x \preceq y' \prec z'$. Choose an "earliest" such y'; that is, choose $y' \in \mathit{Rels}(l)$ such that $x \preceq y' \prec z'$ and there is no $y'' \in \mathit{Rels}(l)$ such that $x \preceq y'' \prec y'$. By definition, $y' \in \mathit{Holds}(l, x)$, so by Lemma 7.9, $y' = y$. Thus, $y \prec z'$. ■

The locked sections of a well-formed computation with locks define *regions*—sets of operations—with special characteristics: The operation that acquires the lock for a locked section S is denoted by $\mathit{gd}(S)$, the *guard* of S. The operation that releases the lock is $\mathit{rgd}(S)$, the *rear guard* of S. A region with a guard and a rear guard is called an *enclosure*. Regions, guards, rear guards and enclosures are formally defined in Section 7.1.

The following lemma summarizes the basic properties of an l-section:

Lemma 7.11 In a well-formed computation C with locks L, for $l \in L$ and $x \in \mathit{Holds}(l)$,

- $S_l(x)$ is an enclosure of C.
- $S_l(x) = \mathit{Holds}(l, \mathit{gd}(S_l(x)))$.
- $S_l(x) \cap \mathit{Acqs}(l) = \{\mathit{gd}(S_l(x))\}$.
- $S_l(x) \cap \mathit{Rels}(l) = \{\mathit{rgd}(S_l(x))\}$.
- $S_l(x) = \{x\} \iff \Lambda(x, l) = \text{A/R}$.

Proof: By Lemma 7.8, we know that $S_l(x)$ is well-defined. By well-formedness and Lemma 7.9, there are $y \in \mathit{Acqs}(l)$ and $z \in \mathit{Rels}(l)$ such that $S_l(x) = \mathit{Holds}(l, y) = [y, z]$, and $S_l(x) \cap \mathit{Acqs}(l) = \{y\}$ and $S_l(x) \cap \mathit{Rels}(l) = \{z\}$. By well-formedness and Lemma 7.10, $S_l(x)$ is an enclosure with $y = \mathit{gd}(S_l(x))$ and $z = \mathit{rgd}(S_l(x))$. Finally, $S_l(x) = \{x\}$ if and only if $x = \mathit{gd}(S_l(x)) = \mathit{rgd}(S_l(x))$, which is true if and only if $x \in \mathit{Acqs}(l) \cap \mathit{Rels}(l)$, that is, if $\Lambda(x, l) = \text{A/R}$. ■

It follows immediately that the operations that acquire l are the guards of the l-sections, and those that release l are the rear guards.

Corollary 7.12 For any lock $l \in L$ of a well-formed computation with locks L, we have $\mathit{Acqs}(l) = \{\mathit{gd}(S_l(x)) : x \in \mathit{Holds}(l)\}$ and $\mathit{Rels}(l) = \{\mathit{rgd}(S_l(x)) : x \in \mathit{Holds}(l)\}$.

From now on, we consider only computations that are well-formed.

We can relax our well-formedness assumptions so that multiple operations release a lock acquired by a single operation. This relaxation requires the memory system to keep track of partial releases of a lock, which greatly complicates the semantics. We opt for the simpler semantics in this thesis.

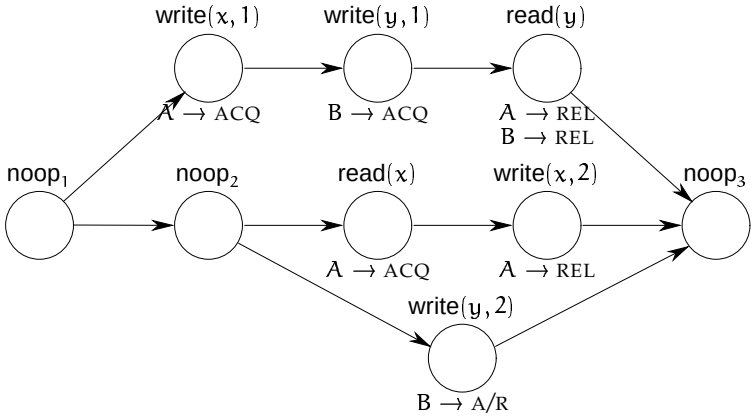
Even if operations that acquire and release locks must come in matching pairs, the well-formedness condition may be relaxed by allowing an operation that acquires a lock but is not a guard of the region for which it acquires the lock. For example, Cheng's proposal for introducing concurrency within critical sections in Cilk [24] allows this relaxation. However, the resulting locked regions do not have the nice properties of enclosures, and so are more difficult to reason about. Again, we opt for the simpler semantics.

Some systems have an even stronger well-formedness condition: They do not permit concurrency while a lock is being held; that is, every locked section must be serial. In a traditional setting with threads, a thread cannot fork or spawn another thread while it holds a lock. Formally, we model this condition as a client restriction, the ***serial locked sections restriction***:

$$\mathit{SerLockSec} = \{C \in \mathit{WF}_L : \text{every locked section of } C \text{ is serial}\}.$$

Example 7.9 The computation from Example 7.3 is in *SerLockSec*, but the one from Example 7.7 is not. ■

Example 7.10 The following computation has serial locked sections:



■

Although the serial locked sections restriction is a very strong restriction, it is commonly assumed when programming concurrent systems. For example, for processor-centric systems, discussed in Chapter 6, any well-formed computation with locks satisfies this condition, because the processors that acquire and release locks are sequential. We use this restriction in Section 7.8.

Much of the literature on data race detection [96, 25] also assumes serial locked sections, which greatly simplifies the algorithms for detecting data races. If locked sections may have concurrency within them, then operations within the locked section may comprise a data race. We discuss data races in more detail in Section 7.6.

7.4 Respecting Locking

Clients specify locking for the same reason they specify precedence dependencies: to restrict how the system may execute the operations of a computation. In the traditional setting, a system with locks ensures that no lock is held by two threads at the same time. In this section, we adapt this notion to the computation-centric framework, where there is no notion of threads or time, by defining what it means for a schedule to *respect locking*. We also prove several lemmas about schedules that respect locking. Just as we require schedules to respect precedence dependencies for memory models in general, in modeling memories with locks, we restrict our attention to schedules that respect locking.

Informally, each lock must be acquired before it is released, and it must be released before it is acquired again. Clients may specify concurrent requests to acquire a lock; the system must ensure that the lock is released before allowing it to be acquired again. Such a system is said to respect locking.

Formally, for a schedule α of a computation C with locks L, we define the ***acquire-release schedule*** of α for a lock $l \in L$ to be the projection onto the operations that acquire or release l, denoted by $\alpha^l = \alpha|_{Acqs(l) \cup Rels(l)}$. A schedule α ***respects*** l if $\alpha^l{}_{[1]} \in Acqs(l)$ and $\Lambda(\alpha^l{}_{[i]}, l) = \text{ACQ} \iff \Lambda(\alpha^l{}_{[i+1]}, l) = \text{REL}$, for $1 \leq i < |\alpha^l|$; that is, the first operation of an acquire-release schedule acquires the lock, and an operation acquires but does not release the lock if and only if the next operation in the acquire-release schedule releases and does not acquire the lock. A schedule ***respects (mutex) locking*** for L if it respects l for all $l \in L$. We denote the set of schedules of C that respect locking by $RespLock(C)$.

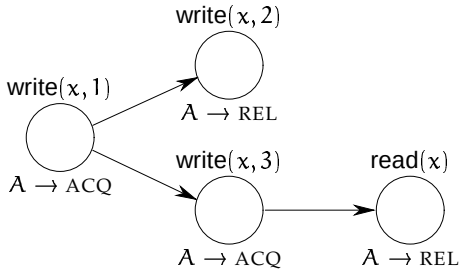
Example 7.11 For the computation from Example 7.7, only two schedules respect locking:

$$\mathsf{write}(x,1), \mathsf{read}(x)_1, \mathsf{write}(x,2), \mathsf{write}(x,3), \mathsf{read}(x)_2$$
$$\mathsf{write}(x,1), \mathsf{write}(x,3), \mathsf{read}(x)_1, \mathsf{write}(x,2), \mathsf{read}(x)_2$$

The other schedule, $\mathsf{write}(x,1), \mathsf{read}(x)_1, \mathsf{write}(x,3), \mathsf{write}(x,2), \mathsf{read}(x)_2$, does not respect B. ■

Example 7.12 The computation from Example 7.4 has no schedules that respect locking. ■

Example 7.13 A computation that is not well-formed may have schedules that respects locking. Consider the following computation on $\mathcal{M}$ with locks $\{A, B\}$:



The schedule $\mathsf{write}(x,1), \mathsf{write}(x,2), \mathsf{write}(x,3), \mathsf{read}(x)$ respects locking—it is the only one—even though the computation is not well-formed. ■

In the acquire-release schedule of any schedule that respects locking, an acquire is immediately followed by its matching release.

Lemma 7.13 If $C \in WF_L$ and α respects $l \in L$ then for all i,

$$\Lambda(\alpha^l{}_{[i]}, l) = \text{ACQ} \implies \alpha^l{}_{[i+1]} = rgd(S_l(\alpha^l{}_{[i]})).$$

Proof: Suppose this lemma is not true. Pick the minimum i violating this lemma, that is, with $\Lambda(\alpha^l{}_{[i]}, l) = \text{ACQ}$ and $\alpha^l{}_{[i+1]} \neq rgd(S_l(\alpha^l{}_{[i]}))$. Let $x = \alpha^l{}_{[i+1]}$ and $y = gd(S_l(x))$. Because α respects l, $\Lambda(x, l) = \text{REL}$, so, since C is well-formed, $\Lambda(y, l) = \text{ACQ}$, $x = rgd(S_l(y))$ and $y \prec_C x$. Choose j such that $y = \alpha^l{}_{[j]}$. Because α is a schedule of C, we have $y <_{\alpha^l} x$, or equivalently, $j < i+1$. We know $j \neq i$ because $rgd(S_l(y)) = x \neq rgd(S_l(\alpha^l{}_{[i]}))$. But then $\alpha^l{}_{[j+1]} \neq x = rgd(S_l(y))$, contradicting the minimality of i. ■

For well-formed computations, there is a more natural characterization of schedules that respect locking: A schedule respects a lock if the release corresponding to each acquire of the lock appears before the next acquire of the lock.

Lemma 7.14 For $C \in WF_L$, $\alpha \in Sch(C)$ and $l \in L$, α respects l if and only if for all l-sections S and S', $gd(S) <_\alpha gd(S') \implies rgd(S) <_\alpha gd(S')$.

Proof: Suppose α respects l, and S and S' are l-sections such that $gd(S) <_\alpha gd(S')$. If $S = \{gd(S)\}$ then $rgd(S) = gd(S) <_\alpha gd(S')$. Otherwise, since $gd(S), gd(S') \in Acqs(l)$ by Corollary 7.12, we have $gd(S) = \alpha^l{}_{[i]}$ and $gd(S') = \alpha^l{}_{[i']}$ with $i < i'$, and $\Lambda(gd(S), l) = \text{ACQ}$. By Lemma 7.13, $rgd(S) = \alpha^l{}_{[i+1]}$, so $rgd(S) \leq_\alpha gd(S')$. Since $rgd(S) \neq gd(S')$, $rgd(S) <_\alpha gd(S')$.

Suppose α does not respect l. Since C is well-formed and α is a schedule of C, ${\alpha^l}_{[1]} \in \mathit{Acqs}(l)$, so there exists i such that either $\Lambda({\alpha^l}_{[i]}, l) = \text{ACQ}$ and $\Lambda({\alpha^l}_{[i+1]}, l) \neq \text{REL}$ or $\Lambda({\alpha^l}_{[i]}, l) \neq \text{ACQ}$ and $\Lambda({\alpha^l}_{[i+1]}, l) = \text{REL}$. Let $S = S_l({\alpha^l}_{[i]})$ and $S' = S_l({\alpha^l}_{[i+1]})$.

In the first case, ${\alpha^l}_{[i]} = \mathit{gd}(S) \neq \mathit{rgd}(S)$ and ${\alpha^l}_{[i+1]} = \mathit{gd}(S')$ by Corollary 7.12, since ${\alpha^l}_{[i]}, {\alpha^l}_{[i+1]} \in \mathit{Acqs}(l)$ and ${\alpha^l}_{[i]} \notin \mathit{Rels}(l)$. Since $\mathit{gd}(S) <_{\alpha^l} \mathit{rgd}(S)$, we have $\mathit{gd}(S) <_\alpha \mathit{gd}(S') \leq_\alpha \mathit{rgd}(S)$.

In the second case, ${\alpha^l}_{[i]} = \mathit{rgd}(S)$ and ${\alpha^l}_{[i+1]} = \mathit{rgd}(S') \neq \mathit{gd}(S')$ by Corollary 7.12, since ${\alpha^l}_{[i]}, {\alpha^l}_{[i+1]} \in \mathit{Rels}(l)$ and ${\alpha^l}_{[i+1]} \notin \mathit{Acqs}(l)$. Since $\mathit{gd}(S') <_{\alpha^l} \mathit{rgd}(S')$ and $\mathit{gd}(S) \neq \mathit{gd}(S')$, we have either $\mathit{gd}(S') <_\alpha \mathit{gd}(S) \leq_\alpha \mathit{rgd}(S) <_\alpha \mathit{rgd}(S')$ or $\mathit{gd}(S) <_\alpha \mathit{gd}(S') \leq_\alpha \mathit{rgd}(S) <_\alpha \mathit{rgd}(S')$. ■

It follows immediately from the previous lemma that two schedules that respect locking have the same acquire-release schedule as long as they consistently order the acquires; in an acquire-release schedule, a release, if it is a separate operation, immediately follows its matching acquire.

Corollary 7.15 For $C \in \mathit{WF}_L$ and $\alpha, \beta \in \mathit{RespLock}(C)$, if $\alpha|_{\mathit{Acqs}(l)} = \beta|_{\mathit{Acqs}(l)}$ then $\alpha^l = \beta^l$.

If a schedule respects locking, then any operation that holds a lock l and is scheduled between the guard and rear guard of an l-section is in that l-section. That is, l-sections do not overlap in a schedule that respects locking.

Lemma 7.16 For $C \in \mathit{WF}_L$, if $\alpha \in \mathit{RespLock}(C)$, S is an l-section of C for some $l \in L$, and $x \in \mathit{Holds}(l)$ then $x \in S \iff \mathit{gd}(S) \leq_\alpha x \leq_\alpha \mathit{rgd}(S)$.

Proof: If $x \in S$ then by the definition of $\mathit{gd}(S)$ and $\mathit{rgd}(S)$, $\mathit{gd}(S) \preceq_C x \preceq_C \mathit{rgd}(S)$, and since α is a schedule of C, $\mathit{gd}(S) \leq_\alpha x \leq_\alpha \mathit{rgd}(S)$.

If $x \notin S$ then $\mathit{gd}(S_l(x)) \neq \mathit{gd}(S)$. If $\mathit{gd}(S_l(x)) <_\alpha \mathit{gd}(S)$ then by Lemma 7.14, $x \leq_\alpha \mathit{rgd}(S_l(x)) <_\alpha \mathit{gd}(S)$. If $\mathit{gd}(S) <_\alpha \mathit{gd}(S_l(x))$. then by again Lemma 7.14, $\mathit{rgd}(S) <_\alpha \mathit{gd}(S_l(x)) \leq_\alpha x$. ■

The previous lemma implies that if two operations are in different l-sections, then the rear guard of the first operation is scheduled before the guard of the second operation.

Corollary 7.17 For $C \in \mathit{WF}_L$, $\alpha \in \mathit{RespLock}(C)$, $l \in L$ and $x, y \in \mathit{Holds}(l)$, if $S_l(x) \neq S_l(y)$ and $x <_\alpha y$ then $\mathit{rgd}(S_l(x)) <_\alpha \mathit{gd}(S_l(y))$.

Proof: Since α is a schedule of C, which is well-formed, $x <_\alpha y \leq_\alpha \mathit{rgd}(S_l(y))$. Since $x \notin S_l(y)$, by the previous lemma, $x <_\alpha \mathit{gd}(S_l(y))$. Again, since $\mathit{gd}(S_l(x)) \leq_\alpha x$ and $\mathit{gd}(S_l(y)) \notin S_l(x)$, we have by the previous lemma, $\mathit{rgd}(S_l(x)) <_\alpha \mathit{gd}(S_l(y))$. ■

Respecting locking is a "monotonic" property in that adding precedence dependencies does not introduce any new schedules that respect locking.

Lemma 7.18 If C is stricter than C' then $\mathit{RespLock}(C) \subseteq \mathit{RespLock}(C')$.

Proof: If $\alpha \in \mathit{RespLock}(C)$ then $\alpha \in \mathit{Sch}(C) \subseteq \mathit{Sch}(C')$ respects locking, so $\alpha \in \mathit{RespLock}(C')$. ■

Also, requiring a system to respect locks will not get the system "stuck", provided the clients specify a well-formed computation. That is, every well-formed computation has some schedule that respects locking.

Lemma 7.19 For $C \in \mathit{WF}_L$, $\mathit{RespLock}(C) \neq \emptyset$.

Proof: By Lemma 7.7, there exists $\alpha \in Sch(C)$ such that every enclosure of C appears contiguously in α. We show $\alpha \in RespLock(C)$; that is, for all $l \in L$, α respects l.

Let S and S' be any l-sections of C such that $gd(S) <_\alpha gd(S')$. By Lemma 7.8, $gd(S') \notin S$. Since every enclosure appears contiguously in α and S is an enclosure, $x <_\alpha gd(S')$ for any $x \in S$, and in particular, $rgd(S) <_\alpha gd(S')$. Thus, by Lemma 7.14, α respects l. ■

The definition for respecting locking formally expresses the intuitive semantics of locks, of which there are several variants. We focus primarily on mutex locks, that is, locks that guarantee mutual exclusion. Other kinds of locks, which allow more concurrency than mutex locks, require different definitions for what it means to respect them. Many of the results in this and later sections have obvious analogies for other kinds of locks. In Section 7.9, we consider one variant, *shared/exclusive locks*. For now, however, we restrict our attention to mutex locks.

7.5 Memories with Locks

In this section, we define several computation-centric models for memory systems with locks. Like the memory models defined in Chapter 5, these models are unifications and generalizations of many models or implementations of systems with locks [16, 49, 96, 25], because of the computation-centric framework in which they are defined. We first define *sequential consistency with locking*. Then we define weaker models that allow return values to be explained by different schedules. The models differ in how they synchronize lock accesses, ranging from *generic locking*, which does not guarantee any synchronization, to *strong sequential locking*, in which every lock access synchronizes with every other operation. The most important of these weak models for our purposes is *weak sequential locking*, which synchronizes lock access only with other accesses to the same lock.

A memory system with locks guarantees that the values returned for operations are generated by schedules that respect locking. The strongest memory model we consider simply adds the locking constraint to sequential consistency. Thus, all the return values are explained by a single schedule. Formally, the memory model for ***sequential consistency with locking*** is:

$$SCL = \{(C, \Phi) : \exists \alpha \in RespLock(C), \alpha \text{ explains } \Phi\}$$

Of course, sequential consistency with locking implements sequential consistency.

Lemma 7.20 *SCL* implements *SC*.

Proof: Immediate from the definitions of *SCL* and *SC*, since $RespLock(C) \subseteq Sch(C)$ for any C. ■

Because locks were introduced to structure the programs of early concurrent systems, which were sequentially consistent, sequential consistency with locking is the model programmers typically assume.

The constraints imposed by locks have two components. First, as mentioned above, the schedules used to generate return values are restricted to schedules that respect locking. Second, access to the locks is typically synchronized; that is, the system schedules the lock accesses consistently. We want to minimize the synchronization because it is expensive. To explore this space, we define a model that only restricts the schedules, but

does not synchronize any operations, and then strengthen this model with varying levels of synchronization.

The base model that respects locking is the ***generic locking*** memory model. Formally,

$$GL = \{(C, \rho) : \forall x \in V_C, \exists \alpha \in \mathit{RespLock}(C), \rho(x) = \mathit{retval}(x, \alpha)\}.$$

Except that only schedules that respect locking are used to explain the return values, this definition is identical to the definition from Section 5.1 of generic memory *GM*, the weakest of all memory models. Without synchronization, generic locking is too weak to implement critical sections. We use it as the basis for memory models with locks that synchronize.

Example 7.14 The return value functions defined below are all the observer functions for the computation from Example 7.7.

x	$\mathsf{write}(x,1)$	$\mathsf{read}(x)_1$	$\mathsf{write}(x,2)$	$\mathsf{write}(x,3)$	$\mathsf{read}(x)_2$
$\rho(x)$	ACK	1	ACK	ACK	3
$\rho'(x)$	ACK	3	ACK	ACK	2
$\rho''(x)$	ACK	1	ACK	ACK	2
$\rho'''(x)$	ACK	3	ACK	ACK	3

All these observer functions are also admissible according to *GL*. Only ρ and ρ' are admissible according to *SCL*. ■

Example 7.15 *GL* and *SCL* admit observations for computations that are not well-formed, as long as they have schedules that respect locking. The observer functions for the computation from Example 7.13 are:

x	$\mathsf{write}(x,1)$	$\mathsf{write}(x,2)$	$\mathsf{write}(x,3)$	$\mathsf{read}(x)$
$\rho(x)$	ACK	ACK	ACK	3
$\rho'(x)$	ACK	ACK	ACK	2

Although the computation is not well-formed, *GL* and *SCL* admit ρ, but not ρ', for it. ■

Although the generic locking memory provides very weak guarantees, it is incomparable to, not weaker than, sequential consistency.

Lemma 7.21 *GL* is incomparable to *SC*.

Proof: To see that $GL \not\subseteq SC$, consider the observer function ρ''' from Example 7.14. *GL* admits ρ''' for the computation from Example 7.7, but *SC* does not.

To see that $SC \not\subseteq GL$, consider the observer function ρ' from Example 7.15. *SC* admits ρ' for the computation from Example 7.13, but *GL* does not. ■

We model synchronization using synchronizing transformations, which are formally defined in Section 5.5. Synchronized operations are ordered consistently in every schedule that the system uses to generate return values for operations. Synchronizing transformations are defined using a synchronization predicate, which indicates whether two operations must be synchronized. The transformation adds precedence dependencies to the specified computation so that operations that must be synchronized according to the predicate are ordered by the computation. Thus, synchronized operations appear in the

same order in any schedule of a transformed computation. That is, the synchronizing transformation defined by a synchronization predicate Ψ is

$$\mathcal{T}^{s}_{\Psi} = \left\{ (C, C') : \begin{array}{l} V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge ann_C = ann_{C'} \\ \wedge\ (\Psi(x, y, C) \implies x \preceq_{C'} y \vee y \preceq_{C'} x) \end{array} \right\}.$$

Sequential consistency with locking is the same as generic locking with all operations synchronized.

Lemma 7.22 $SCL = \mathcal{T}^{s}_{True}(GL)$.

Proof: If $(C, \rho) \in SCL$ then there exists $\alpha \in Sch(C)$ that respects locking and explains ρ. Consider the computation $C' \in \mathcal{T}^{s}_{True}[C]$ that is the same as C except that $E_{C'} = <_{\alpha}$. We have $(C', \rho) \in GL$ because $\alpha \in Sch(C')$.

If $(C, \rho) \in \mathcal{T}^{s}_{True}(GL)$ then there exists $C' \in \mathcal{T}^{s}_{True}[C]$ such that $(C', \rho) \in GL$. By the definition of $\mathcal{T}^{s}_{True}$, $V_{C'}$ is totally ordered by $\preceq_{C'}$, so C' has only one schedule. Because $(C', \rho) \in GL$, the lone schedule α respects locking and $\rho(x) = retval(x, \alpha)$ for all $x \in V_{C'}$, so α explains ρ. Thus, $(C, \rho) \in SCL$. ■

We now define three models that specify intermediate levels of synchronization: *weak sequential locking*, *sequential locking*, and *strong sequential locking*. Informally, sequential locking and strong sequential locking are defined by designating every lock access as a synchronization operation. Most shared memory models with locks are defined in this way [2, 26]. Sequential locking guarantees that all synchronization operations are synchronized with each other, which corresponds to weak synchronization from Example 5.1. Strong sequential locking requires that the synchronization operations also synchronize with nonsynchronization operations, corresponding to strong synchronization from Example 5.2. Weak sequential locking relaxes the guarantees of sequential locking by requiring lock accesses to synchronize on a per-lock basis only; accesses to the same lock must be synchronized, but accesses to different locks need not be.

We present these models from strongest to weakest. As we shall see, the results that we prove in this thesis depend only on the guarantees of weak sequential locking. Thus, the formal definitions of sequential locking and strong sequential locking can be skipped without loss of continuity.

Most weakly consistent multiprocessor systems have special synchronization operations that provide stronger consistency guarantees than the ordinary operations. When locks are implemented on such systems, the lock accesses are synchronization operations. Different systems provide different guarantees for synchronization operations. For example, the weak ordering model from Example 6.10 requires every operation to synchronize with the synchronization operations [32]. This corresponds with strong synchronization from Example 5.2. The model resulting from implementing locks on strongly synchronized systems is strong sequential locking.

Formally, an operation acquires or releases a lock $l \in L$ if $\Lambda(x, l) \neq \text{NIL}$. So the strong sequential locking synchronization predicate is

$$\Psi_{SSL}(x, y, C) \equiv (\exists l \in L, \Lambda_C(x, l) \neq \text{NIL}) \vee (\exists l' \in L, \Lambda_C(y, l') \neq \text{NIL}).$$

That is, two operations synchronize if either acquires or releases any lock. The ***strong***

sequential locking memory model is $SSL = \mathcal{T}^{s}_{\Psi_{SSL}}(GL)$.

In other systems, the synchronization operations synchronize only with each other, corresponding with weak synchronization from Example 5.1. Release consistency [43], for example, specifies this level of synchronization. The model resulting from implementing locks with weak synchronization, where two operations synchronize if they both access some lock (not necessarily the same one), is sequential locking. Formally, the ***sequential locking*** memory model is $SL = \mathcal{T}^{s}_{\Psi_{SL}}(GL)$, where

$$\Psi_{SL}(x, y, C) \equiv (\exists l \in L, \Lambda_C(x, l) \neq \text{NIL}) \wedge (\exists l' \in L, \Lambda_C(y, l') \neq \text{NIL}).$$

For the results we prove in this thesis, it suffices to synchronize access to a lock only with other accesses to the same lock. That is, two operations synchronize if there is some lock that they both access. This allows different locks to be maintained by parts of a distributed system that do not communicate with each other. Formally, the ***weak sequential locking*** memory model is $WSL = \mathcal{T}^{s}_{\Psi_{WSL}}(GL)$, where

$$\Psi_{WSL}(x, y, C) \equiv \exists l \in L, (\Lambda_C(x, l) \neq \text{NIL} \wedge \Lambda_C(y, l) \neq \text{NIL}).$$

Example 7.16 Consider the observer functions from Example 7.14. *WSL*, *SL* and *SSL* all admit ρ and ρ', but not ρ'' and ρ''', for the computation from Example 7.7. ■

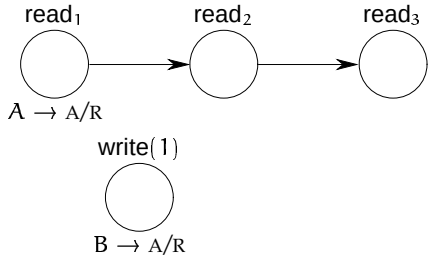
As the names suggest, strong sequential locking implements sequential locking, which implements weak sequential locking. Sequential consistency with locking is stronger than all of these models.

Lemma 7.23 *$SCL \subseteq SSL \subseteq SL \subseteq WSL$.*

Proof: This lemma follows immediately from Lemma 5.8 because $SCL = \mathcal{T}^{s}_{True}(GL)$ by Lemma 7.22 and $\Psi_{WSL}(x, y, C) \implies \Psi_{SL}(x, y, C) \implies \Psi_{SSL}(x, y, C)$. ■

None of these models are the same: Sequential consistency with locking guarantees sequential consistency even when no locks are accessed; all three sequential locking variants are equivalent to generic memory in this case. We can see that the sequential locking variants differ from each other in the following example:

Example 7.17 Consider the following computation on $\mathcal{R}$ (from Example 2.1) with locks $\{A, B\}$:



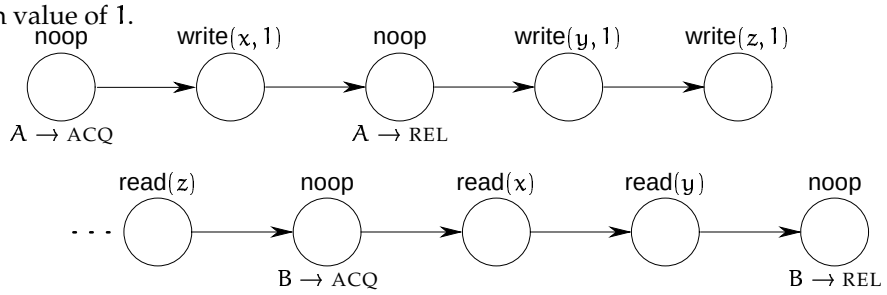
x	read1	read2	read3	write(1)
$\rho(x)$	1	0	0	ACK
$\rho'(x)$	0	1	0	ACK

WSL admits both ρ and ρ' for this computation; *SL* admits only ρ'; *SSL* does not admit either of these observer functions. ■

Example 7.18 Consider the following program:

```
                x,y,z = 0

P1: Acquire(A)            P2: repeat{t <- read(z)}
    write(x,1)                  until (t = 1)
    Release(A)                Acquire(B)
    write(y,1)                a <- read(x)
    write(z,1)                b <- read(y)
                              Release(B)
```

On a system that doesn't allow reordering, the program generates the following computation, except that there may be multiple instances of the read(z) operation. The last one, we know, gets a return value of 1.



If the return value for read(z) is 1, then the only schedule that explains this value orders all the operations of P1 before any operations of P2. Consider the observer functions for this computation with the following return values for the read operations:

x	read(z)	read(x)	read(y)
$\rho(x)$	1	1	1
$\rho'(x)$	1	1	0
$\rho''(x)$	1	0	1
$\rho'''(x)$	1	0	0

If this program on a memory system that guarantees only *WSL*, it may observe any of these observer value functions. Running on *SL*, the read(x) and read(y) are after the acquire of B. Since write$(x, 1)$ appears before a release of A, and the release of A and acquire of B are synchronized, the read(x) must see the write. Because the write$(y, 1)$ does not have any synchronization operation after it, its value need not propagate throughout the system, so read(y) can get 0. Thus, *SL* admits both ρ and ρ'. However, *SSL* requires all operations to be ordered with respect to the synchronization operations, so it only admits ρ. ■

Although it may seem a natural relaxation to synchronize lock accesses on a per-lock basis, few multiprocessor systems allow this. The only ones we are aware of are the scope consistency and entry consistency models [59, 15], and a proposed memory model for Java [94].[1]

7.6 Data Races Under Locking

Because a memory with locks synchronizes lock accesses, locks are used to resolve races in computations. Operations that do not hold any lock in common, however, may form

[1]The current memory model for Java [49] is similar to sequential locking with coherence.

unsynchronized races, or *data races*. In this section, we formally define data races for memories with locks, and prove that clients that avoid data races can assume sequential consistency when running on a system that guarantees only weak sequential locking.

A system guarantees that the locks are acquired and released in a consistent way, using the lock accesses as synchronization operations. Because locked sections for the same lock do not overlap in any schedule that respects locking, competing operations in different locked sections of the same lock cannot actually be concurrent. We say that the lock *arbitrates* the race. Operations that are not arbitrated are not synchronized; if they compete, they form a data race.

Formally, two operations of a well-formed computation with locks L ***share a lock*** $l \in L$ if they both hold l. They are ***arbitrated by*** l if they share l and are not in the same l-section. Two operations ***compete under locking*** if they compete and are not arbitrated by any lock; they form a ***data race under locking***. A well-formed computation is ***data-race-free under locking*** if no operations compete under locking, that is, if all races are arbitrated.

A program is ***data-race-free under locking*** if it generates only computations that are data-race-free under locking. The client restriction for data-race-freedom under locking is:

$$DRF_L = \{C : C \text{ is data-race-free under locking}\}.$$

If a computation is data-race-free under locking, then schedules that consistently order the accesses to each lock are strongly equivalent.

Lemma 7.24 If $C \in DRF_L$, $\alpha, \beta \in RespLock(C)$ and for all $l \in L$, $\alpha^l = \beta^l$, then $\alpha \equiv_{str} \beta$.

Proof: The relation $E_C \cup \bigcup_{l \in L} <_{\alpha^l}$ is a strict partial order because it is contained in $<_\alpha$. Let C' be the computation such that $V_{C'} = V_C$ and $E_{C'} = E_C \cup \bigcup_{l \in L} <_{\alpha^l}$. Since $\alpha^l = \beta^l$ for all $l \in L$, both α and β are schedules of C'. We show that C' is completely race-free, which implies that $\alpha \equiv_{str} \beta$.

For $x, y \in V_C$ such that $x <_\alpha y$, if x and y compete in C then they are protected by some lock l, since C is date-race-free under locking. So x and y both hold l and $x \notin S_l(y)$. By Corollary 7.17, $rgd(S_l(x)) <_\alpha gd(S_l(y))$. Because C is well-formed, $rgd(S_l(x)) \in Rels(l)$ and $gd(S_l(y)) \in Acqs(l)$, so $(rgd(S_l(x)), gd(S_l(y))) \in E_{C'}$, and $x \preceq_{C'} rgd(S_l(x)) \prec_{C'} gd(S_l(y)) \preceq_{C'} y$. So x and y do not compete in C'. Thus, C' is completely race-free. ■

We now prove the main result of this chapter: When clients are data-race-free under locking, weak sequential locking implements sequential consistency.

Theorem 7.25 *WSL* implements *SCL* under DRF_L.

Proof: If $(C, \rho) \in WSL|_{DRF_L}$ then $C \in DRF_L$ and $(C', \rho) \in GL$ for some $C' \in \mathcal{T}^s_{\Psi_{WSL}}[C]$, so for every $x \in V_{C'}$, $\rho(x) = retval(x, \alpha)$ for some $\alpha \in RespLock(C') \subseteq RespLock(C)$. Let $\alpha \in RespLock(C)$ be a schedule that explains the return value of some operation $x \in V_C$. We show that for all $y \in V_C$, $\rho(y) = retval(y, \alpha)$.

Let $\beta \in RespLock(C')$ be the schedule that explains $\rho|_{\{y\}}$. For all $l \in L$, $Acqs(l) \cup Rels(l)$ is totally ordered by $\preceq_{C'}$, so $\alpha^l = \beta^l$. By Lemma 7.24, $\alpha \equiv_{str} \beta$, since $C \in DRF_L$, so $\rho(y) = retval(y, \alpha)$. Thus, α explains ρ, and $(C, \rho) \in SCL$. ■

It follows immediately that all the sequential locking variants are equivalent to sequential consistency with locking under data-race-free clients.

Corollary 7.26 *WSL*, *SL*, *SSL* and *SCL* are all equivalent under DRF_L.

Proof: Immediate from Theorem 7.25 and Lemmas 7.23, 4.3 and 4.4. ■

Theorem 7.25 is a crucial result for using weakly consistent memories that have locks. It is similar to Theorem 5.4, which requires a programmer to write only completely race-free programs. However, complete race-freedom is a strict discipline, suitable for "embarrassingly parallel" problems or systems that can specify barrier synchronization and provide it cheaply. Depending on the facilities of the programming system, it may be difficult to structure a program so that it is completely race-free. In contrast, data-race-freedom under locking is the recommended style for programming concurrent systems with locks [22, 16, 51, 70], even when the system guarantees sequential consistency. Data races are usually considered bugs, and there is a rich body of research on algorithms to detect data races in systems with locks [31, 96, 25, 24], again often on systems that guarantee sequential consistency. Theorem 7.25 says that if you write a program that is data-race-free under locking for a sequentially consistent system, then you can run the program *unchanged* on a system that provides only weak sequential locking (assuming the systems have the same interface).

An alternative proof for Theorem 7.25 builds directly on the proof for Theorem 5.4. It uses the following lemma, which says that with weak sequential locking, a data-race-free computation is completely race-free, or else does not have any schedules that respect locking.

Lemma 7.27 If $C \in DRF_L$ and $C' \in \mathcal{T}^{s}_{\Psi_{WSL}}[C]$ then either $RespLock(C') = \emptyset$ or $C' \in RF$.

Proof: Suppose $RespLock(C') \neq \emptyset$. Let $\alpha \in RespLock(C') \subseteq RespLock(C)$. Consider $x, y \in V_{C'}$. Assume, without loss of generality, that $x <_\alpha y$. If x and y do not compete in C then they do not compete in C' since $E_C \subseteq E_{C'}$. If x and y compete in C then, since $C \in DRF_L$, they are arbitrated by some lock $l \in L$; that is, $x, y \in Holds(l)$ and $S_l(x) \neq S_l(y)$. ($S_l(x)$ and $S_l(y)$ are l-sections and enclosures of C.) Let $x' = rgd_C(S_l(x))$ and $y' = gd_C(S_l(y))$. By Corollary 7.17, $x' <_\alpha y'$. Because $E_C \subseteq E_{C'}$, $x \preceq_{C'} x'$ and $y' \preceq_{C'} y$. Since $x' \in Rels(l)$, $y' \in Acqs(l)$ and $C' \in \mathcal{T}^{s}_{\Psi_{WSL}}[C]$, we have either $x' \prec_{C'} y'$ or $y' \prec_{C'} x'$. The latter is not possible because $x' <_\alpha y'$ and $\alpha \in Sch(C')$. So $x \preceq_{C'} x' \prec_{C'} y' \preceq_{C'} y$, and thus, x and y do not compete in C'. ■

Theorem 7.25 follows immediately from this lemma, because if $(C, \rho) \in WSL|_{DRF_L}$ then $(C', \rho) \in GL$ for some $C' \in \mathcal{T}^{s}_{\Psi_{WSL}}[C]$. So $RespLock(C') \neq \emptyset$, and thus, $C' \in RF$. By Lemma 5.3, *GL* and *SCL* are equivalent under *RF*, so $(C', \rho) \in SCL$, and thus, $(C, \rho) \in SCL$.

7.7 Locks vs. Direct Synchronization

The literature contains many results similar or related to the results of the previous section [43, 45, 5, 11, 70, 59]. The most widely cited result, by Gharachorloo, et al. [43], says that release consistency guarantees sequential consistency for "properly labeled" programs. Much of the work on similar results has been done in the context of programmer-centric memory models, which we discuss in Section 6.7. In this section, we discuss the differences between the programmer-centric results in that section and the results of this chapter.

There are two basic modeling differences. First, the results of the previous chapter assume a processor-centric view of memory; the results of this chapter do not. That is, the precedence dependencies may define an arbitrary partial order on the operations as

long as the locked sections form enclosures in the computation, and the data type of the memory may be arbitrary, rather than being restricted to a simple read/write memory. This flexibility in the partial order allows us to model systems such as in Cilk [105] and Java [49], which do not have a fixed assignment of computation to processors.[2] Second, we model locks explicitly in the computation rather than specifying the synchronization to implement them. The data-race-free computations of the programmer-centric models implement locking using synchronized reads and writes on the memory, and they encode the order of these synchronization operations, and thus the order in which the locks are acquired, using precedence dependencies. We discuss the problems of using precedence dependencies to model synchronization in Section 5.5. The computations in this chapter have more abstract information and look more similar to a high level program.

In addition to differing in the approach to modeling, the results in this chapter also differ from the results about programmer-centric models in the level of synchronization required of the system. Theorem 6.13 uses weak ordering, which is similar to—even slightly stronger than—strong sequential locking. We could also have used release consistency, which is similar to sequential locking. In either case, every operation that accesses a lock is synchronized with every other operation that accesses a lock, even if they access different locks. Theorem 7.25 uses weak sequential locking, which requires operations to be synchronized only if they access the same lock, and thus admits more efficient implementations. More complicated programmer-centric models, such as the data-race-free-1 and properly-labeled-2 models [5, 41], distinguish additional types of operations, such as *unpaired synchronization* or *nsync* operations, which only synchronize with conflicting synchronization operations, allowing them to handled more efficiently than ordinary synchronization operations. However, nsync operations cannot be used to implement locks or synchronize access to data by "ordinary" operations, so they have limited utility.

Finally, because we model locks explicitly, Theorem 7.25 guarantees sequential consistency *with locking*, not just sequential consistency, for computations that are data-race-free under locking. Because the theorems about programmer-centric models in the literature guarantee only sequential consistency, we would need to reason separately about the locking guarantees.

7.8 Locks and Locations

For a memory with locations, there is a simple *locking discipline* that guarantees data-race-freedom under locking: Associate a lock with each location in memory, and access the location only when the lock is held. This locking discipline is widely used, or at least widely recommended [16, 70]; various forms of it appear in the early literature on concurrent programming and transaction processing [28, 57, 22, 35]. In this section, we express this discipline in the computation-centric framework, and show that, together with some implicit assumptions about the system, it guarantees data-race-freedom under locking.

[2]The current specification of Java's memory model [49], as well as the proposals to replace it [82, 94, 46], are processor-centric; they assume that there is a virtual processor for each thread. Pugh and Manson propose to model the dependencies implied by dynamic thread creation as an imaginary synchronized (volatile) memory access [94].

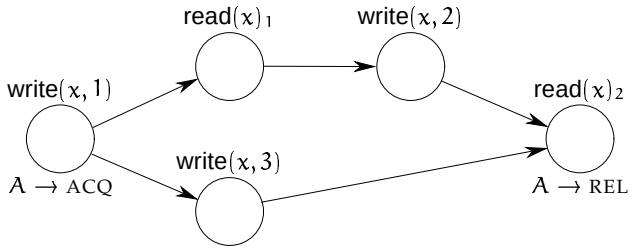
There are many variations, usually relaxations, of this discipline. We discuss some of them briefly at the end of this section.

A lock $l \in L$ is ***associated with*** a location $\ell \in \mathcal{L}$ in a well-formed computation $C \in WF_L$ if every operation performed on ℓ holds l; that is, $V_C|_\ell \subseteq Holds_C(l)$. A computation ***locks all locations*** if every location has an associated lock. We characterize clients that lock all locations by $LockLocs = \{C \in WF_L : \forall \ell \in \mathcal{L}, \exists l \in L, l \text{ is associated with } \ell\}$. Locking all locations is the basic locking discipline used to avoid data races under locking. The dynamic data race detector Eraser [96] checks that this discipline is obeyed, reporting violations as potential data races.

Example 7.19 The computation from Example 7.10 locks all locations; A is associated with x and B is associated with y. ■

The discipline of locking all locations originated in the context of systems that did not expect concurrency within a locked section. Locking all locations does not "protect" against races within a single locked section. We avoid such races by requiring the computations to satisfy the serial locked sections restriction from Section 7.3: $SerLockSec = \{C \in WF_L : \text{every locked section of } C \text{ is serial}\}$.

Example 7.20 Consider the following computation with locks $\{A\}$. (This computation is the computation from Example 7.7 without the lock B.) Although A is associated with x, this computation is not data-race-free under locking.



■

Clients that lock all locations and have serial locked sections are data-race-free under locking.

Lemma 7.28 $LockLocs \cap SerLockSec \subseteq DRF_L$.

Proof: If $C \in LockLocs \cap SerLockSec$ and two operations compete in C then they must be performed on the same location. Since $C \in LockLocs$, they must both hold some lock l. They cannot be in the same l-section, because all l-sections in C are serial, and so do not contain races. Thus, the operations are in different l-sections, and are arbitrated by l. ■

From the previous lemma, it immediately follows that for clients that lock all locations and have serial locked sections, sequential locking implements sequential consistency.

Theorem 7.29 *WSL* implements *SCL* under $LockLocs \cap SerLockSec$.

Proof: Immediate from Theorem 7.25 and Lemma 7.28. ■

There are several ways we can relax the simple discipline of locking all locations, and still guarantee data-race-freedom. On a read/write memory, we can take advantage of the

semantics of the operations, recognizing that reads do not conflict. For example, we can associate several locks with each location, and require that a write hold all the associated locks, while a read hold at least one. Although we do not formalize this discipline, it is easy to see that it guarantees data-race-freedom under locking. Alternatively, we can use *read/write locks*, which provide weaker exclusion guarantees than mutex locks. We discuss read/write locks, also called *shared/exclusive locks*, in Section 7.9.

We can also relax this discipline by not requiring clients to acquire the lock for operations that are not in any races. For example, operations that initialize a location should be ordered before any other operations to that location. Also, some locations—*final variables* in Java [49], for example—are written only at initialization. Savage, et al. [96] extend Eraser to accommodate these two cases. We can generalize this discipline even further by allowing concurrency within locked sections, as long as every race is arbitrated by some lock.

Formally, a location $\ell \in \mathcal{L}$ is ***protected by*** $\mathtt{l} \in \mathtt{L}$ in a well-formed computation $\mathtt{C} \in WF_{\mathrm{L}}$ if every pair of competing operations on ℓ is arbitrated by $\mathtt{l}$. A computation ***protects all locations*** if every location is protected by some lock. Clients that protect all locations are characterized by $ProtLocs = \{\mathtt{C} \in WF_{\mathrm{L}} : \forall \ell \in \mathcal{L}, \exists \mathtt{l} \in \mathtt{L}, \mathtt{l} \text{ protects } \ell\}$.

Example 7.21 In the computation from Example 7.7, B protects x but A does not. Ironically, A is associated with x but B is not. ■

Lemma 7.30 *ProtLocs* $\subseteq DRF_L$.

Proof: Immediate from definition. ■

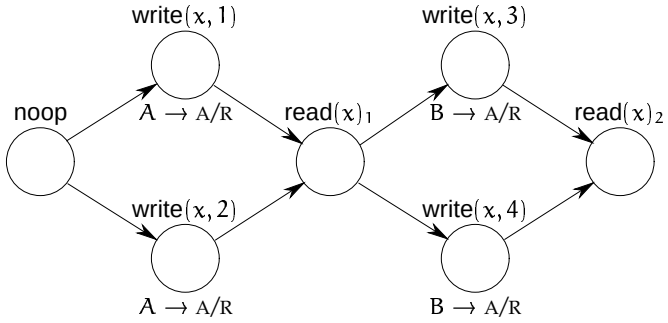
Theorem 7.31 *WSL* implements *SCL* under *ProtLocs*.

Proof: Immediate from Theorem 7.25 and Lemma 7.30. ■

The *umbrella locking discipline* of Cheng, et al. [25] is slightly less restrictive than *ProtLocs*, but still implies data-race-freedom under locking. Informally, the computation is divided into a sequence of phases, so that every operation in one phase logically precedes every operation in the next phase. The umbrella locking discipline allows a location to be "protected" by different locks in different phases. This discipline is used to efficiently detect potential races in Cilk computations. Cheng extends their algorithm to computations that allow concurrency within locked sections [24]. We do not formalize this discipline here.[3]

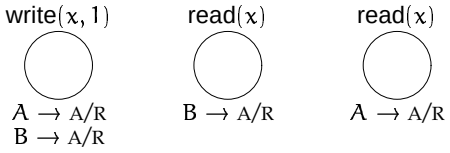
Example 7.22 The following computation is data-race-free under locking. It obeys the umbrella locking discipline.

[3] Its exact formalization requires computations to be representable as *series-parallel parse trees*, which holds for computations generated by Cilk programs but not for arbitrary well-formed computations.



■

Example 7.23 The following computation does not lock all locations, nor does it protect all locations or even obey the umbrella locking discipline. It is, however, data-race-free.



■

7.9 Shared/Exclusive Locks

In this chapter, we have focused on mutex locks, which are the simplest and strongest kind of locks. To allow greater concurrency, some systems provide more flexible locks [14, 51]. The results in this chapter can be easily extended to results about other kinds of locks. In this section, we discuss how to model the most common variant, *shared/exclusive locks*, which are often called *read/write locks*.

Shared/exclusive locks extend mutex locks by allowing a lock to be acquired in one of two modes, which the client must specify when it attempts to acquire the lock. A *shared acquire* acquires the lock in *shared mode*, while an *exclusive acquire* acquires the lock in *exclusive mode*. The well-formedness condition is the same as with mutex locks, except that acquires now specify whether they are shared or exclusive. The mode of a lock held by an operation in a locked section is determined by the operation that acquires the lock.

Shared/exclusive locks differ from mutex locks in how they constrain the schedules allowed by the system. A shared acquire may proceed as long as no thread holds the lock in exclusive mode. An exclusive acquire may proceed only if no thread holds the lock in either mode. Thus, several threads may hold the lock concurrently in shared mode, but if any thread holds the lock in exclusive mode, no other thread may hold it in either shared or exclusive mode.

This difference is reflected in the definition of what it means to respect locking. For a well-formed computation, a schedule *respects shared/exclusive locking* if

- the release corresponding to any shared acquire occurs before the next exclusive acquire of the same lock, and
- the release corresponding to any exclusive acquire occurs before the next acquire, shared or exclusive, of the same lock.

Several shared acquires may occur before any of the corresponding releases, but exclusive acquires may not occur until the releases corresponding to all previous acquires have occurred. This characterization of schedules that respect shared/exclusive locking is similar to the characterization in Lemma 7.14 of schedules that respect locking, except that a lock may be acquired multiple times in shared mode without being released. The condition for mutex locking is identical to the case in shared/exclusive locking where all lock acquisitions are exclusive.

Because the condition for respecting locking is relaxed, we also modify the definition of data-race-freedom under locking. In particular, two shared acquisitions of a lock no longer arbitrate a race. Two operations are *arbitrated by a shared/exclusive lock* if they are in different lock sections of the lock, and at least one operation holds the lock in exclusive mode. A *data race under shared/exclusive locking* is a race that is not arbitrated by any shared/exclusive lock. A computation is *data-race-free under shared/exclusive locking* if there are no data races under shared/exclusive locking.

With these definitions, we can modify the results from earlier sections of this chapter, so that they hold for shared/exclusive locking. In fact, the weak sequential locking model may be further relaxed so that it no longer needs to synchronize the shared lock accesses with each other. They must, of course, be synchronized with the exclusive lock accesses.

When the data type of the memory is read/write memory, each location can have an associated lock, as discussed in Section 7.8. If every write operation holds the lock in exclusive mode, and every read operation holds the lock in either shared or exclusive mode, and lock sections are serial, then the computation is guaranteed to be data-race-free under shared/exclusive locking. For this reason, these locks are often called read/write locks, with shared acquires called *read acquires*, and exclusive acquires called *write acquires*.

7.10 Discussion

In the literature, it is common to characterize the pattern of synchronization reads and writes needed to implement locking, describe the consistency guarantees of a system for these synchronization operations, and show that programs that follow the prescribed pattern run on the system as though it were sequentially consistent. However, locks guarantee more than just sequential consistency, and the programmer was left to reason that the exclusion constraints implied by locking were also guaranteed. Although this latter reasoning could be done assuming sequential consistency, experience has shown that even sequentially consistent systems are difficult to program correctly [70].

In this chapter, we model locks directly instead of modeling the synchronization mechanisms used to implement locking. In doing so, we adopt a more abstract view of memory systems with locks. The computations capture the programmer's intent, as expressed in the program. The memory model captures the properties that the programmer relies on when reasoning about a program. This more abstract view has the usual advantages of good abstraction. The analysis is simpler, corresponds more closely with the programmer's informal reasoning, and is more robust under changes to the program or the system. We can also easily extend our theory to model different kinds of locks. We take the approach of specifying the programmer's intent even further in the next chapter, where we show how to model *transactions*, which locks are often used to implement.

In many systems, locks are assumed to be held by a single processor, and no concurrency is allowed within a locked section. The use of computations makes this restriction unnecessary and also allows us to model systems that provide arbitrary data types.

Because we focus on locking, the models in this chapter capture only the properties needed to use locks effectively, decoupling them from properties such as coherence, which are typically also provided by memory systems. By separating these properties, we give system designers greater flexibility and control in defining the guarantees of their systems. In particular, the freedom from having to maintain coherence gives compiler writers much needed room for improving the code generated by reordering operations [93].

An obvious way to extend the work in this chapter is to study other kinds of locks and the conditions needed to use them effectively, or refine the conditions given in this chapter for mutex and shared/exclusive locks. Similarly, we can explore how to relax the well-formedness conditions for computations with locks. For example, Cheng [24] notes that in the Cilk programming system, operations that acquire locks need not guard their locked sections.

We might also want to allow locks to be passed in a more flexible way. For example, in some multithreaded languages [16, 49, 96], new threads may be created (spawned or forked) while a lock is being held, but "ownership" of a lock is not passed to newly created threads. We can model this behavior by assigning every vertex in the computation at most one *primary edge*, which indicates the flow of control of the parent thread. A lock must be released by the same thread that acquired it, and an operation holds a lock only if there is a path from an acquisition of the lock along primary edges and without any intervening releases of the lock.

Chapter 8

Transactions

Although locking is a high level synchronization primitive, it is primarily a mechanism to implement *critical sections*. A critical section is a section of code that is intended to appear as though it executes atomically; that is, operations outside the section do not see any state internal to the section. Thus, the critical section appears to be a single complex operation, often called a *transaction*. In this chapter, we define memories that allow clients to explicitly specify transactions rather than a mechanism that implements them. We show how these memories can be used and implemented. To our knowledge, no other frameork for specifying weak memory consistency handles transactions explicitly.

Informally, a transaction is a collection of operations with four "ACID" properties [51]: *atomicity*, *consistency*, *isolation* and *durability*. We are most concerned with isolation, also called *serializability* [14], which asserts that each transaction appears to execute alone, that is, without other operations executing concurrently. We also briefly consider consistency[1] or *integrity*, which requires transactions to maintain certain constraints on the state of the memory. Atomicity[2] and durability primarily address fault-tolerance, which we do not deal with in this thesis.

Transactions are a powerful concept for structuring concurrent programs, encouraging a style that has been proven to produce large fault-tolerant distributed systems, a "distributed system application development approach that can be used by mere mortals [51]." In particular, the ability to consider transactions in isolation supports abstraction and modularity, both laterally and hierarchically. The literature on transactions is vast; the discussion in this chapter barely scratches the surface.

In practice, transactions are usually implemented using *two-phase locking* [35]. We show how to model two-phase locking in the computation-centric framework, and, using a technique called *reserialization*, we prove it implements transactions. Although two-phase locking typically assumes sequential consistency, we show that systems that provide weak sequential locking, as defined in Chapter 7, are sufficient to guarantee the transactional behavior. We also show how to use a different technique called *program reduction* [73] to show that a collection of consecutive operations appears as a transaction.

[1] This property is distinct from the memory consistency we discuss in this thesis.

[2] Lynch, et al. [80] use atomicity in a much stronger sense, combining the atomicity, isolation, and durability properties above.

Transactions are expensive to implement because they inhibit concurrency between transactions that access the same data. To mitigate this cost, weaker models have been defined, that admit more efficient implementations but guarantee weaker "degrees of isolation". Unfortunately, it is difficult to characterize the guarantees of these systems. We define *internally consistent transactions*, which can be implemented efficiently, but which still provide isolation.

Outline: Section 8.1 defines computations with transactions, gives a well-formedness condition for these computations, and characterizes schedules that respect transactions. In Section 8.2, we define the basic transactional memory, *sequentially consistent transactional memory*. Section 8.3 introduces the technique of reserialization, which is used in Section 8.4, where we discuss two phase locking. We discuss program reduction in Section 8.5. In Section 8.6, we introduce weaker transactional memories, which we show share several useful properties of sequentially consistent transactions. In particular, in Section 8.7, we formally define integrity, and show how internally consistent transactions maintain it. In Section 8.8. we define what it means for two transactions to compete, and we give conditions under which the weaker transactional models are indistinguishable from sequentially consistent transactions. Section 8.9 discusses the work presented and the many future directions that are possible.

Reading Guide: This chapter is more the beginning of an exploration than a finished work. The first four sections parallel and build on the work on memory systems with locks in Chapter 7. Section 8.5 is an aside, exploring a rather different approach to reasoning about concurrent programs; it is independent of the other sections. Sections 8.6 through 8.8 are explore relaxed transactional models and the properties they guarantee. These sections are more tentative and speculative than the rest of this thesis because there are no real systems that have guarantees like the ones proposed here. We believe that further investigation of these and similar models would be very fruitful, and these sections are an attempt to point out some directions.

8.1 Computations with Transactions

A transaction is specified by indicating the operations that begin and end the transaction; it consists of the operations between its beginning and its end. Each transaction is assigned an identifier by the operation that begins it. A transaction should appear to occur atomically to operations outside the transaction. In this section, we define the formal mechanism to describe transactions in the computation-centric framework, and the well-formedness condition for computations with transactions. We also formally characterize the schedules that *respect transactions*, that is, the schedules in which the transactions appear atomic.

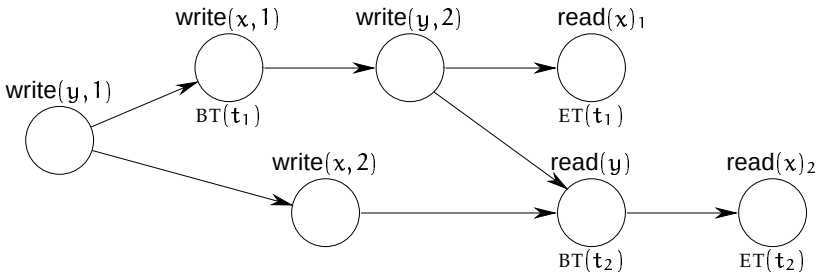
A system that supports transactions has a set TI of ***transaction identifiers***. An operation that begins a transaction is annotated by $\text{BT}(t)$, where $t \in TI$ is the identifier of the transaction. Similarly, the end of a transaction is annotated by $\text{ET}(t)$.

Formally, a ***computation with transactions*** is a computation with annotation set $A_T = \{\text{BT}(t) : t \in TI\} \cup \{\text{ET}(t) : t \in TI\}$. For a computation $C \in \mathfrak{C}^{\mathcal{D}}_{A_T}$, we denote the set of identifiers

used in C by $C.tids = \{t : ann_C(x) \in \{BT(t), ET(t)\}$ for some $x \in V_C\}$.

Example 8.1 The computation from Example 3.13 is a computation with transactions. ■

Example 8.2 Another computation with transactions:



■

We model the clients as assigning the transaction identifiers, placing on them the burden of maintaining unique identifiers. We could instead let the clients specify only that a transaction is beginning or ending, leaving the system to choose the identifier. Because there are no constraints on the form of the identifiers, there is little semantic difference between these choices. Requiring transaction identifiers to be explicit in the computation makes the well-formedness condition easier to state.

Well-formedness. As the terminology suggests, the identifier for each transaction must be unique, the operation that begins a transaction must precede the operation that ends it. Because a transaction should appear atomic, operations outside the transaction that precede any operation of the transaction must precede the beginning of the transaction. Similarly, operations outside the transaction that follow any operation of the transaction must follow the end of the transaction. Finally, a transaction cannot start or end inside another transaction.

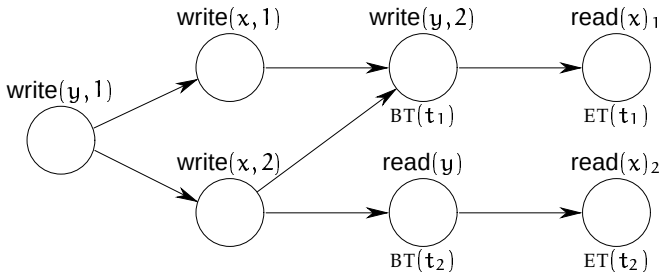
Formally, a computation $C \in \mathfrak{C}^{\mathcal{D}}_{A_T}$ with transactions is ***well-formed*** if for all $t \in C.tids$, the following conditions hold:

- there is a unique operation $bt_C(t) \in V_C$ such that $ann_C(bt_C(t)) = BT(t)$,
- there is a unique operation $et_C(t) \in V_C$ such that $ann_C(et_C(t)) = ET(t)$,
- $[bt_C(t), et_C(t)]$ is an enclosure of C, and
- $ann_C(x) = NIL$ for all $x \in (bt_C(t), et_C(t))$.

The first two clauses assert the uniqueness of transaction identifiers. The third clause expresses the next two informal conditions, which follow from the definition of an enclosure of a dag in Section 7.1; that is, that $bt_C(t)$ is a guard, and $et_C(t)$ is a rear guard, of the transaction. The last clause restricts our computations to "flat" transactions; that is, transactions are all disjoint. It is not difficult to extend this framework to nested transactions, but we do not do so in this thesis. We denote the set of well-formed computations with transactions by WF_T.

Example 8.3 The computation from Example 3.13 is well-formed; the one from Example 8.2 is not. ■

Example 8.4 The following computation with transactions is well-formed:



■

From now on, we consider only computations that are well-formed.

Transactions and isolated operations. For $C \in WF_T$ and $t \in C.tids$, the operations $bt_C(t)$ and $et_C(t)$ are well-defined, and the ***transaction*** of t is $Tr_C(t) = [bt_C(t), et_C(t)]$. Such transactions are ***nontrivial***; they have at least two operations, $bt(t)$ and $et(t)$. An operation that is not in the transaction of any transaction identifier used in C is ***isolated***, and the set of isolated operations is denoted by $Isol_C = V_C - \bigcup_{t \in C.tids} Tr(t)$. We omit the subscript when the computation is clear from context. Any singleton set containing an isolated operation is a ***trivial transaction***.

Example 8.5 The computation from Example 8.4 uses two transaction identifiers, t_1 and t_2, with transactions $Tr(t_1) = \{\mathsf{write}(y,2), \mathsf{read}(x)_1\}$ and $Tr(t_2) = \{\mathsf{read}(y), \mathsf{read}(x)_2\}$ respectively. It has three isolated operations, $\mathsf{write}(y,1)$, $\mathsf{write}(x,1)$ and $\mathsf{write}(x,2)$. ■

By definition, every operation is in some transaction, and the trivial transactions are disjoint from each other and from the nontrivial transactions. We show that the nontrivial transactions are also disjoint.

Lemma 8.1 If $C \in WF_T$ and $Tr_C(t) \cap Tr_C(t') \neq \emptyset$ then $t = t'$.

Proof: Since C is well-formed, $Tr(t)$ and $Tr(t')$ are enclosures of C, and by Lemma 7.6, $S = Tr(t) \cap Tr(t')$ is an enclosure with $gd(S) \in \{gd(Tr(t)), gd(Tr(t'))\}$. Without loss of generality, suppose $gd(S) = gd(Tr(t)) = bt(t)$. Then $bt(t) \in S \subseteq Tr(t')$. Since $ann_C(bt(t)) = \text{BT}(t)$, $ann_C(et(t')) = \text{ET}(t')$, and $ann_C(y) = \text{NIL}$ for $y \in Tr(t') - \{bt(t'), et(t')\}$, we have $bt(t) = bt(t')$. And since $ann_C(bt(t')) = \text{BT}(t')$, we have $t = t'$, as required. ■

The previous lemma implies that $\{Tr(t)\}_{t \in C.tids}$ partitions $V_C - Isol$; that is, the nontrivial transactions partition the non-isolated operations. We associate with each non-isolated operation, the identifier of the transaction that contains it; that is, $x.tid = t$ if $x \in Tr(t)$.[3] The ***transaction*** of $x \in V_C$ is $Tr(x) = Tr(x.tid)$ if $x \notin Isol$, or $Tr(x) = \{x\}$ if $x \in Isol$.

The transaction of any operation is an enclosure.

[3] An alternative way to specify transactions is to annotate each operation with the identifier of its transaction, or NIL if the operation is isolated. The well-formedness condition is that $Tr(t) = \{x : ann(x) = t\}$ is an enclosure of the computation for every transaction identifier t that annotates an operation.

Lemma 8.2 If $C \in WF_T$ and $x \in V_C$ then $Tr(x)$ is an enclosure of C.

Proof: If $x \notin Isol$ then $Tr(x) = Tr(x.tid) = [bt(x.tid), et(x.tid)]$, which is an enclosure because C is well-formed. Otherwise, $x \in Isol$, so $Tr(x) = \{x\}$, which is an enclosure by Lemma 7.2 ■

Often, each transaction is requested by a single sequential client, which we model as a client restriction. Formally, the ***serial transactions restriction*** is:

$$SerTr = \{C \in WF_T : \text{every transaction of } C \text{ is serial}\}.$$

Example 8.6 The computation from Example 8.4 satisfies the serial transactions restriction; the one from Example 3.13 does not. ■

Respecting transactions. We now formalize the informal semantics that a computation with transactions is intended to specify. Following our treatment of locking, we characterize the schedules that *respect transactions*. Informally, a transaction should appear to be performed atomically; that is, operations outside the transaction should not observe the effects of some but not all of the operations of the transaction, and operations within the transaction should not observe any interference by outside operations. This property is variously called *atomicity*,[4] *serializability* or *isolation* [14, 51, 80]. We model it by requiring the operations of a transaction to be scheduled consecutively; that is, they must appear contiguously in the schedule.

Formally, suppose C is a well-formed computation with transactions. A schedule $\alpha \in Sch(C)$ ***respects transactions*** if for all $t \in C.tids$, $Tr_C(t)$ appears contiguously in α. We denote the set of schedules that respect transactions by $RespTr(C)$.

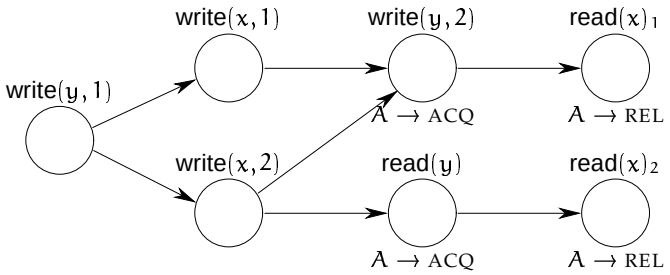
Example 8.7 The schedules of the computation from Example 8.4 that respect transactions are:

$$\mathsf{write}(y,1), \mathsf{write}(x,1), \mathsf{write}(x,2), \mathsf{write}(y,2), \mathsf{read}(x)_1, \mathsf{read}(y), \mathsf{read}(x)_2$$
$$\mathsf{write}(y,1), \mathsf{write}(x,1), \mathsf{write}(x,2), \mathsf{read}(y), \mathsf{read}(x)_2, \mathsf{write}(y,2), \mathsf{read}(x)_1$$
$$\mathsf{write}(y,1), \mathsf{write}(x,2), \mathsf{write}(x,1), \mathsf{write}(y,2), \mathsf{read}(x)_1, \mathsf{read}(y), \mathsf{read}(x)_2$$
$$\mathsf{write}(y,1), \mathsf{write}(x,2), \mathsf{write}(x,1), \mathsf{read}(y), \mathsf{read}(x)_2, \mathsf{write}(y,2), \mathsf{read}(x)_1$$
$$\mathsf{write}(y,1), \mathsf{write}(x,2), \mathsf{read}(y), \mathsf{read}(x)_2, \mathsf{write}(x,1), \mathsf{write}(y,2), \mathsf{read}(x)_1$$

There are sixteen schedules all together, so restricting to those that respect transactions significantly decreases the number of schedules to reason about. ■

Example 8.8 Because of isolated operations, merely acquiring a system-wide lock at the beginning of each transaction and releasing it at the end does not guarantee that a schedule that respects locking will also respect transactions. Consider the following modification of the computation from Example 8.4:

[4] In the transaction processing context, *atomicity* also refers to a related property, that either all or none of the effects of a transaction are seen. The concern in that context is that the transaction may fail to complete because of some fault in the system. We do not treat fault tolerance in this thesis, so the issue does not arise.



The following is a schedule of this computation, but not one of the original computation:

$$\mathsf{write}(y,1), \mathsf{write}(x,2), \mathsf{read}(y), \mathsf{write}(x,1), \mathsf{read}(x)_2, \mathsf{write}(y,2), \mathsf{read}(x)_1. \qquad \blacksquare$$

As the term "serializability" suggests, transactions appear not to execute concurrently. That is, in a schedule that respects transactions, if any operation of one transaction precedes any operation of another, then every operation of the first precedes every operation of the second.

Lemma 8.3 Suppose C is a well-formed computation with transactions, $\alpha \in \mathit{RespTr}(C)$, and $x, y \in V_C$ with $\mathit{Tr}(x) \neq \mathit{Tr}(y)$. If $x <_\alpha y$ then $x' <_\alpha y'$ for all $x' \in \mathit{Tr}(x)$ and $y' \in \mathit{Tr}(y)$.

Proof: Immediate from the definition of respecting transactions, because the operations of $\mathit{Tr}(x)$ appear contiguously in α, as do the operations of $\mathit{Tr}(y)$. $\blacksquare$

Every well-formed computation has some schedule that respects transactions.

Lemma 8.4 For $C \in \mathit{WF}_T$, $\mathit{RespTr}(C) \neq \emptyset$.

Proof: Immediate from Lemmas 7.7 and 8.2 $\blacksquare$

8.2 Sequentially Consistent Transactions

In this section, we define a *sequentially consistent transactional memory*. This model guarantees the traditional serializable transactions. Sequentially consistent transactions provide a powerful way to structure concurrent programs.

The definition of ***sequentially consistent transactional memory*** is analogous to that of sequential consistency with locking: Any observation of this model can be explained by a single schedule that respects transactions. Formally,

$$\begin{aligned} \mathit{SCT} &= \{(C,\rho) : \exists \alpha \in \mathit{RespTr}(C), \alpha \text{ explains } \rho\} \\ &= \{(C,\rho) : \exists \alpha \in \mathit{RespTr}(C), \forall x \in V_C, \rho(x) = \mathit{retval}(x,\alpha)\} \end{aligned}$$

This model is often called *serializable transactions* or *atomic transactions* [14, 80].

Obviously, a sequentially consistent transactional memory is sequentially consistent.

Lemma 8.5 *SCT* implements *SC*.

Proof: Immediate from the definitions since $\mathit{RespTr}(C) \subseteq \mathit{Sch}(C)$. $\blacksquare$

Every well-formed computation has some observation admissible according to *SCT*.

Lemma 8.6 *SCT* is complete under WF_T.

Proof: Immediate from the definitions and Lemma 8.4. ■

Although comparing the "naturalness" of different kinds of guarantees is difficult, sequentially consistent transactional memory seems to be an attractive model to program. Many techniques for structuring or reasoning about programs—critical sections [30], locking [14, 51], "pretending atomicity" [69], and others [80, 70]—are essentially ways to use the guarantees of this model.

The chief attraction of transactions is that they support modularity and abstraction: A programmer using a memory that guarantees sequentially consistent transactions does not need to consider possible interactions among transactions except at the beginning and end of each transaction. Thus, the correctness of one transaction does not depend on the correctness of other transactions, making it possible to build complex operations using a memory that supports a fairly simple data type. Unlike programming a memory with locks, there is no need for a programmer to obey a locking discipline to guarantee correct behavior; in particular, there is no need to keep track of which locks must be held to safely perform each operation.

Because the system must maintain the appearance of serializable transactions as well as sequential consistency, sequentially consistent transactions can be expensive to implement. They are often implemented using *two-phase locking*, a method we discuss in Section 8.4. For now, it suffices to note that this method often requires locks to be held for a long time, restricting the possible concurrency. Relaxations of sequentially consistent transactions have been defined to improve performance by permitting more concurrency. We discuss some relaxed transactional memory models in Section 8.6.

8.3 Reserialization

In this section, we introduce a technique called *reserialization* for proving that a schedule of a computation with transactions is equivalent to some schedule that respects transactions. This technique is used in the next section to prove that two-phase locking implements transactional memory, and it is necessary to understand the proofs in that section. It can be skipped initially and referred to as needed. Only the main result, Theorem 8.11, is needed outside of this section.

The idea behind reserialization is this: For each transaction, designate one operation as its *serialization point*. A schedule is *reserialized* by reordering the operations according to their serialization points, yielding a schedule that respects transactions. If conflicting operations have not been reordered, the resulting schedule is equivalent to the original schedule.

Reserialization is defined for any serialization of a set using a function that specifies the serialization point for each element of the set. Given a set X, a function $sp\colon X \to X$ and a serialization α of X, the ***reserialization*** of α according to sp, denoted $reserial_{sp}(\alpha)$, is a serialization β of X such that $x <_\beta y$ if and only if $sp(x) <_\alpha sp(y)$ or $sp(x) = sp(y)$ and $x <_\alpha y$. Informally, $sp(x)$ is the serialization point for x and $reserial_{sp}(\alpha)$ orders the elements of X according to their serialization points. Elements with the same serialization point retain their order from α. The set of serialization points is $range(sp)$.

Elements with the same serialization point are contiguous in the reserialization.

Lemma 8.7 If α is a serialization of X and $sp : X \to X$ then for all $x \in range(sp)$, the set $sp^{-1}(x)$ appears contiguously in $reserial_{sp}(\alpha)$.

Proof: Immediate from the definition of $reserial_{sp}(\alpha)$, since the elements are ordered first by their image under sp. ■

Given a computation, if the operations corresponding to each serialization point form an enclosure containing the serialization point then the reserialization of any schedule is also a schedule of the computation.

Lemma 8.8 For any computation C, if $\alpha \in Sch(C)$ and $sp : V_C \to V_C$ such that $sp^{-1}(x)$ is an enclosure of C with $x \in sp^{-1}(x)$ for all $x \in range(sp)$, then $reserial_{sp}(\alpha) \in Sch(C)$.

Proof: Let $\beta = reserial_{sp}(\alpha)$. Suppose that $(x, y) \in E_C$. If $sp(x) = sp(y)$ then $x <_\alpha y$ since $\alpha \in Sch(C)$, so $x <_\beta y$, as required. Otherwise, let $S = sp^{-1}(sp(x))$ and $S' = sp^{-1}(sp(y))$. Note that S and S' are disjoint enclosures with $sp(x) \in S$ and $sp(y) \in S'$. By Lemma 7.4 and the definitions of guard and rear guard, $sp(x) \preceq rgd(S) \prec gd(S') \preceq sp(y)$, so $sp(x) <_\alpha sp(y)$, so $x <_\beta y$, as required. ■

For a computation with transactions, if each transaction has a single serialization point within the transaction, then the reserialization of any schedule is a schedule that respects transactions.

Lemma 8.9 For $C \in WF_T$ and $sp : V_C \to V_C$ such that $sp^{-1}(sp(x)) = Tr(x)$ for all $x \in V_C$, if $\alpha \in Sch(C)$ then $reserial_{sp}(\alpha) \in RespTr(C)$.

Proof: Let $\beta = reserial_{sp}(\alpha)$. By the definition of sp, for all $x \in sp(V_C)$, $sp^{-1}(x) = Tr(x)$, which is an enclosure of C. Since $x \in Tr(x)$, by Lemma 8.8, $\beta \in Sch(C)$. By Lemma 8.7, $Tr(x)$ appears contiguously in β, so $\beta \in RespTr(C)$. ■

If conflicting operations are not reordered, that is, the order of their serialization points is consistent with their order, the reserialization is equivalent to the original serialization.

Lemma 8.10 Given a set X of operations and sp: $X \to X$, if α is a serialization of X such that $sp(x) \leq_\alpha sp(y)$ whenever x and y conflict and $x <_\alpha y$, then $reserial_{sp}(\alpha) \equiv_{str} \alpha$.

Proof: Let $\beta = reserial_{sp}(\alpha)$. If $x <_\alpha y$ and $y <_\beta x$, then $sp(y) <_\alpha sp(x)$. Thus, $sp(x) \not\leq_\alpha sp(y)$, so x and y do not conflict. By Theorem 2.16, $\beta \equiv_{str} \alpha$, as required. ■

We combine the previous two lemmas to get the main result of this section, which gives conditions under which a schedule of a computation with transactions can be reserialized to yield an equivalent schedule that respects transactions. The key to using this theorem is choosing a serialization point for each transaction so that conflicting operations of different transactions are ordered in the schedule consistently with their serialization points.

Theorem 8.11 For $C \in WF_T$ and $sp : V_C \to V_C$ such that $sp^{-1}(sp(x)) = Tr(x)$ for all $x \in V_C$, if $\alpha \in Sch(C)$ such that $sp(x) \leq_\alpha sp(y)$ whenever x and y conflict and $x <_\alpha y$, then there exists $\beta \in RespTr(C)$ such that $\beta \equiv_{str} \alpha$.

Proof: By Lemmas 8.9 and 8.10, $\mathit{reserial}_{sp}(\alpha) \in \mathit{RespTr}(C)$ and $\mathit{reserial}_{sp}(\alpha) \equiv_{\mathrm{str}} \alpha$. ■

Theorem 8.11 is a variant of the *serializability theorem* [35, 14], which is usually used to prove that a system implements sequentially consistent transactions. *Serializability* is the property that a schedule has an equivalent schedule that respects transactions, although the notion of equivalence used is even stronger than strong equivalence.[5] The serializability theorem gives a necessary and sufficient condition for the serializability of a schedule, which is equivalent to the condition of Theorem 8.11.

8.4 Two-Phase Locking

In practice, transactions are almost always implemented using *two-phase locking* [35], and in particular, a variant called *strict two-phase locking* [14]. A two-phase locking implementation consists of a *transaction manager*[6] and a sequentially consistent system with locking. The clients submit operations to the transaction manager, which adds lock accesses before propagating the operations to the underlying system. In this section, we show how to model the transaction manager as a transformation from computations with transactions to computations with the appropriate lock accesses. We also show that such a manager guarantees sequentially consistent transactions even when the underlying system only supports weak sequential locking.

Informally, a transaction manager for two-phase locking ensures that the computation executed on the underlying system is data-race-free under locking. In addition, once a lock has been released for a transaction, no new locks may be acquired. Thus, each transaction can be divided into two phases: a *growing phase*, during which locks are acquired, and a *shrinking phase*, during which locks are released, with the growing phase preceding the shrinking phase.

In the literature, two-phase locking is defined for systems in which transactions are serial, data-race-freedom is enforced by associating locks with *data items* or *entities* as described in Section 7.8, and locks are not held across transactions. Because these additional restrictions are not necessary for the correctness of two-phase locking, we omit them from our definition. We require only that the computation is data-race-free and that for each transaction, every acquire operation precedes every release operation. We allow the transaction manager to add precedence dependencies to enforce data-race-freedom.

Formally, a region S of a computation with locks L is ***two-phase*** if every operation in the region that acquires a lock precedes every operation in the region that releases any lock; that is, if $x \preceq y$ for all $x \in S \cap \bigcup_{l \in L} \mathit{Acqs}(l)$ and $y \in S \cap \bigcup_{l \in L} \mathit{Rels}(l)$. A ***two-phase locking transaction manager*** is modeled by the following computation transformation:

$$\mathcal{T}_{\mathrm{2PL}} = \left\{ (C, C') : \begin{array}{l} V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge C \in WF_T \wedge C' \in DRF_L \\ \wedge\ \forall t \in C.\mathit{tids}, \mathit{Tr}_C(t) \text{ is a two-phase enclosure of } C' \end{array} \right\}$$

[5] Bernstein, et al. also discuss a weaker notion of serializability, called *view serializability*, based on strong equivalence, which they call *view equivalence* [14].

[6] Our transaction manager combines some of the functionality of the transaction manager and scheduler of Bernstein, et al. [14], and of the resource managers and the transaction manager of Gray and Reuter [51].

This transformation takes a well-formed computation with transactions and yields a data-race-free computation with locks in which the transactions of the original computation are two-phase enclosures of the transformed computation.

To show that using this transformation, sequential consistency with locking implements sequentially consistent transactions, we use the reserialization technique defined in the previous section: We identify a *serialization point* for each transaction, and show that for any schedule that respects locks, if the operations are reordered according to their serialization points, the result is an equivalent schedule that respects transactions.

For two-phase locking implementations, the serialization point for each transaction is an operation that holds every lock held by the transaction in the transformed computation; that is, an operation in the same l-section as any operation in the enclosure that holds l. First, we show that an operation in an enclosure of a well-formed computation with locks has this property exactly when it follows every operation in the enclosure that acquires l and precedes every operation in the enclosure that release l.

Lemma 8.12 For $C \in WF_L$, if S is an enclosure of C and $x \in S$ then, for all $l \in L$,

$$(\forall y \in S \cap \mathit{Acqs}(l), y \preceq x) \wedge (\forall y \in S \cap \mathit{Rels}(l), x \preceq y) \iff \forall y \in S \cap \mathit{Holds}(l), (x \in \mathit{Holds}(l) \wedge S_l(x) = S_l(y))$$

Proof: Suppose $(\forall y \in S \cap \mathit{Acqs}(l), y \preceq x) \wedge (\forall y \in S \cap \mathit{Rels}(l), x \preceq y)$ and $y \in S \cap \mathit{Holds}(l)$. If $\mathit{gd}(S_l(y)) \in S$ then $\mathit{gd}(S_l(y)) \preceq x$ since $\mathit{gd}(S_l(y)) \in \mathit{Acqs}(l)$. Otherwise, since $\mathit{gd}(S_l(y)) \preceq y \in S$ and S is an enclosure of C', $\mathit{gd}(S_l(y)) \preceq \mathit{gd}(S) \preceq x$. In either case, $\mathit{gd}(S_l(y)) \preceq x$. Similarly, $x \preceq \mathit{rgd}(S_l(y))$. Thus, $x \in S_l(y)$. Since $C \in WF_L$, we have $x \in \mathit{Holds}(l)$ and $S_l(x) = S_l(y)$, as required.

Suppose $\forall y \in S \cap \mathit{Holds}(l), (x \in \mathit{Holds}(l) \wedge S_l(x) = S_l(y))$. If $y \in S \cap \mathit{Acqs}(l) \subseteq S \cap \mathit{Holds}(l)$, then $x \in \mathit{Holds}(l)$ and $S_l(x) = S_l(y)$. Since $y \in \mathit{Acqs}(l)$, $y = \mathit{gd}(S_l(y)) = \mathit{gd}(S_l(x))$, so $y \preceq x$. Similarly, if $y \in S \cap \mathit{Rels}(l)$, then $x \preceq \mathit{rgd}(S_l(x)) = \mathit{rgd}(S_l(y)) = y$. ∎

In any two-phase enclosure of a well-formed computation with locks, there is some operation that follows every lock acquisition and precedes every lock release.

Lemma 8.13 An enclosure S of a well-formed computation C with locks L is two-phase if and only if there exists $x \in S$ such that for all $l \in L$, $y \preceq x$ for all $y \in S \cap \mathit{Acqs}(l)$ and $x \preceq y$ for all $y \in S \cap \mathit{Rels}(l)$.

Proof: Suppose S is two-phase. If $S \cap \bigcup_{l \in L} \mathit{Acqs}(l) = \emptyset$ then for any $l \in L$, we have $S \cap \mathit{Acqs}(l) = \emptyset$, and $\mathit{gd}(S) \preceq y$ for all $y \in S \cap \mathit{Rels}(l)$. Otherwise, choose $x \in S \cap \bigcup_{l \in L} \mathit{Acqs}(l)$ such that $x \not\prec y$ for all $y \in S \cap \bigcup_{l \in L} \mathit{Acqs}(l)$. For any lock l, we have $x \preceq y$ for all $y \in S \cap \mathit{Rels}(l)$ because S is two-phase. For $y \in S \cap \mathit{Acqs}(l)$, we have $x \preceq \mathit{rgd}(S_l(y)) \subseteq \mathit{Rels}(l)$, because $x \preceq \mathit{rgd}(S) \prec \mathit{rgd}(S_l(y))$ if $\mathit{rgd}(S_l(y)) \notin S$, since S is an enclosure. Because C is well-formed and $x \not\prec y$, we have $x \in S_l(y)$, so $y = \mathit{gd}(S_l(y)) \preceq x$.

The other direction is trivial: For all $y \in S \cap \bigcup_{l \in L} \mathit{Acqs}(l)$ and $y' \in S \cap \bigcup_{l \in L} \mathit{Rels}(l)$, we have $y \preceq x \preceq y'$. ∎

Together, the previous two lemmas give an alternative characterization of the two-phase locking transformation, in which every transaction has an operation that holds every lock held by any operation in the transaction.

Lemma 8.14

$$\mathcal{T}_{2PL} = \left\{ (C, C') : \begin{array}{c} V_C = V_{C'} \wedge E_C \subseteq E_{C'} \wedge C \in WF_T \wedge C' \in DRF_L \\ \wedge\ \forall t \in C.tids,\ Tr_C(t) \text{ is an enclosure of } C' \\ \wedge\ \exists x \in Tr_C(t), \forall l \in L, \forall y \in Tr_C(t) \cap Holds_{C'}(l), \\ (x \in Holds_{C'}(l) \wedge S_l(x) = S_l(y)) \end{array} \right\}$$

Proof: Except for the last condition, this definition is identical to the definition of $\mathcal{T}_{2PL}$. The last condition follows from Lemmas 8.12 and 8.13. ■

We use this characterization and the reserialization theorem from the previous section to prove that, using the two-phase locking transformation, every schedule of the transformed computation that respects locking is equivalent to some schedule of the original computation that respects transactions.

Lemma 8.15 If $\alpha \in RespLock(C')$ for some $C' \in \mathcal{T}_{2PL}[C]$ then there exists $\beta \in RespTr(C)$ such that $\beta \equiv_{str} \alpha$.

Proof: By Lemma 8.14, for every $t \in C.tids$, there exists $sp_t \in Tr(t)$ such that for all $y \in Tr(t)$ and $l \in L$, if y holds l then sp_t holds l and $S_l(sp_t) = S_l(y)$. Define sp: $V_C \to V_C$ so that $sp(x) = sp_t$ if $x \in Tr(t)$ and $sp(x) = x$ otherwise, that is, if $x \in Isol$. Note that $sp^{-1}(sp(x)) = Tr_C(x)$ for all $x \in V_C$.

Except for the annotations, $\alpha \in Sch(C)$ because $E_C \subseteq E_{C'}$. If x and y conflict and $x <_\alpha y$ then because $C' \in DRF_L$, either x and y are arbitrated by some lock $l \in L$, or $x \prec_{C'} y$. In the first case, $S_l(x) \neq S_l(y)$, so by Corollary 7.17, $rgd(S_l(x)) <_\alpha gd(S_l(y))$. Since $S_l(sp(x)) = S_l(x)$ and $S_l(sp(y)) = S_l(y)$, we have $sp(x) \leq_\alpha rgd(S_l(x)) <_\alpha gd(S_l(y)) \leq_\alpha sp(y)$. In the second case, either $sp(x) = sp(y)$ or $Tr(x)$ and $Tr(y)$ are disjoint enclosures of C', so by Lemma 7.4, $sp(x) \preceq_{C'} rgd_{C'}(Tr(x)) \prec_{C'} gd_{C'}(Tr(y)) \preceq_{C'} sp(y)$. In either case, $sp(x) \leq_\alpha sp(y)$, so by Theorem 8.11, there exists $\beta \in RespTr(C)$ such that $\beta \equiv_{str} \alpha$. ■

We now prove the main theorem of this section: Two-phase locking—that is, sequential consistency with locking using the two-phase locking transformation—implements sequentially consistent transactions.

Theorem 8.16 $\mathcal{T}_{2PL}(SCL)$ implements SCT.

Proof: If $(C, \rho) \in \mathcal{T}_{2PL}(SCL)$ then there exist $C' \in \mathcal{T}_{2PL}[C]$ and $\alpha \in RespLock(C')$ such that α explains ρ. By Lemma 8.15, there exists $\beta \in RespTr(C)$ such that $\beta \equiv_{str} \alpha$. Thus, β also explains ρ, and $(C, \rho) \in SCT$. ■

Our formulation of Theorem 8.16 is a bit more general than the usual statement of this theorem [35, 14, 51] because we do not assume that transactions are serial nor that there is a fixed association between locks and data items. Also, we allow the transaction manager to take advantage of the way the transactions are requested, and to use additional precedence dependencies rather than locks to enforce data-race-freedom. For example, the manager does not need to use locks to arbitrate conflicting operations of transactions that are not concurrent, or it may explicitly serialize two concurrent transactions instead of using locks to serialize them. We also allow locks to be held across transactions.

The decomposition of a two-phase locking system into a transaction manager and a sequentially consistent system with locking is merely conceptual. It may not correspond with any physical decomposition of the system.

Real implementations of two-phase locking typically use shared/exclusive locks, as discussed in Section 7.9: Locks are held in shared mode for data that will only be read, and in exclusive mode for data that may be written. Although our definition implicitly assumes mutex locks, all the lemmas and theorems in this section hold, with essentially the same proofs, for systems using shared/exclusive locks, provided the notions of respecting locking and data-race-freedom under locking are adapted as described in Section 7.9.

Two-phase locking with weak sequential locking. Although Theorem 8.16 assumes that the underlying system is sequentially consistent with locking, we know from Chapter 7 that a data-race-free computation executes on a system that guarantees only weak sequential locking as though the system were sequentially consistent with locking. Because the transaction manager ensures that the computation for the underlying system is data-race-free, two-phase locking implements sequentially consistent transactions even when the underlying system guarantees only weak sequential locking, which requires significantly less synchronization than sequential consistency with locking.

Corollary 8.17 $\mathcal{T}_{2PL}(WSL)$ implements *SCT*.

Proof: Immediate from previous theorem and Theorem 7.25 because $range(\mathcal{T}_{2PL}) \subseteq DRF_L$. ■

Strict two-phase locking. For various reasons that are outside the scope of this thesis, almost all implementations of two-phase locking guarantee *strict two-phase locking*, in which every lock acquired during a transaction is held until the end of the transaction, at which point all locks are released.

A transaction manager cannot release any lock until it knows that no more locks need to be acquired to enforce data-race-freedom. A strict two-phase locking manager satisfies this requirement simply by not releasing any locks under the end of the transaction. We also restrict the manager from adding precedence dependencies. Formally, a ***strict two-phase locking transaction manager*** is modeled by the following transformation:

$$\mathcal{T}_{S2PL} = \left\{ (C, C') : \begin{array}{l} V_C = V_{C'} \wedge E_C = E_{C'} \wedge C \in WF_T \wedge C' \in DRF_L \\ \wedge\ \forall l \in L, Rels_{C'}(l) = \{et(t) : t \in C.tids\} \cap Holds_{C'}(l) \end{array} \right\}$$

A strict two-phase locking manager is a two-phase locking manager.

Lemma 8.18 $\mathcal{T}_{S2PL}$ is more restrictive than $\mathcal{T}_{2PL}$.

Proof: If $(C, C') \in \mathcal{T}_{S2PL}$ then $V_C = V_{C'}$, $E_C = E_{C'}$, $C \in WF_L$, and $C' \in DRF_L$. For $t \in C.tids$, $Tr_C(t) \cap \bigcup_{l \in L} Rels_{C'}(l) = \{et(t)\}$ and $x \preceq rgd(Tr_C(t)) = et(t)$ for all $x \in Tr_C(t) \cap \bigcup_{l \in L} Acqs_{C'}(l)$. Thus, $Tr_C(t)$ is two-phase in C'. ■

Thus, using strict two-phase locking, weak sequential locking implements sequentially consistent transactions.

Theorem 8.19 $\mathcal{T}_{S2PL}(WSL)$ implements *SCT*.

Proof: Immediate from previous lemma and Corollary 8.17. ■

Degrees of isolation. Two-phase locking, especially strict two-phase locking, may require locks to be held for a long time by each transaction, inhibiting a lot of potential concurrency. To allow greater concurrency, the two-phase locking requirement is often relaxed, holding some locks more briefly at the expense of allowing some partial effects of a transaction to be observed by other transactions. These relaxations are said to provide lower *degrees of consistency* or *degrees of isolation* [50, 51].

The degrees of isolation are most easily defined in terms of how and when locks may be acquired and released, assuming that a shared/exclusive lock is associated with each data item. Third degree isolation requires two-phase locking: Any operation that accesses a data item must hold its associated lock—in exclusive mode if the data is written—and every transaction must be two-phase. Second degree isolation relaxes the two-phase requirement for shared access; that is, additional locks may be acquired after a lock held in shared mode has been released. First degree isolation dispenses with using locks to protect reads altogether; locks are only acquired when an item is to be written. Transactions that run with only first degree isolation may introduce data races into the computation. Zeroth degree isolation requires only that operations that write data hold the associated lock in exclusive mode. Different transactions may run with different degrees of isolation.

Unfortunately, it is not easy to characterize the guarantees of systems with transactions running at lower than third degree of isolation. In particular, in a computation that has transactions with a lower degree of isolation, even a transaction with the third degree isolation may observe inconsistencies.[7] This difficulty is due in part to the definition of degrees of isolation as relaxations of the two-phase locking protocol, rather than by the properties that they guarantee. In Section 8.6, we present relaxed transactional memory models that directly relax the serializability requirement while retaining the some of the spirit of transactions.

8.5 Program Reduction

Even when a concurrent program is not data-race-free, we can reduce the complexity of reasoning about its behavior if we can prove that a collection of its operations can be combined into a single atomic operation. This technique is called *program reduction* [73]. The reduced program is easier to reason about because it has fewer operations, and thus fewer possible schedules. In the literature, program reduction is described and proven by modeling a concurrent system as a state machine [69, 70]. In this section, we adapt this technique to the computation-centric framework, focusing on the specific case of systems with locks, and compare our results to those in the literature. Our exposition most closely follows that of Lampson [70]. This section is independent of the other sections of this chapter.

In a state machine formalism, the operations of a program are modeled as sets of transitions of the state machine. An operation is *enabled* in a state when it has a transition from that state to another (possibly the same) state. Only a subset of the operations are

[7] Inconsistencies here refer to violations of *integrity constraints* in the state of the memory, not the need to use different schedules to explain the values returned for different operations. We discuss integrity further in Section 8.7.

enabled in any state. When operations are combined, their transitions are replaced with the transitions for the new atomic operation.

The basic reduction result may be paraphrased as follows: The "sequential composition" of two atomic operations—that is, one operation immediately followed by another in the program of a thread—may be considered atomic if the first operation "right commutes" with every operation of every other thread that may be enabled after the first operation.

In the computation-centric framework, the sequential composition of two operations corresponds to an enclosure comprised of the two operations. Rather than introduce the formal machinery to model the "compound operation" that would result from combining the operations, we prove that the sequential composition appears atomic as we did for transactions: by showing that any schedule is equivalent to some schedule in which the operations appear contiguously.

Unlike operations of state machines, every operation in our framework is total and deterministic. We introduce nondeterminism only by scheduling the operations, and limit how operations may be applied by requiring schedules to respect the precedence dependencies and the constraints specified by the annotations. For example, acquiring a lock, which can cause a thread to block, is considered an operation of the state machine; we specify lock acquisition using the annotations and restrict schedules to those that respect locking.

The conditions for when one operation is enabled after another, and when one right commutes with another depend on the condition for respecting the constraints implied by the annotations. An operation x is *enabled after* another operation y if x appears immediately after y in some schedule that respects the constraints specified by the annotations. An operation x *right commutes* with y if, whenever $\langle x, y\rangle$ may appear in a schedule, it may be replaced by $\langle y, x\rangle$; that is, if $\alpha{\cdot}\langle x, y\rangle{\cdot}\alpha'$ is a schedule that respects the constraints specified by the annotations, then $\alpha{\cdot}\langle y, x\rangle{\cdot}\alpha'$ is too, and $\alpha{\cdot}\langle x, y\rangle{\cdot}\alpha' \equiv_{\mathrm{str}} \alpha{\cdot}\langle y, x\rangle{\cdot}\alpha'$.

Considering only the precedence dependencies, an operation is enabled after another if they are concurrent, and an operation right commutes with another if they are independent.[8] Locking imposes the following additional restrictions: If x holds and does not release a lock l, then no operation that holds l is enabled after x unless it is in the same l-section. Also, if x releases a lock and y acquires it, then x does not right commute with y.

Formally, an operation z is ***enabled (immediately) after*** x in a computation with locks L if they are concurrent and for each $l \in L$, either they are not arbitrated by l or $x \in \mathit{Rels}(l)$. An operation x ***right commutes*** with z if they are independent and there is no lock l such that $x \in \mathit{Rels}(l)$ and $z \in \mathit{Acqs}(l)$.

If x and y are sequentially composed, then any operation scheduled between x and y is enabled after x.

Lemma 8.20 Suppose C is a well-formed computation with locks L, and $S = \{x, y\}$ is an enclosure of C with $\mathit{gd}(S) = x$ and $\mathit{rgd}(S) = y$. If $\alpha \in \mathit{RespLock}(C)$ then $x <_\alpha z <_\alpha y$ implies that z is enabled after x for all $z \in V_C$.

[8] By our informal description, if $(x, y) \in E_C$ then y may be enabled after x. In that case, x does not right commute with y, by the informal description, even if they are independent. These two changes cancel each other. Defining y not to be enabled after x simplifies the proofs. It is a simple but tedious exercise to verify that Lemma 8.20 and Theorem 8.21 would hold had we followed the informal definitions more closely.

Proof: By Lemma 7.5, z is concurrent with $x = gd(S)$. For $l \in L$, if either x or z does not hold l then they are not arbitrated by l. If they both hold l and $x \notin Rels(l)$ then $y \in S_l(x)$. By Lemma 7.16, $gd(S_l(x)) \leq_\alpha x <_\alpha z <_\alpha y \leq_\alpha rgd(x)$, so $z \in S_l(x)$. Thus, x and z are not arbitrated by l for all $l \in L$, so z is enabled after x. ■

For computations with locks, the basic reduction theorem is:

Theorem 8.21 Suppose C is a well-formed computation with locks L, $S = \{x, y\}$ is an enclosure of C with $gd(S) = x$ and $rgd(S) = y$, and x right commutes with every operation that is enabled after it. If $\alpha = \beta \cdot x \cdot \beta' \cdot y \cdot \beta'' \in RespLock(C)$ then $\alpha' = \beta \cdot \beta' \cdot \langle x, y \rangle \cdot \beta'' \in RespLock(C)$ and $\alpha' \equiv_{str} \alpha$.

Proof: By Lemma 8.20, every operation in β' is enabled after x, and thus, is concurrent with x. Also, x right commutes with every operation enabled after it, so x is independent of every operation in β' and if $x \in Rels(l)$ for some $l \in L$, then no operation in β' acquires l.

Because x is independent of every operation in β', $x \cdot \beta' \equiv_{str} \beta' \cdot x$ by Lemma 2.15. So $\alpha' \equiv_{str} \alpha$.

If $(z, z') \in E_C$, then $z <_\alpha z'$, so $z <_{\alpha'} z'$ unless $z = x$ and $z' \in elems(\beta')$. This latter case is not possible because x is concurrent with every operation in β'. So $\alpha' \in Sch(C)$.

For any $l \in L$, we show below that either $x \notin Acqs(l) \cup Rels(l)$ or $z \notin Acqs(l) \cup Rels(l)$ for all $z \in elems(\beta')$. In either case, $\alpha'^{|l} = \alpha^{|l}$ and thus, α' respects l. So $\alpha' \in RespLock(C)$.

If $x \in Rels(l)$, then, as reasoned above, no operation in β' acquires l. Since x releases l and no operation in β' acquires it, no operation in β' can acquire l.

If $x \in Acqs(l) - Rels(l)$ then suppose, for contradiction, that some operation in β' releases l. Let z_0 be the first such operation in β'. Because C is well-formed and $\alpha \in RespLock(C)$, $rgd(S_l(x)) = z_0$ by Lemma 7.13. Since x acquires l and does not release it, and no operation in β' releases l, neither can any operation in β' acquire l. ■

By applying this theorem repeatedly, we can show that the sequential composition of several operations appears atomic, as long as each operation right commutes with every operation enabled immediately after it.

Theorem 8.21 captures the essence of Lampson's result [70], although his result does not restrict the mechanism for enabling or disabling operations to locking and allows nondeterministic operations. We use the condition for respecting locking to explicitly restrict the possible schedules; other mechanisms imply different restrictions on the schedules, and require their own condition to specify them.

There are several ways in which this result can be extended. For example, the condition that $\{x, y\}$ is an enclosure is stronger than necessary. It is sufficient to require that y be a rear guard. Thus, by repeated application of that theorem, we can show any region with a rear guard can be considered atomic provided each operation in the region, except for the rear guard, right commutes with every operation enabled after it. Also, we can get analogous results for an operation that "left commutes" with operations that are "enabled before" it. Lamport and Schneider state and prove a rather general theorem of this kind [69], about when an invariant is preserved by reduction.

Lampson assumes sequential consistency in his analysis, as do Lamport and Schneider in theirs. This assumption is reflected in our theorems by the use of schedules: Having a single schedule is equivalent to sequentially consistent semantics. As we have shown throughout this thesis, results about the equivalence of schedules can be used to reason about memories with weak consistency. However, we do not develop these results any further in this thesis.

8.6 Relaxed Transactional Memory Models

In this section, we define three relaxed memory models—*generic transactional memory*, *internally consistent transactional memory*, and *internally synchronized transactional memory*—that adhere to the spirit of transactions more closely than the degres-of-isolation models. To our knowledge, these models have not been defined before, and no systems have been designed to implement them.[9] However, they retain several useful properties of sequentially consistent transactions, as we demonstrate in the next two sections.

We begin, as we did for systems with locks, by defining a generic transactional memory that guarantees only that schedules respect transactions. Formally, the model for ***generic transactional memory*** is

$$GT = \{(C, \rho) : \forall x \in V_C, \exists \alpha \in \mathit{RespTr}(C), \rho(x) = \mathit{retval}(x, \alpha)\}.$$

The generic transactional memory is the weakest possible memory that respects transactions. Although every return value is explained by some schedule that respects transactions, a generic transactional memory allows the schedules used to explain the return values to order both the transactions and the operations within each transaction in completely different ways. Thus, a generic transactional memory does not guarantee isolation: a transaction may observe behaviors that it could never observe when executed in isolation. However, in Section 8.8, we give conditions under which generic transactional memory implements sequentially consistent transactional memory.

The two other models do guarantee isolation but give up serializability. Specifically, each transaction must have a single consistent view of the memory, with no partial effects of other transactions; we say that the transactions are *internally consistent*. The first model allows a transaction to be observed by other transactions in an order different from the one used to generate its return values; the second model forbids this by synchronizing operations within the same transaction. Unlike the degrees-of-isolation models, which assume read/write memory with locks, these models are defined for arbitrary data types.

Formally, the model for ***internally consistent transactional memory*** is

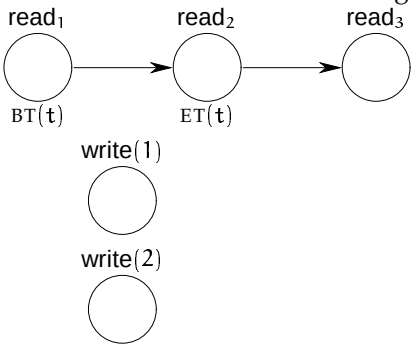
$$\mathit{ICT} = \left\{(C, \rho) : \forall x \in V_C, \exists \alpha \in \mathit{RespTr}(C), \alpha \text{ explains } \rho|_{\mathit{Tr}(x)}\right\}.$$

Unlike the sequential locking models in the previous chapter, *ICT* is *not* merely a generic transactional memory with extra synchronization. Synchronization guarantees that some operations are ordered consistently in all schedules used to explain the value returned for any operation, whether in a transaction or not. Internally consistent transactional memory makes no guarantees about the schedules of operations outside transactions other than that they respect transactions. Instead it requires the return values of operations in the same transaction to be explained by a single schedule.

Example 8.9 Consider a memory model that synchronizes at the beginning and end of every transaction. That is, $\mathit{SBET} = \mathcal{T}^{S}_{\Psi_{\mathrm{SBET}}}(GT)$, where $\Psi_{\mathrm{SBET}}(x, y, a_x, a_y) \equiv a_x \neq \mathrm{NIL} \vee a_y \neq \mathrm{NIL}$. This memory is similar to the strong synchronization memory from Example 5.2, except that it also respects

[9] Of course, any system that implements sequentially consistent transactional memory also implements these relaxed models.

transactions. Consider the following computation and observer functions:



x	$read_1$	$read_2$	$read_3$	write(1)	write(2)
$\rho(x)$	1	2	2	ACK	ACK
$\rho'(x)$	1	1	0	ACK	ACK

SBET admits ρ, but not ρ', for this computation; *ICT* admits ρ' but not ρ. So the two models are incomparable. ■

To get internally synchronized transactions, we strengthen *ICT* by synchronizing the operations of each transaction. In adding edges to the computation, we need to be careful not to incorporate any isolated operation into a nontrivial transaction, so we cannot directly use a synchronizing transformation of the kind defined in Section 5.5. Formally, if C is a well-formed computation with transactions, then $\Psi_{\mathrm{Tr}}(x, y, C) \equiv Tr_C(x) = Tr_C(y)$. The model for ***internally synchronized transactional memory*** is

$$IST = \mathcal{T}_{IST}(ICT), \quad \text{where } \mathcal{T}_{IST} = \left\{(C, C') \in \mathcal{T}^{s}_{\Psi_{\mathrm{Tr}}} : Isol_{C'} = Isol_C\right\}.$$

The computation resulting from transforming a well-formed computation using $\mathcal{T}_{IST}$ has serial transactions.

Lemma 8.22 $\mathcal{T}_{IST}[C] \subseteq SerTr$ for all $C \in WF_T$.

Proof: Immediate from the definitions of $\mathcal{T}_{IST}$ and *SerTr*. ■

Sequentially consistent transactions implement internally synchronized transactions, which implement internally consistent transactions.

Lemma 8.23 $SCT \subseteq IST \subseteq ICT$.

Proof: Immediate from the definitions. ■

For computations with serial transactions, internally consistent transactions and internally synchronized transactions are equivalent.

Lemma 8.24 *IST* and *ICT* are equivalent under *SerTr*.

Proof: If $C \in SerTr$ then $C \in \mathcal{T}_{IST}[C]$, so *ICT* implements *IST* under *SerTr*. ■

Because there is no synchronization across transactions, internally consistent or internally synchronized transactions may be significantly cheaper to implement than sequentially consistent transactions. For example, on a replicated system, each replica can service transactions independently. The effect of a transaction is then lazily propagated among the replicas. Each replica processes one transaction at a time. For internally synchronized transactions, the replicas must also propagate the order in which the operations of each transaction were applied. Because the transactions may not be processed in the same order at different replicas, the transactions may not be serializable.

Another possible implementation borrows from *certification* or *optimistic concurrency* techniques for implementing sequentially consistent transactions [14, 70]. The processor executing a transaction caches the parts of the shared memory it accesses, and writes them back atomically at the end of the transaction. If the parts of the memory that were read for the transaction have since been written by another transaction, the processor must redo the transaction using the new state of the memory. Because the new results may differ from those returned to the client, the original transaction must be aborted to implement sequentially consistent transactions. Aborting transactions is not necessary, however, for internally consistent or internally synchronized transactions.

8.7 Integrity

Historically, one of the main motivations for transactions was to maintain the *integrity* of the data in a database.[10] Integrity is modeled by constraints on the state of the database, which are specified by the system. The clients violate integrity if they specify transactions that, executing in isolation, would falsify an integrity constraint; the system violates it if it does not properly isolate concurrent transactions. In this section, we show how to model integrity constraints and prove that an internally consistent transactional memory maintains integrity provided every transaction maintains integrity.

A database typically stores information about the real world; transactions are used to update that information to reflect changes in the world. Although there is no way to determine, looking only at the database, whether it is consistent with the world—this would be true integrity—we want to ensure that it is at least consistent with some possible state of the world. We formalize this requirement using *integrity constraints* that characterize states of the database that correspond to possible states of the world. Programmers often assume integrity for the correctness of their programs. A transaction maintains the integrity of the memory if, whenever the transaction is applied (in isolation) to a state that satisfies the integrity constraints, the final state also satisfies the integrity constraints.

Formally, for a memory with data type $\mathcal{D} = (\Sigma, \hat{\sigma}, O, R, \tau)$, an integrity constraint is a predicate $I\colon \Sigma \to \mathit{Bool}$ that is true for $\hat{\sigma}$. For $C \in WF_T$, a transaction T of C ***maintains*** I in C if $I(\sigma) \implies I(\tau_\Sigma^*(\sigma, \alpha))$ for all $\sigma \in \Sigma$ and $\alpha \in \mathit{Sch}(C|_T)$. A computation ***maintains*** I if each of its transactions maintains I. We characterize the computations that maintain I by the following client restriction:

$$\mathit{Maintains}(I) = \{C \in WF_T : C \text{ maintains } I\}$$

We would like a transactional memory to maintain integrity whenever the computation maintains integrity. That is, if the computation does not violate integrity, the system will not either. Unfortunately, in the computation-centric framework, we cannot easily determine whether the state of the memory satisfies an integrity constraint; the state of the memory is only well-defined in the serial setting. We cannot even easily characterize the set of possible states of the memory "after" an operation. We can, however, tell whether

[10] This property is also called consistency—the 'C' of the ACID properties of transactions [51].

the values returned for the operations of a transaction are consistent with the memory being in some state satisfying the integrity constraint at the beginning of the transaction. In that case, the return values for that transaction *satisfy the integrity constraint*. A memory model *maintains an integrity constraint* if for any computation that maintains the constraint, it admits only observer functions that satisfy the constraint for all transactions.

Formally, a return value function $\rho\colon V_C \to R$ ***satisfies*** I ***for*** T (in C) if there exists $\sigma \in \Sigma$ and $\alpha \in Sch(C|_T)$ such that $I(\sigma)$ and $\rho(x) = retval_\sigma(x, \alpha)$ for all $x \in T$. A return value function ***satisfies*** I for $C \in WF_T$ if it satisfies I for all transactions of C. A memory model M ***maintains*** I if for all $C \in Maintains(I)$ and $\rho \in M[C]$, ρ satisfies I for C.

If a computation maintains an integrity constraint, then the return values of any schedule that respects transactions satisfy the constraint.

Lemma 8.25 For any integrity constraint I, if $C \in Maintains(I)$, $\alpha \in RespTr(C)$ and $\rho\colon V_C \to R$ such that $\rho(x) = retval(x, \alpha)$ for all $x \in V_C$, then ρ satisfies I for C.

Proof: Since α respects transactions, $\alpha = \beta_1 \cdot \beta_2 \cdot \cdots \cdot \beta_n$, where $\beta_i \in Sch(C|_{T_i})$ for the transactions $T_1, T_2, \ldots, T_n$ of C. Let $\sigma_0 = \hat{\sigma}$ and recursively define $\sigma_i = \tau^*_\Sigma(\sigma_{i-1}, \beta_i)$ for $i = 1, \ldots, n$. Since $I(\hat{\sigma})$ and C maintains I, we have $I(\sigma_i)$ for all i by induction. By Lemma 2.7, $\rho(x) = retval(x, \alpha) = retval_{\sigma_i}(x, \beta_i)$ for each $x \in T_i$, so ρ satisfies I for all T_i. ■

Internally consistent transactional memory maintains any integrity constraint.

Theorem 8.26 *ICT* maintains any integrity constraint.

Proof: Let I be an integrity constraint, C be a computation that maintains I and ρ be admitted by *ICT* for C. For any transaction T of C, there exists $\alpha \in RespTr(C)$ that explains $\rho|_T$. By Lemma 8.25, $retval(\cdot, \alpha)$ satisfies I for C, so it satisfies I for T. Since $\rho(x) = retval(x, \alpha)$ for all $x \in T$, ρ satisfies I for T. Because this proof holds for any transaction, ρ satisfies I for C. ■

It immediately follows from this theorem that internally synchronized and sequentially consistent transactional memory also maintain any integrity constraint.

Corollary 8.27 *IST* and *SCT* maintain any integrity constraint.

Transactional systems are often constructed so that there is fixed set of programs that run as transactions. These programs are developed and checked carefully to ensure that they maintain integrity. Application programs invoke these transaction programs to access the shared database, and so are guaranteed to maintain integrity. Typically, each transaction is serial and essentially deterministic; that is, the observable effects of the transaction, from the application programmer's point of view, depend only on the apparent state of the database when the transaction begins.[11] In this case, we can abstract away the underlying system into a memory whose data type has states that correspond to the states of the underlying data type that satisfy the integrity constraints, and whose operations are the transaction programs. An internally consistent transactional memory is the equivalent

[11] This state is not necessarily the actual state of the database when the operation that begins the transaction is executed, which may not even satisfy the integrity constraints. Rather, it is the state that transaction appears to begin in. In a two-phase locking implementation, this state would correspond to the state of the database at the serialization point of the transaction.

of a generic memory for this abstract system. Thus, transactional memory models form the basis for a hierarchical construction of systems. We do not develop this construction further in this thesis.

8.8 Races within Transactions and Transaction Races

Because transactions are isolated, programmers of transactional memory systems can reason about each transaction separately. In particular, they need to consider only races within a single transaction, not those between operations in different transactions. However, they do need to consider *transaction races*, that is, "races" between whole transactions. In this section, we formally define transaction races, and show that for computations with no transaction races, internally synchronized transactional memory, which resolves races within each transaction, implements sequentially consistent transactions. Furthermore, for computations with no transaction races and no races within transactions, any transactional memory implements sequentially consistent transactional memory.

Informally, each transaction can be viewed as a single complex operation. With this view, we can ask whether any transactions compete; that is, does the order in which the transactions are applied matter? To answer this question, we define *transaction races*. Because the result of a transaction—its effect on the state of the memory and the values returned for its operations—may depend on how the transaction is scheduled, we adapt this definition to consider different schedules for each transaction: concurrent transactions do not compete if the order in which they are applied does not matter regardless of how each transaction is scheduled internally.

Formally, two distinct transactions, T and T', of a well-formed computation $\mathsf{C} \in WF_{\mathrm{T}}$ are ***independent*** if $\alpha{\cdot}\alpha' \equiv_{\mathrm{str}} \alpha'{\cdot}\alpha$ for all $\alpha \in Sch(\mathsf{C}|_{\mathsf{T}})$ and $\alpha' \in Sch(\mathsf{C}|_{\mathsf{T}'})$. They ***conflict*** if they are not independent. They are ***concurrent*** if any operation of one is concurrent with any operation of the other.[12] They ***compete*** if they conflict and are concurrent. Two competing transactions comprise a ***transaction race***. A computation with no transaction races is ***transaction-race-free***. The corresponding client restriction is

$$TRF = \{\mathsf{C} \in WF_{\mathrm{T}} : \mathsf{C} \text{ is transaction-race-free}\}.$$

Transaction-race-freedom guarantees that an internally synchronized transactional memory implements sequentially consistent transactions.

Theorem 8.28 *IST* implements *SCT* under *TRF*.

Proof: This theorem follows immediately from Theorem 8.31 and Lemma 8.29, proved below. ■

Internally consistent transactional memory does not guarantee sequentially consistent transactions for transaction-race-free computations: The transactions may be nondeterministic, and this nondeterminism need not be resolved consistently for different transactions. That is, the order in which a transaction observes its operations to be scheduled

[12] Because transactions are enclosures, this condition implies that every operation of one is concurrent with every operation of the other.

may not be the same order observed by a later transaction. This problem does not arise if the transactions are serial, because serial transactions are deterministic. We can generalize this observation to transactions that appear deterministic to outside operations, that is, for determinate transactions. A transaction is guaranteed to be determinate if it is race-free, that is, if none of its operations compete.

Formally, a convex region S of a computation C is ***(completely) race-free*** in C if $C|_S$ is completely race-free. If C is a well-formed computation with transactions and $Tr(t)$ is race-free for all $t \in C.tids$, we say that C ***has race-free transactions***. The client restriction for computations with race-free transactions is:

$$RFT = \{C \in WF_T : C \text{ has race-free transactions.}\}$$

Although complete race-freedom is a strong restriction for computations, individual transactions are typically race-free. In particular, serial transactions are race-free.

Lemma 8.29 *SerTr* is more restrictive than *RFT*.

Proof: Immediate because a serial computation has no concurrent operations. ■

If a computation is transaction-race-free and its transactions are race-free then all schedules that respect transactions are equivalent.

Lemma 8.30 If $C \in TRF \cap RFT$ and $\alpha, \alpha' \in RespTr(C)$ then $\alpha \equiv_{str} \alpha'$.

Proof Sketch: Suppose $T_1, T_2, \ldots, T_n$ are the transactions of C (including trivial transactions). Let $\beta_i = \alpha|_{T_i}$ and $\beta'_i = \alpha'|_{T_i}$ for $i = 1, \ldots, n$. Because α and α' respect transactions, there exist permutations π and π' of $\{1, \ldots, n\}$ such that $\alpha = \beta_{\pi(1)} \cdot \beta_{\pi(2)} \cdot \cdots \cdot \beta_{\pi(n)}$ and $\alpha' = \beta'_{\pi'(1)} \cdot \beta'_{\pi'(2)} \cdot \cdots \cdot \beta'_{\pi'(n)}$. Thus, we have

$$\begin{aligned}
\alpha &= \beta_{\pi(1)} \cdot \beta_{\pi(2)} \cdot \cdots \cdot \beta_{\pi(n)} \\
&\equiv_{str} \beta'_{\pi(1)} \cdot \beta'_{\pi(2)} \cdot \cdots \cdot \beta'_{\pi(n)} && \text{because } C \in RFT \\
&\equiv_{str} \beta'_{\pi'(1)} \cdot \beta'_{\pi'(2)} \cdot \cdots \cdot \beta'_{\pi'(n)} && \text{because } C \in TRF \\
&= \alpha'
\end{aligned}$$

■

The above lemma implies that any memory that respects transactions implements sequentially consistent transactions.

Theorem 8.31 *GT* implements *SCT* under $TRF \cap RFT$.

Proof: Immediate from Lemma 8.30. ■

8.9 Discussion

One of the main lessons in the development of large distributed systems is that they are hard to manage; they are extraordinarily complex and even seemingly simple programs may have subtle behavior. We badly need methods to understand and organize distributed systems. Transactions are one of the few viable candidates.

A transactional system supports modularity through integrity and isolation. If each transaction maintains the integrity of the data when running alone and the memory system guarantees isolation, then the composed system will also maintain integrity. Thus, programmers of a transactional memory can assume integrity and consider each transaction independently of all others, as long as they ensure it maintains integrity. This modularity is the main reason we believe transactions may be the foundation of a method to organize large distributed systems. The work in this chapter should be seen as a beginning to further study of transactions and how to use them to structure concurrent programs.

Although there has been a lot of work on transactions [51], most of the formal work has focused on serializable transactions [14]. We are interested in relaxed transactional memory models. In particular, what properties should such a model guarantee, what can we implement with these properties, and can a system implement this model efficiently? The property we believe is ultimately important is modularity.

We have shown that, unlike the degrees-of-isolation models, internally consistent transactions guarantee enough isolation to maintain integrity as described above. Clients can see inconsistent views but not within a single transaction. Is this guarantee strong enough for programmers to build large distributed systems?

One way in which transactions may be used to structure systems is to implement powerful abstract data types on top of a memory that has a simple data type. This approach encourages the data-oriented programming that has been successful in sequential programming. The advantage of this method is that there are a limited number of transaction programs written, so each of them can be checked carefully to ensure that it maintains integrity. Several aspects of the computation-centric framework need to be extended to accommodate this approach. In particular, we need to have some mechanism for hiding the values returned for "internal" operations, and we may consider several different return values to be logically equivalent. These changes require a new notion of equivalence for operator sequences.

Nondeterministic data types are another extension to the framework that may be useful for modeling abstract data types implemented by transactions. A transaction may cause the state to change in any one of several nonequivalent ways, and the client is willing to deal with this nondeterminism. Determinacy may be unrealistic—and unnecessary–for concurrent systems. If we have nondeterministic data types, we probably want to define an implementation relation between data types.

One difficulty with modeling transactions in the computation-centric framework is that the operations that comprise a transaction may depend on the values returned for the early operations of the transaction. This problem is not unique to transactions; rather, it is endemic to the computation-centric approach. In Chapter 9, we propose state machine models to address the problem with computations as a whole. To use the same approach for transactions would effectively require clients to submit a state machine as a transaction. This approach was taken by Lynch, et al. [80] in their book on atomic transactions.

We also want to consider how transaction interact with other mechanisms for synchronizing or organizing concurrent programs. We want to do this with every mechanism, but it is particularly important for transactions or any mechanism that may serve as the basis for organizing concurrent systems into modules. Ideally, each module could use its own form of synchronization, and some other form of synchronization could be used over

the transactions. In locking implementions of transactions, the synchronization within a transaction can interfere with synchronization within other transactions—in fact, it must to guarantee transactions—which goes against the idea of transactions providing modularity.

One other aspect of transactions that is attractive is that they were developed originally for fault tolerance. The atomicity and durability of transactions have little meaning without the possibility of failures. Asynchronous concurrency and fault tolerance are the two most difficult aspects distributed computing; it is serendipitous that both are addressed with a single mechanism.

Chapter 9

Dynamic Memory Models

Although computation-centric memory models provide a basis for a simple and powerful way to reason about shared memory systems, they do not capture the dynamic interaction between the memory and its clients. In particular, a computation-centric model specifies what values may be returned when a given computation is requested by the clients, but the computation requested may depend on values returned by the memory. In this chapter, we develop a theory based on state machines that captures this dynamic interaction. This theory builds on the computation-centric theory developed in earlier chapters, using results about computation-centric models to derive analogous results about the dynamic models. Indeed, we give a generic translation from the computation-centric framework to the dynamic framework that preserves the results of the computation-centric framework.

In addition to being more expressive than computation-centric models, using state machines to specify memory consistency guarantees has the advantage that state machines are widely used to model computer systems. Nonetheless, as we discussed in Section 4.6, the state machine approach to specifying memory models has not been as successful as static postmortem approaches because the state machine representation of a system does not make clear what consistency properties the system guarantees. By building on the computation-centric theory, we can define dynamic memory models that capture the properties expressed by computation-centric models and leverage the results from the computation-centric theory in our development of a theory for dynamic memory models.

The key feature of the computation-centric framework that enables us to use it as a basis for the dynamic framework is the clean split between the clients and the memory provided by computations and observer functions. We maintain this split by using *input/output (I/O) automata* [81, 79] to formalize the dynamic models. The computation is specified by a *clients automaton*, the observer function by a *memory automaton*. The interface between these automata is illustrated in Figure 9-1.

To allow the computation to depend on the return values, a clients automaton specifies the computation by requesting one operation at a time, and a memory automaton responds with one return value at a time. The clients may request several operations before receiv-

The automaton models in this chapter are similar to models used in earlier papers [77, 36], but the explicit connection to computation-centric models, which constitutes most of the work presented in this chapter is new.

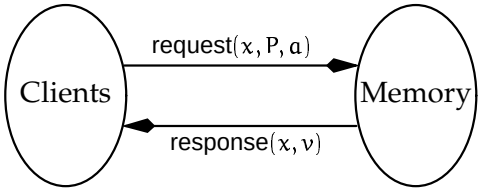


Figure 9-1: Interaction diagram for the clients and the memory. The clients request an operation x with a set P of operations that x depends on and an annotation a. The memory responds with the operation x and its return value v.

ing responses to earlier operations. In keeping with the spirit of the computation-centric framework, we require the clients to specify the precedence dependencies and annotation for each operation. Thus, the requests at any point of an execution define a computation. The memory's responses include the operation being responded to and its return value, defining a partial return value function on the requested operations. By using precedence dependencies and annotations, we preserve the linguistic neutrality of the computation-centric framework.

We develop general mechanisms to translate concepts from the computation-centric framework into the dynamic framework. We show that if one computation-centric model implements another, the translation of the first implements the translation of the second, allowing us to leverage much of the work we did in the computation-centric framework into the dynamic framework. Since the dynamic framework is richer, this translation is not uniquely defined; in fact, we give two such translations, a simple one for complete memory models and a more complicated one that takes into account well-formedness conditions.

Outline: Section 9.1 provides an overview of the I/O automaton model, which we use in Section 9.2, to define the dynamic interface between the clients and the memory. In Section 9.3, we discuss how systems, and particularly programs, are modeled using automata. In Section 9.4, we define dynamic versions of some simple memory models, including sequential consistency, and in Section 9.5, we define the client restrictions of safety and race-freedom, and we prove the analogue of Theorem 5.4, that any memory implements sequential consistency under race-free clients. In Section 9.6, we show how to derive computation-centric models from automaton models for systems, and we compare the dynamic and computation-centric frameworks we have proposed. Sections 9.7 and 9.8 reverse the derivation of the previous section and define two simple general translations from the computation-centric framework to the dynamic framework.

Reading Guide: This chapter is just a beginning of a dynamic theory, leveraging the computation-centric theory we have developed in this thesis. All the results we derive are either analogues of results in the computation-centric theory, or show how we can translate results from the computation-centric framework to the dynamic framework. The chief contribution of this chapter is to show how the computation-centric framework can

serve as a basis and guide to a full theory of memory consistency, addressing the concerns we expressed in Sections 3.8 and 4.6.

9.1 The Input/Output Automaton Model

The *input/output (I/O) automaton* [81] is the formal device we use to model asynchronous distributed systems. This section provides an overview of the I/O automaton model and introduces notation and terminology used to describe and reason about I/O automata. Because we do not deal with liveness in this thesis, we omit the aspects of I/O automata related to liveness. Most of this terminology is standard in the literature. Readers familiar with I/O automata may skip this section.

An I/O automaton is a labeled state-transition system, where each transition, or *step*, from one state to another is labeled with an *action*. The actions are partitioned into *input*, *output* and *internal* actions. The input and output actions represent the visible, or *external*, aspects of an automaton; other aspects of the automata—states and internal actions—are not visible outside the automaton. The behavior of an automaton is modeled by a sequence of external actions, called a *trace*.

A distributed system composed of modular interacting components can be modeled as the *composition* of several I/O automata, one for each component. The composition of the component automata, which models the entire system, is also an I/O automaton. Component interaction is modeled by shared actions; the output actions of one automaton may be input actions of other automata.

In addition to describing systems, I/O automata can specify system requirements. That is, I/O automata can model both system implementations and specifications. One automaton *implements* another if every behavior of the first is a possible behavior of the second. An automaton that describes the system at one level of detail may be a specification for a more detailed view of the system.

9.1.1 Formal Definitions

This subsection contains the formal definitions and results from I/O automaton theory that are required in this chapter. It may be skipped initially and referred to as needed. A full discussion of I/O automata can be found in the literature (e.g., [81, 79]).

An I/O automaton is a *labeled state-transition system*. An *execution* of the automaton is a sequence of labeled *steps*. The labels, called *actions*, indicate the kind of *event* that happened in the system that caused the state to change. Actions may be *input*, *output* or *internal*. The behavior of the system being modeled is described by the *traces* of the automaton, which are the input and output actions that label an execution of the automaton.

Formally, a ***(nonlive) input/output (I/O) automaton*** $\mathcal{A}$ consists of:

- a set $acts(\mathcal{A})$ of ***actions***, partitioned into three (disjoint) sets: $in(\mathcal{A})$, $out(\mathcal{A})$ and $int(\mathcal{A})$;
- a set $states(\mathcal{A})$ of ***states***;
- a nonempty subset $start(\mathcal{A})$ of ***start states***;

- a set $steps(\mathcal{A}) \subseteq states(\mathcal{A}) \times acts(\mathcal{A}) \times states(\mathcal{A})$ of ***steps*** such that for all $s \in states(\mathcal{A})$ and $\pi \in in(\mathcal{A})$, there exists $s' \in states(\mathcal{A})$ with $(s, \pi, s') \in steps(\mathcal{A})$.

We call the actions in $in(\mathcal{A})$, $out(\mathcal{A})$ and $int(\mathcal{A})$ the ***input***, ***output*** and ***internal*** actions respectively. The input and output actions are also called ***external actions***, and the set of external actions is denoted by $ext(\mathcal{A})$. We say that s is the ***pre-state*** and s' is the ***post-state*** of the step (s, π, s'). We write $s \xrightarrow[\mathcal{A}]{\pi} s'$ or just $s \xrightarrow{\pi} s'$ as shorthand for $(s, \pi, s') \in steps(\mathcal{A})$. An action π is ***enabled*** in s if there exists s' such that $s \xrightarrow{\pi} s'$. Note that every input action is enabled in every state.

Executions and traces. An automaton executes by beginning in a start state and taking steps. An ***execution fragment*** $s_0, \pi_1, s_1, \pi_2, s_2, \ldots$ is a finite or infinite sequence of alternating states and actions such that $s_{i-1} \xrightarrow{\pi_i} s_i$ for all i. An ***execution*** is an execution fragment whose first state is a start state; that is, where $s_0 \in start(\mathcal{A})$. We denote the set of executions of $\mathcal{A}$ by $execs(\mathcal{A})$. An ***event*** is an occurence of an action in an execution. A state is ***reachable*** in $\mathcal{A}$ if it appears in any execution of $\mathcal{A}$. An ***invariant*** of $\mathcal{A}$ is a predicate that is true of every reachable state of $\mathcal{A}$.

The behavior of a system is modeled by its *external image*. The states and internal actions of an execution cannot be seen by external observers. A system is characterized by its *traces*, which model the behaviors that might be observed when the system executes. Formally, the ***external image*** of an execution fragment α is its projection $\alpha|_{ext(\mathcal{A})}$ onto its external actions. A ***trace*** of $\mathcal{A}$ is the external image of an execution, and the set of traces is denoted by $traces(\mathcal{A})$. Two executions with the same trace cannot be distinguished, even if they are executions of different automata.

Composition. We often describe a distributed system by specifying the *components* that comprise the system. The entire system is modeled by an automaton that is the *composition* of the automata that model the components. Informally, composition identifies actions with the same name at different component automata. When an action is executed, it is executed by all components with that action. The composite automaton has the actions of all its components. Actions are classified as internal or external according to their classification by the component automata, and the input actions of the composition are those input actions of component automata that are not output actions of any other component. Some restrictions on the automata to be composed are necessary so that the composition makes sense. In particular, internal actions cannot be shared, an action can be the output action of at most one component, and actions cannot be shared by infinitely many components.

Formally, a family $\{\mathcal{A}_i\}_{i \in I}$ of automata is ***compatible*** if $int(\mathcal{A}_i) \cap acts(\mathcal{A}_j) = \emptyset$ and $out(\mathcal{A}_i) \cap out(\mathcal{A}_j) = \emptyset$ for all $i, j \in I$ such that $i \neq j$, and no action is in $acts(\mathcal{A}_i)$ for infinitely many $i \in I$. The ***composition*** $\mathcal{A} = \prod_{i \in I} \mathcal{A}_i$ of a compatible family $\{\mathcal{A}_i\}_{i \in I}$ of automata has the following components:

- $in(\mathcal{A}) = \bigcup_{i \in I} in(\mathcal{A}_i) - \bigcup_{i \in I} out(\mathcal{A}_i)$
  $out(\mathcal{A}) = \bigcup_{i \in I} out(\mathcal{A}_i)$
  $int(\mathcal{A}) = \bigcup_{i \in I} int(\mathcal{A}_i)$
- $states(\mathcal{A}) = \prod_{i \in I} states(\mathcal{A}_i)$

- $start(\mathcal{A}) = \prod_{i \in I} start(\mathcal{A}_i)$
- $steps(\mathcal{A}) = \left\{(s, \pi, s') : \forall i \in I, s_i \xrightarrow{\pi}_{\mathcal{A}_i} s_i' \vee (\pi \notin acts(\mathcal{A}_i) \wedge s_i = s_i')\right\}$

We denote the composition of two compatible automata $\mathcal{A}$ and $\mathcal{B}$ by $\mathcal{A} \times \mathcal{B}$. Composition is associative and commutative up to isomorphism.

Given an execution of a composite automaton, we often want to look at the part of that execution that "belongs" to one of the components. That is, we want the part of the execution, called the *projection* of the execution onto a component, that consists of the actions of that component, and the component of the state that corresponds to the component automaton. We define the projection of a trace similarly.

Formally, suppose $\{\mathcal{A}_i\}_{i \in I}$ is a compatible family of automata. For $\alpha \in execs(\prod_{i \in I} \mathcal{A}_i)$, the ***projection*** $\alpha|_{\mathcal{A}_i}$ onto $\mathcal{A}_i$ is the sequence α' consisting of alternating states and actions of $\mathcal{A}_i$ such that $\alpha'|_{acts(\mathcal{A}_i)} = \alpha|_{acts(\mathcal{A}_i)}$ and the states of α' are the ith component of the states in α preceding the actions in α' (and the final state if α is finite). Similarly, for $\beta \in traces(\prod_{i \in I} \mathcal{A}_i)$, its ***projection*** $\beta|_{\mathcal{A}_i}$ onto $\mathcal{A}_i$ is its projection $\beta|_{acts(\mathcal{A}_i)}$ onto the actions of $\mathcal{A}_i$. We also write $execs(\prod_{i \in I} \mathcal{A}_i)|_{\mathcal{A}_i}$ and $traces(\prod_{i \in I} \mathcal{A}_i)|_{\mathcal{A}_i}$ for the sets of projections onto $\mathcal{A}_i$ of executions and traces of $\prod_{i \in I} \mathcal{A}_i$.

We now prove several results that are useful in reasoning about composite automata.

The projections of executions and traces of composite automata onto their components are executions and traces of the components.

Theorem 9.1 If $\alpha \in execs(\prod_{i \in I} \mathcal{A}_i)$ then $\alpha|_{\mathcal{A}_i} \in execs(\mathcal{A}_i)$.

Proof: Straightforward by induction on the length of the execution. ■

Corollary 9.2 If $\beta \in traces(\prod_{i \in I} \mathcal{A}_i)$ then $\beta|_{\mathcal{A}_i} \in traces(\mathcal{A}_i)$.

The next three results are the converse, in a fashion, of the previous two. The first lemma says that executions of the components that meet certain requirements—that their common actions appear in the same order—can be "pasted" together into an execution of the composite automaton.

Lemma 9.3 Suppose $\beta \in acts(\prod_{i \in I} \mathcal{A}_i)^*$ and $\alpha_i \in execs(\mathcal{A}_i)$ such that $\beta|_{acts(\mathcal{A}_i)} = \alpha_i|_{acts(\mathcal{A}_i)}$ for all $i \in I$. Then there exists $\alpha \in execs(\prod_{i \in I} \mathcal{A}_i)$ such that $\beta = \alpha|_{acts(\prod_{i \in I} \mathcal{A}_i)}$ and $\alpha_i = \alpha|_{\mathcal{A}_i}$ for all $i \in I$.

Proof Sketch: By induction on the length of β. Note that input actions are always enabled, and β_{last} is an internal or output action of at most one of the component automata. ■

The next theorem is exactly the same as the previous lemma, except that it only looks at the external actions.

Theorem 9.4 Suppose $\beta \in ext(\prod_{i \in I} \mathcal{A}_i)^*$ and $\alpha_i \in execs(\mathcal{A}_i)$ such that $\beta|_{ext(\mathcal{A}_i)} = \alpha_i|_{ext(\mathcal{A}_i)}$ for all $i \in I$. Then there exists $\alpha \in execs(\prod_{i \in I} \mathcal{A}_i)$ such that $\beta = \alpha|_{ext(\prod_{i \in I} \mathcal{A}_i)}$ and $\alpha_i = \alpha|_{\mathcal{A}_i}$ for all $i \in I$.

Proof Sketch: Every action in α_i but not β is an internal action of $\mathcal{A}_i$ and not an action of any other component automaton. Create a new sequence $\beta' \in acts(\prod_{i\in I} \mathcal{A}_i)^*$ by inserting the internal actions of each α_i into β so that $\beta'|_{acts(\mathcal{A}_i)} = \alpha_i|_{acts(\mathcal{A}_i)}$ for each $i \in I$. By the previous lemma, there exists $\alpha \in execs(\prod_{i\in I} \mathcal{A}_i)$ such that $\beta' = \alpha|_{acts(\prod_{i\in I} \mathcal{A}_i)}$ and $\alpha_i = \alpha|_{\mathcal{A}_i}$ for all $i \in I$. Thus, $\beta = \beta'|_{ext(\prod_{i\in I} \mathcal{A}_i)} = \alpha|_{ext(\prod_{i\in I} \mathcal{A}_i)}$. ■

A corollary of the previous theorem states that traces of component automata can be pasted together if they have the same common actions; that is, a sequence of external actions of a composite automata that projects to a trace on each component automaton is a trace of the composite automaton.

Corollary 9.5 If $\beta \in ext(\prod_{i\in I} \mathcal{A}_i)^*$ such that $\beta|_{ext(\mathcal{A}_i)} \in traces(\mathcal{A}_i)$ for all $i \in I$ then $\beta \in traces(\prod_{i\in I} \mathcal{A}_i)$.

Proof: Since $\beta|_{ext(\mathcal{A}_i)} \in traces(\mathcal{A}_i)$, there exists $\alpha_i \in execs(\mathcal{A}_i)$ such that $\beta|_{ext(\mathcal{A}_i)} = \alpha_i|_{ext(\mathcal{A}_i)}$ for each $i \in I$. By the previous theorem, there exists $\alpha \in execs(\prod_{i\in I} \mathcal{A}_i)$ such that $\beta = \alpha|_{ext(\prod_{i\in I} \mathcal{A}_i)}$, so $\beta \in traces(\prod_{i\in I} \mathcal{A}_i)$. ■

If all the automata share all their external actions, that is, if the output actions of each automaton are input actions to all the others, then the traces of the composite automaton are exactly those that are traces of all the component automata.

Corollary 9.6 If $\{\mathcal{A}_i\}_{i\in I}$ is a compatible family of automata and $ext(\mathcal{A}_i) = ext(\mathcal{A}_{i'})$ for all $i, i' \in I$, then $traces(\prod_{i\in I} \mathcal{A}_i) = \bigcap_{i\in I} traces(\mathcal{A}_i)$.

Proof: Immediate from Corollaries 9.2 and 9.5. ■

Implementation and simulations. In addition to modeling distributed systems, I/O automata can be used as specifications for such systems: An automaton specifies what behaviors a system is allowed to exhibit when it executes. A system that exhibits only those behaviors (but not necessarily all of them) is said to *implement* the specification.

Formally, we say that an automaton $\mathcal{A}$ ***implements*** another automaton $\mathcal{B}$, and write $\mathcal{A} \subseteq \mathcal{B}$, if $in(\mathcal{A}) = in(\mathcal{B})$, $out(\mathcal{A}) = out(\mathcal{B})$, and $traces(\mathcal{A}) \subseteq traces(\mathcal{B})$. We say that $\mathcal{A}$ and $\mathcal{B}$ are ***equivalent***, and write $\mathcal{A} \equiv \mathcal{B}$, if they implement each other.

The "implements" relation is not symmetric: If one automaton—the implementation—implements another—the specification—then any behavior the implementation exhibits is a behavior that the specification allows. However, the specification may allow behaviors that the implementation never exhibits. Given the implementation, an external observer cannot distinguish it from the specification. But another automaton implementing the specification may exhibit a behavior that the original implementation would not.

Because the behaviors of an implementation may be more restricted than those of the specification, the guarantees provided by the implementation may be stronger than those provided by the specification. Nonetheless, we often prefer to reason about the specification automaton because it is usually simpler than the implementation automaton.

Implementation is a composable property; that is, if $\mathcal{A}$ and $\mathcal{B}$ are composite automata with corresponding components, and if each component of $\mathcal{A}$ implements its corresponding component in $\mathcal{B}$ then $\mathcal{A}$ implements $\mathcal{B}$. If a system is described as a composition, we

need to check only that each component implements its specification to guarantee that the entire system implements its specification.

Theorem 9.7 If $\mathcal{A}_i \subseteq \mathcal{B}_i$ for all $i \in I$ then $\prod_{i\in I} \mathcal{A}_i \subseteq \prod_{i\in I} \mathcal{B}_i$.

Proof: If $\beta \in traces(\prod_{i\in I} \mathcal{A}_i)$ then $\beta|_{\mathcal{B}_i} = \beta|_{\mathcal{A}_i} \in traces(\mathcal{A}_i) \subseteq traces(\mathcal{B}_i)$ [since $ext(\mathcal{A}_i) = ext(\mathcal{B}_i)$] for all $i \in I$. Thus, by Corollary 9.5, $\beta \in traces(\prod_{i\in I} \mathcal{B}_i)$. ■

Similarly, if one composite automaton implements another, then replacing one component with an automaton that implements it preserves the implementation relation.

Theorem 9.8 If $\mathcal{A} \times \mathcal{B} \subseteq \mathcal{A}' \times \mathcal{B}$ and $\mathcal{B}' \subseteq \mathcal{B}$ then $\mathcal{A} \times \mathcal{B}' \subseteq \mathcal{A}' \times \mathcal{B}'$.

Proof: If $\beta \in traces(\mathcal{A} \times \mathcal{B}')$ then $\beta|_{\mathcal{A}} \in traces(\mathcal{A}) \subseteq traces(\mathcal{A}')$ and $\beta|_{\mathcal{B}'} \in traces(\mathcal{B}')$ by Corollary 9.2. Thus, $\beta \in traces(\mathcal{A}' \times \mathcal{B}')$ by Corollary 9.5. ■

One common way to show that one automaton implements another is to use a *simulation*, which establishes a correspondence between the states of the two automata. Formally, if $\mathcal{A}$ and $\mathcal{B}$ are automata with $in(\mathcal{A}) = in(\mathcal{B})$ and $out(\mathcal{A}) = out(\mathcal{B})$ then a ***forward simulation*** from $\mathcal{A}$ to $\mathcal{B}$ is a relation F between $states(\mathcal{A})$ and $states(\mathcal{B})$ such that:

- If $s \in start(\mathcal{A})$ then there exists some $u \in start(\mathcal{B})$ such that $(s, u) \in F$.
- For reachable states s and u of $\mathcal{A}$ and $\mathcal{B}$, if $(s, u) \in F$ and $s \xrightarrow{\pi}_{\mathcal{A}} s'$, then there exists some u' such that $(s', u') \in F$ and there is some execution fragment of $\mathcal{B}$ from u to u' with the same external image as π.

We denote $\{u : (s, u) \in F\}$ by $F[s]$, and we typically write $u \in F[s]$ instead of $(s, u) \in F$.

Theorem 9.9 If there is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$ then $\mathcal{A} \subseteq \mathcal{B}$.

Proof: By induction on the length of an execution of $\mathcal{A}$. ■

There is a particularly simple case, when two automata are identical except that one has a more restrictive step relation. We need to consider only steps whose pre-states are reachable.

Corollary 9.10 Suppose $\mathcal{A}$ and $\mathcal{B}$ are identical I/O automata except for steps and internal actions. If $int(\mathcal{A}) \subseteq int(\mathcal{B})$ and $s \xrightarrow{\pi}_{\mathcal{A}} s'$ implies $s \xrightarrow{\pi}_{\mathcal{B}} s'$ for all reachable states s then $\mathcal{A} \subseteq \mathcal{B}$.

Proof: Immediate from Theorem 9.9 because the identity function is a forward simulation. ■

9.1.2 State Variables and Precondition-Effect Statements

I/O automata are often described in a stylized way using *state variables* to describe the states and *precondition-effect statements* to describe the steps. This style, common in the literature, typically produces a concise description of the automaton.[1] We briefly discuss the main features of this style.

[1] Automaton descriptions tend to be longer than programs written for a concurrent system because such systems have a restricted model of concurrency, which is assumed implicitly by the programs. For example, programs almost always have an implicit program counter; concurrent programs may have several. When these counters are modeled explicitly, the programs become as long as the automaton descriptions.

We describe the state of an automaton by a set of state variables, each with a given domain. The state space of the automaton is the product of the domains of its state variables. We denote the value of a state variable x in state s by $s.x$. Typically, each state variable has a unique initial value, which defines a unique start state for the automaton.

Actions are usually fixed labels with parameters. Each choice of parameters defines a different action, but actions with the same fixed label are similar. The steps are described by precondition-effect statements, typically with a single precondition-effect statement for each group of actions sharing a fixed label. The precondition defines when the action is enabled. The effect is described by a simple program, usually written in pseudo-code with only simple control structures, which defines the relation between the pre-state and post-state of a step. All variables not specifically mentioned are unchanged between the pre-state and the post-state.

Sometimes it is natural to state the enabling condition of an action in terms of the post-state of the step. We use *primed variables* in the precondition to indicate the value of the variable in the post-state. That is, if x is a state variable of an automaton, then x' in the precondition of a step statement refers to the value that x would have after applying the effect clause for that step.

If a state variable has a unique initial value and is changed only by external actions, which update it deterministically, then the value of the variable in any reachable state of the automaton can be determined by looking only at the external image of an execution that leads to that state. We call such variables ***trace (state) variables***; they are determined by the trace. Several compatible automata may have the same trace variable with the same initial value updated in the same way by all the automata. An invariant of the composition of these automata is that the variable has the same value in all components. In this case, we often think of there being only one variable for the composite automaton.

Sometimes there is a condition that we want to check, but is cumbersome to write in the code. In this case, we employ ***derived (state) variables***, which are not really state variables at all. Instead, they are functions of the real state variables. Derived variables may be used in the precondition-effect statements, but they are never assigned a new value. A derived variable may change even when not mentioned specifically in the precondition-effect statement, if a state variable it is derived from changes. A derived variable is also a trace variable if it is a function of trace variables only.

9.2 The Dynamic Interface

In the computation-centric framework, the clients specify a computation and the memory specifies an observer function for the computation. This interface fails to capture the dynamic dependence that the computation may have on the values returned by the memory. In this section, we formally define the interface illustrated in Figure 9-1 on page 170 using I/O automata. This interface captures the dynamic interaction between the clients and the memory by allowing the clients to request one operation at a time and the memory to respond to one operation at a time. We define a *generic clients automaton*, which models the interface from the clients' side, and a *generic memory automaton*, which models the interface from the memory's side. These generic automata express basic well-formedness requirements for all clients and memory automata.

Because we are concerned only with the interaction between the clients and the memory, we model the clients as a single automaton; communication among clients is internal to this automaton. We call this automaton a *clients* automaton to emphasize that it models all the clients together. Real implementations of the clients, of course, are usually distributed. Similarly, we model the memory as a single automaton, although it too may be distributed.

Each client request specifies one operation, together with its precedence dependencies and annotation. The precedence dependencies are specified by a set of previously requested operations on which the new operation depends. Although the clients request operations one at a time, they need not block; a client may request new operations before it receives return values for operations requested earlier.

Each memory response contains the return value for one operation. The memory need not respond to the operations in the order it received them; it may even respond to an operation before it responds to another operation that logically precedes it. Such an inversion reflects the variation in the delay from the time an operation is applied to the time its return value is propagated to the clients.

There are some simple well-formedness conditions on these automata: The clients must ensure that each operation is unique and that the operations it depends on have been requested previously. The memory may return values only once for each requested operation. In addition, the memory cannot make up return values; every return value must be explained by some schedule of the requested operations. Because the condition on the clients automaton prevents cyclic and dangling dependencies, the requests define a computation, which we call the *client constraints graph*. The responses define a partial return value function on the requested operations. We prove these properties in Invariants 9.11 and 9.12 below.

The full formal definitions of the generic clients automaton and the generic memory automaton appear in Figure 9-2. Both automata are parameterized by the data type $\mathcal{D}$ and the annotation set A, and they maintain the same state variables: *reqs* records all the operations that have been requested; *deps* records the precedence dependencies specified in the requests; *ann* records the annotation for each requested operation; and *resps* records the values returned for the operations that have received responses. The ***client constraints graph*** *CCG* is derived from *reqs*, *deps* and *ann*. The client constraints graph is a computation in any reachable state of the generic clients automaton. We call its precedence order the ***client-specified order***, or simply the ***client order***, and denote it by $\prec_c$, which is also a derived state variable of this automaton.

The action $\mathsf{request}(x, P, a)$ is an output action of the clients automaton and an input action of the memory automaton. It is enabled in the clients automaton if x has not yet been requested and all the operations in P have been. It is always enabled in the memory automaton because it is an input action. The effect in both automata is to add x to *reqs*.

The response actions are output actions of the memory and input actions of the clients. The precondition of the response action in the memory automaton ensures that responses are generated only once for each requested operation. The value returned must be explained by some schedule of the client constraints graph. The effect of a response action in both automata is to record the value returned for the operation.

Because all the state variables are updated deterministically by external actions, they

Generic Clients Automaton: $\mathcal{GC}(\mathcal{D}, A)$

Actions

Input:
  response(x, v), where $x \in O$ and $v \in R$
Output:
  request(x, P, a), where $x \in O$, $P \subset O$ and $a \in A$

Generic Memory Automaton: $\mathcal{GM}(\mathcal{D}, A)$

Actions

Input:
  request(x, P, a), where $x \in O$, $P \subset O$ and $a \in A$
Output:
  response(x, v), where $x \in O$ and $v \in R$

State Variables

reqs: a subset of O, initially empty
deps: a binary relation on O, initially empty
ann: $O \to A_\perp$, initially all $\perp$
resps: $O \to R_\perp$, initially all $\perp$
Derived variable: *CCG*: the graph $(reqs, deps)$ annotated by $ann|_{reqs}$
Derived variable: $\prec_c = TC(deps)$, a binary relation on *reqs*

Steps

Output request(x, P, a)
  Pre: $x \notin reqs$
    $P \subseteq reqs$
  Eff: $reqs \leftarrow reqs \cup \{x\}$
    $deps \leftarrow deps \cup \{(y, x) : y \in P\}$
    $ann(x) \leftarrow a$

Input response(x, v)
  Eff: $resps(x) \leftarrow v$

Steps

Input request(x, P, a)
  Eff: $reqs \leftarrow reqs \cup \{x\}$
    $deps \leftarrow deps \cup \{(y, x) : y \in P\}$
    $ann(x) \leftarrow a$

Output response(x, v)
  Pre: $x \in reqs - domain(resps)$
    $v = retval(x, \alpha)$ for some schedule α of *CCG*
  Eff: $resps(x) \leftarrow v$

Figure 9-2: The generic clients and memory automata with data type $\mathcal{D} = (\Sigma, \hat{o}, O, R, \tau)$ and annotation set A. They have the same state variables.

are trace variables; their values in any reachable state can be determined from any trace leading to that state. Furthermore, they are updated in the same way by both automata, so their values in each automaton are the same, as suggested by the joint listing of the variables in Figure 9-2.

A ***clients automaton*** for $\mathcal{D}$ and A is any automaton that implements $\mathcal{GC}(\mathcal{D}, A)$, and a ***memory automaton*** for $\mathcal{D}$ and A is any automaton that implements $\mathcal{GM}(\mathcal{D}, A)$. Because the data type and annotation set are fixed, we usually drop them from the notation. For convenience in later discussion and analysis, we assume that every clients automaton and every memory automaton has the variables listed in Figure 9-2 and updates them in exactly the same way. Real implementations need not maintain these variables, of course; we maintain them abstractly for analysis.

The following invariant implies that the client constraints graph is a computation in any reachable state of a clients automaton.

Invariant 9.11 For $\mathcal{GC}$: $domain(ann) = reqs$ and *deps* is a strict partial order.

Proof: Immediate by induction on the length of the execution. ■

We say that two operations are ***concurrent*** in a state if they are concurrent in the client

constraints graph of that state, and they ***compete*** in a state if they compete in the client constraints graph of the state.

In any reachable state of a memory automaton, only requested operations have responses.

Invariant 9.12 For $\mathcal{GM}$: $domain(resps) \subseteq reqs$.

Proof: Immediate by induction on the length of an execution. ■

Because $\mathcal{GM}$ is the most general memory automaton and $\mathcal{GC}$ is the most general clients automaton, the composition of any memory automaton and any clients automaton maintains the invariants of both $\mathcal{GM}$ and $\mathcal{GC}$. Furthermore, *resps* defines a "partial observer function" for *CCG*; that is, every return value recorded in *resps* is explained by some schedule of *CCG*.

Invariant 9.13 For $\mathcal{GM} \times \mathcal{GC}$: For every $x \in domain(resps)$, $resps(x) = retval(x, \alpha)$ for some $\alpha \in Sch(CCG)$.

Proof: We prove this invariant by induction on the length of an execution. It is trivially true in the initial state because $domain(resps) = \emptyset$.

If this invariant holds in s and $s \xrightarrow{\mathsf{request}(x,P,a)} s'$ then for $y \in domain(s'.resps) = domain(s.resps)$, there exists $\alpha \in Sch(s.CCG)$ such that $s'.resps(y) = s.resps(y) = retval(y, \alpha) = retval(y, \alpha{\cdot}x)$, and $\alpha{\cdot}x \in Sch(s'.CCG)$ since x, which was just added to *reqs*, does not precede any operations in $s'.CCG$. So the invariant holds in s'.

If the invariant holds in s and $s \xrightarrow{\mathsf{response}(x,v)} s'$ then we have $resps(x) = v = retval(x, \alpha)$ for some $\alpha \in Sch(s.CCG)$ from the precondition of $\mathsf{response}(x, v)$. For $y \in domain(s'.resps) - \{x\} = domain(s.resps)$, we have $s'.resps(y) = s.resps(y) = retval(y, \alpha)$ for some $\alpha \in Sch(s.CCG)$. Since $s'.CCG = s.CCG$, the invariant holds in s'. ■

9.3 Modeling Memory Systems in the Dynamic Framework

One of the advantages of using state machines to model memory systems is that describing a system as a state machine is relatively straightforward and well understood throughout computer science. For an I/O automaton model of a system, we also need to define the input and output actions that form the interface for the system. I/O automata have been used for many years to describe distributed systems and there is a mature theory supporting them [79].

To model the behavior of a program, we must have a model for the system that the program is executed on. Sequential systems typically have a simple model, in which the progress through the program is tracked by a program counter. This approach is commonly extended to concurrent systems by using multiple program counters.

Example 9.1 Consider the following program from Example 3.20:

```
int x = 0
sync int y = 0

P1: write(x,1)
```

```
    write(y,1)

P2: repeat
      t1 <- read(y)
    until t1 = 1
    read(x)
```

We model this program by two simple I/O automata, one for each process. The clients automaton is the composition of these two automata. We assume the processes are blocking; that is, processes that wait for a response before requesting the next operation. The output actions of each automaton are its request actions, and its input actions are response actions. Each automaton has its own program counter, which indicates the current operation and whether the operation has been requested. At the end of the program, the program counter is set to DONE. To make every operation unique, the automaton for P2 also has a counter *index*, which indexes each $\mathsf{read}(y)$ operation.

Because response actions are input actions, controlled by the memory, these actions have no precondition; a process must accept any response from the memory, but it can ignore the responses by not changing state when a response is received. In these automata, a response are ignored unless it is for the operation that was last requested.

P1

State Variables

$pc_1 \in \{\text{preWX}, \text{postWX}, \text{preWY}, \text{postWY}, \text{DONE}\}$, initially preWX.

Steps

Output $\mathsf{request}(\mathsf{write}(x,1), \emptyset, \text{NIL})$
Pre: $pc_1 = \text{preWX}$
Eff: $pc_1 \leftarrow \text{postWX}$

Output $\mathsf{request}(\mathsf{write}(y,1), \{\mathsf{write}(x,1)\}, \text{SYNC})$
Pre: $pc_1 = \text{preWY}$
Eff: $pc_1 \leftarrow \text{postWY}$

Input $\mathsf{response}(x,v)$
Eff: if $pc_1 = \text{postWX}$ and $x = \mathsf{write}(x,1)$
then $pc_1 \leftarrow \text{preWY}$
if $pc_1 = \text{postWY}$ and $x = \mathsf{write}(y,1)$
then $pc_1 \leftarrow \text{DONE}$

P2

State Variables

$pc_2 \in \{\text{preRY}, \text{postRY}, \text{preRX}, \text{postRX}, \text{DONE}\}$, initially preRY.

$index \in \mathbb{N}$, initially 0.

Steps

Output $\mathsf{request}(\mathsf{read}(y)_i, P, \text{SYNC})$
Pre: $pc_2 = \text{preRY}$
$i = index + 1$
$P = \begin{cases} \{\mathsf{read}(y)_{index}\} & \text{if } index > 0 \\ \emptyset & \text{otherwise} \end{cases}$
Eff: $index \leftarrow index + 1$
$pc_2 \leftarrow \text{postRY}$

Output $\mathsf{request}(\mathsf{read}(x), P, \text{NIL})$
Pre: $pc_2 = \text{preRX}$
$P = \{\mathsf{read}(y)_{index}\}$
Eff: $pc_2 \leftarrow \text{postRX}$

Input $\mathsf{response}(x,v)$
Eff: if $pc_2 = \text{postRY}$ and $x = \mathsf{read}(y_{index})$
then $pc_2 \leftarrow \begin{cases} \text{preRX} & \text{if } v = 1 \\ \text{preRY} & \text{otherwise} \end{cases}$
if $pc_2 = \text{postRX}$ and $x = \mathsf{read}(x)$
then $pc_2 \leftarrow \text{DONE}$

■

Example 9.2 We can also model nonblocking processes; that is, processes that do not wait for a response to a requested operation. The automata that model nonblocking processes for the program from Example 9.2 are:

P1

State Variables

$pc_1 \in \{\text{WX}, \text{WY}, \text{RY}\}$, initially WX.

Steps

Output request(write$(x, 1), \emptyset,$ NIL)
Pre: $pc_1 = \text{WX}$
Eff: $pc_1 \leftarrow \text{WY}$

Output request(write$(y, 1)$, {write$(x, 1)$}, SYNC)
Pre: $pc_1 = \text{WY}$
Eff: $pc_1 \leftarrow \text{RY}$

Input response(x, v)
Eff: None.

P2

State Variables

$pc_2 \in \{\text{RX}, \text{CONSUME}, \text{RY}\}$, initially RX.

$index \in \mathbb{N}$, initially 0.

Steps

Output request(read$(y)_i$, P, SYNC)
Pre: $pc_2 = \text{RX}$
$i = index + 1$
$P = \begin{cases} \{\text{read}(y)_{index}\} & \text{if } index > 0 \\ \emptyset & \text{otherwise} \end{cases}$
Eff: $index \leftarrow index + 1$

Output request(read(x), P, NIL)
Pre: $pc_2 = \text{CONSUME}$
$P = \{\text{read}(y)_{index}\}$
Eff: $pc_2 \leftarrow \text{RY}$

Input response(x, v)
Eff: if $x = \text{read}(y)_i$ for some $i \in \mathbb{N}$
and $v = 1$ and $pc_2 = \text{RX}$
then $pc_2 \leftarrow \text{CONSUME}$

■

As in the computation-centric framework, we can use a memory automaton as a specification for a memory system and a clients automaton to express a client restriction. In this case, we use the I/O automaton notion of implementation—that is, trace inclusion—to determine whether a memory system implements its specification. Similarly, a clients automaton meets a client restriction if it implements the automaton that models the client restriction. In the next two sections, we give examples of simple dynamic memory models and client restrictions, analogous to some of the models and client restrictions in Chapter 5, and we prove results analogous to those proved in that chapter.

9.4 Simple Dynamic Memory Models

In this section, we define dynamic analogues for some of the simple computation-centric memory models in Chapter 5. In particular, we define memory automata for sequential consistency, coherence and weak synchronization.

In this chapter, we define memory automata by adapting the definition of $\mathcal{GM}$, describing the parts that differ from $\mathcal{GM}$—often only an extra clause in the precondition of the response actions. It is often easier to express the extra precondition in terms of the post-state, for which we use primed variables as described in Section 9.1.2.

For sequential consistency, there must be a single schedule that explains all the return values. Thus, the dynamic model for sequential consistency is

$\mathcal{SC}$: Changes from $\mathcal{GM}$

Output response(x, v)
Pre: $x \in reqs - domain(resps)$
α explains $resps'$ for some schedule α of CCG.
Eff: $resps(x) \leftarrow v$

The use of *resps'* in the precondition means that the schedule must also explain v as a return value for x. So this clause supercedes the clause in the definition of $\mathcal{GM}$.

In any reachable state of $\mathcal{SC}$, some schedule of the client constraints graph explains all the return values.

Invariant 9.14 For $\mathcal{SC}$: There is a schedule of *CCG* that explains *resps*.

Proof: Immediate by induction on the length of an execution. ■

One aspect of this definition that may seem odd is that each response action may use a different schedule. The "real" schedule is never recorded in the state. Nonetheless, this automaton models sequential consistency because there is always some schedule of the client constraints graph that explains all the values returned so far.

If the schedule preceding an operation were fixed when the memory returns a value for that operation, the resulting model would be *linearizability* [53], which is strictly stronger than sequential consistency. However, linearizability and sequential consistency are indistinguishable in the computation-centric framework; a linearizable system is modeled by *SC*. We discuss the relationship between linearizability and sequential consistency in Section 9.6.

The automaton for coherent memory is similar to $\mathcal{SC}$, except that the schedule needs to explain only those operations on the same location. Formally, we have

$\mathcal{C}oh$**: Changes from** $\mathcal{GM}$

Output response(x, v)
  Pre: $x \in reqs - domain(resps)$
    α explains $resps'|_{x.loc}$ for some schedule α of *CCG*.
  Eff: $resps(x) \leftarrow v$

In any reachable state of $\mathcal{C}oh$ and for each location of the data type, there is a schedule of the client constraints graph that explains the return values of the operations on that location.

Invariant 9.15 For $\mathcal{C}oh$: For each location ℓ, there is a schedule of *CCG* that explains $resps|_\ell$.

Proof: Immediate by induction on the length of an execution. ■

Obviously, sequentially consistent memory implements coherence.

Lemma 9.16 $\mathcal{SC} \subseteq \mathcal{C}oh$.

Proof: Immediate from Corollary 9.10 because $\mathcal{SC}$ and $\mathcal{C}oh$ are identical except that the precondition on the response actions for $\mathcal{C}oh$ are weaker than for $\mathcal{SC}$. ■

For a weakly synchronized memory, all operations must see the same order for the synchronization operations. The system synchronizes an operation by fixing its order relative to the other synchronization operations, and it does not allow any operation to see the effects of a synchronization operation until it has been synchronized.

To model weak synchronization, we add to the generic memory automaton an internal synchronize(x) action and a state variable *so* that records the order in which operations are synchronized. The synchronize(x) action appends a synchronization operation x to the

sequence *so*, which represents the "synchronization order". The schedule that explains the return value of an operation must be consistent with the synchronization order, and synchronization operations may precede the operation in that schedule only if they have been synchronized. Formally, we have

WSync: **Changes from** $\mathcal{GM}$

Additional State Variable

$so \in O^*$, initially ϵ; the "synchronization order".

Additional and Modified Steps

Internal synchronize(x)
  Pre: $x \in reqs - elems(so)$
    $a(x) = \text{SYNC}$
    $\{y : a(y) = \text{SYNC} \wedge y \prec_c x\} \subseteq elems(so)$
  Eff: append x to so

Output response(x, v)
  Pre: $x \in reqs - domain(resps)$
    for some schedule α of CCG,
      $v = retval(x, \alpha)$
      α is consistent with $<_{so}$
      $y \leq_\alpha x \wedge a(y) = \text{SYNC} \implies y \in elems(so)$
  Eff: $resps(x) \leftarrow v$

9.5 Client Restrictions in the Dynamic Framework

In this section, we define clients automata that specify client restrictions. We retain the flavor of the computation-centric client restrictions by simply restricting the client constraints graphs that the automata may generate. We give clients automata that model the client restrictions of safety and race-freedom. We also prove that any memory automaton appears sequentially consistent to safe, or race-free, clients in the dynamic framework.

Like the memory automata in the previous section, we define the clients automata in this section by adapting the definition of the generic clients automaton $\mathcal{GC}$ and only showing the parts that differ. In this case, we typically augment the precondition of the request actions, again using primed variables for the value of the state variables in the post-state.

As in the computation-centric framework, we use determinacy and race-freedom to define restricted clients. A clients automaton is ***safe*** if the client constraints graph in every reachable state is determinate. It is ***(completely) race-free*** if the client constraints graph in every reachable state is completely race-free. We express these restrictions using $\mathcal{S}afe$ and $\mathcal{RF}$, where a clients automaton is safe if and only if it implements $\mathcal{S}afe$, and it is race-free if and only if it implements $\mathcal{RF}$. Formally,

$\mathcal{S}afe$: Changes from $\mathcal{GC}$

Output request(x, P, a)
Pre: $x \notin reqs$
$P \subseteq reqs$
CCG' is determinate
Eff: $reqs \leftarrow reqs \cup \{x\}$
$deps \leftarrow deps \cup \{(y, x) : y \in P\}$
$ann(x) \leftarrow a$

$\mathcal{RF}$: Changes from $\mathcal{GC}$

Output request(x, P, a)
Pre: $x \notin reqs$
$P \subseteq reqs$
CCG' is completely race-free
Eff: $reqs \leftarrow reqs \cup \{x\}$
$deps \leftarrow deps \cup \{(y, x) : y \in P\}$
$ann(x) \leftarrow a$

With these characterizations of safe and race-free clients, we can prove results analogous to those in Section 5.3. The main result is that any memory appears sequentially consistent to race-free clients. First we prove that race-free clients are safe, and that any memory appears sequentially consistent to safe clients.

Lemma 9.17 $\mathcal{RF} \subseteq \mathcal{S}afe$.

Proof: Immediate from Corollary 9.10 because every race-free computation is determinate. ■

Lemma 9.18 $\mathcal{GM} \times \mathcal{S}afe \subseteq \mathcal{SC} \times \mathcal{S}afe$.

Proof: $\mathcal{GM} \times \mathcal{S}afe$ and $\mathcal{SC} \times \mathcal{S}afe$ are identical except for the precondition of the response actions. By Invariant 9.13, for all $x \in domain(resps)$, there exists $\alpha \in Sch(CCG)$ such that $resps(x) = retval(x, \alpha)$. Since CCG is determinate in any reachable state of $\mathcal{GM} \times \mathcal{S}afe$, all schedules have the same return values for all operations, so there exists $\alpha \in Sch(CCG)$ such that for all $x \in domain(resps)$, $resps(x) = retval(x, \alpha)$. Thus, the preconditions for $\mathcal{GM}$ and $\mathcal{SC}$ are logically equivalent in all reachable states of $\mathcal{GM} \times \mathcal{S}afe$. By Corollary 9.10, $\mathcal{GM} \times \mathcal{S}afe \subseteq \mathcal{SC} \times \mathcal{S}afe$. ■

Theorem 9.19 $\mathcal{GM} \times \mathcal{RF} \subseteq \mathcal{SC} \times \mathcal{RF}$.

Proof: Immediate from Lemmas 9.17 and 9.18. ■

9.6 Relating the Two Frameworks

Because we have proposed two different frameworks for modeling and reasoning about memory systems, it is natural to inquire about the relationship between these frameworks. In this section, we show how to derive computation-centric models from dynamic models, and we discuss the relationship between corresponding models in each framework.

As we discussed in Section 9.3, it is usually straightforward to give a dynamic model of a system that corresponds closely to the actual operation of the system. Given automaton models for a memory system and its clients, we can formalize the intuition in Chapters 3 and 4 about how to construct the corresponding computation-centric models. For a clients automaton $\mathcal{C}$ and a memory automaton $\mathcal{M}$, we use $cc(\mathcal{C})$, $cc(\mathcal{M})$ and $cc(\mathcal{M} \times \mathcal{C})$ to denote the computation-centric versions of the clients, memory and composed system respectively.

For a clients automaton $\mathcal{C}$, the corresponding client restriction $cc(\mathcal{C})$ is the set of client constraints graphs in any reachable state of the automaton. Formally,

$$cc(\mathcal{C}) = \{s.CCG : s \text{ is a reachable state of } \mathcal{C}\}.$$

For a memory automaton $\mathcal{M}$, we cannot use the set of all pairs of $(CCG, resps)$ in any reachable state of $\mathcal{M}$ for the computation-centric model because *resps* may not be defined on all operations of *CCG*. So we only include those pairs that are observations, that is, the pairs for which there is a return value for every requested operation. Formally,

$$cc(\mathcal{M}) = \left\{ (s.CCG, s.resps) : s \text{ is a reachable state of } \mathcal{M} \text{ and } s.resps|_{s.reqs} \text{ is total} \right\}.$$

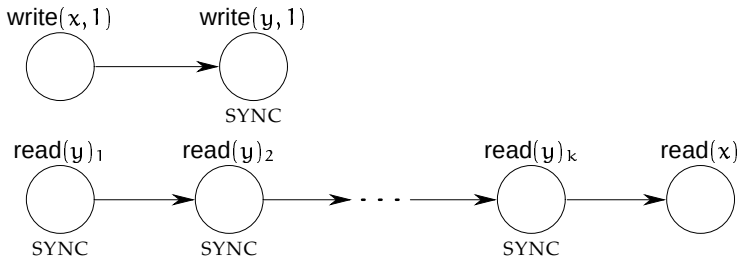
Because a memory system with a client restriction is just a more restricted memory system in the computation-centric framework, the computation-centric version of a memory automaton composed with a clients automaton is derived in the same way as the computation-centric version of a memory automaton. That is,

$$cc(\mathcal{M} \times \mathcal{C}) = \left\{ (s.CCG, s.resps) : s \text{ is a reachable state of } \mathcal{M} \times \mathcal{C} \text{ and } s.resps|_{s.reqs} \text{ is total} \right\}.$$

Some systems that exhibit different behaviors in the dynamic framework have the same computation-centric model. As mentioned earlier, although linearizability and sequential consistency do not allow the same set of behaviors—linearizability is strictly stronger—they are both modeled by *SC* in the compuation-centric framework. Linearizability and sequential consistency can be distinguished only by the time that responses are received: An operation must be scheduled after any operation that received a response before it was requested; that is, strict temporal precedence implies logical precedence. This implication is deliberately absent from observations in the computation-centric framework. A client that sees a value and requests an operation based on that value should include a precedence dependency to reflect that.

Because the computation-centric framework takes a static approach to modeling memory guarantees, the computation-centric versions of the automaton models do not capture the dynamic dependence that the computations may have on the values returned by the memory. Behaviors that are not possible because of this dependence may seem possible if we study the system only within the computation-centric framework. Unlike the distinction between linearizability and sequential consistency, the failure to capture the dependence of computations on the return values is a shortcoming of the computation-centric framework, and it is for this reason that we developed the automaton models for memory.

Example 9.3 Consider the clients automaton from Example 9.1. The computation generated by this automaton is



When composed with a sequentially consistent memory automaton, that is, a memory automaton that implements $\mathcal{SC}$, the return value for read(x) must be 1. However, *SC* admits observer functions with all the read operations returning 0 for this computation. ■

Despite the gap between computation-centric models and the systems they represent, the results derived in the computation-centric framework can be used directly to make claims about the automaton models. In particular, the computation-centric models are conservative in the following sense: Any observation of the real system, as modeled by memory and clients automata, is admissible according to the computation-centric models for the system.

Theorem 9.20 $cc(\mathcal{M} \times \mathcal{C}) \subseteq cc(\mathcal{M})|_{cc(\mathcal{C})}$.

Proof: If $(C, \rho) \in cc(\mathcal{M} \times \mathcal{C})$ then for some reachable state of $\mathcal{M} \times \mathcal{C}$, $CCG = C$ and $resps|_{reqs} = \rho$. So there is some reachable state of $\mathcal{M}$ with $CCG = C$ and $resps|_{reqs} = \rho$, and some reachable state of $\mathcal{C}$ with $CCG = C$. Thus, $(C, \rho) \in cc(\mathcal{M})$ and $C \in cc(\mathcal{C})$, and thus, $(C, \rho) \in cc(\mathcal{M})|_{cc(\mathcal{C})}$ ■

Theorem 9.20 implies that if we can prove within the computation-centric framework that a property holds for every observation of a system, that property holds for every observation of the system, as modeled by I/O automata.

Corollary 9.21 Suppose $\mathcal{M}$ is a memory automaton and $\mathcal{C}$ is a clients automaton, and let $M = cc(\mathcal{M})$ and $CR = cc(\mathcal{C})$ be their computation-centric versions. For a predicate P on observations, if $P(C, \rho)$ for all $(C, \rho) \in M|_{CR}$ then in any reachable state of $\mathcal{M} \times \mathcal{C}$ in which every requested operation has received a response, $P(CCG, resps|_{reqs})$.

9.7 From Computation-Centric to Dynamic Models

In this section, we show how to "translate" memory models and client restrictions from the computation-centric framework into the dynamic framework. We show that our translation preserves implementation under a client restriction. In the next section, we define a more careful translation that is sometimes necessary for memory models that are not constructible.

The basic idea of the translation is simple: A computation-centric client restriction translates to a clients automaton that generates only computations that satisfy the restriction. A computation-centric memory model translates to a memory automaton that returns values consistent with some admissible observer function for the client constraints graph.

Formally, let $\mathfrak{A}$ be the set of all I/O automata. The ***translation*** of a computation-centric memory model $M \in \mathfrak{M}_A^{\mathcal{D}}$ by the function $dyn\colon \mathfrak{M}_A^{\mathcal{D}} \to \mathfrak{A}$ is a memory automaton $dyn(M)$ that is identical to $\mathcal{GM}$ except for an additional clause in the precondition of the response(x, v) action that restricts the return values to be consistent with some observation admissible according to M.

$dyn(M)$**: Changes from** $\mathcal{GM}$

Output response(x, v)
  Pre: $x \in reqs - domain(resps)$
    $v = retval(x, \alpha)$ for some schedule α of CCG
    $(CCG, \rho) \in M$ for some extension ρ of $resps'$
  Eff: $resps(x) \leftarrow v$

The new precondition ensures that the value returned is part of a total response allowed by the memory model. Because $resps'$ is the value of $resps$ in the post-state, it includes the

assignment of v to $resps(x)$. Because the memory responds to one operation at a time, we cannot require $(CCG, resps')$ to be an admissible observation of M; some operations may not yet have responses. Instead, we require only that there be some extension of $resps'$ in $M[CCG]$.

In any reachable state of $dyn(M)$, M admits some observer function for CCG that is consistent with all the values returned for operations so far.

Invariant 9.22 For $dyn(M)$: $(CCG, \rho) \in M$ for some extension ρ of $resps$.

Proof: Immediate by induction on the length of an execution. The invariant holds in the initial state because the empty computation with the null function is admissible according to any memory model. ■

$\mathcal{SC}$ and $\mathcal{C}oh$ from Section 9.4 are equivalent to the translations of SC and Coh by dyn.

Lemma 9.23 $\mathcal{SC} \equiv dyn(SC)$ and $\mathcal{C}oh \equiv dyn(Coh)$.

Proof: This is trivial for $\mathcal{SC}$ and $dyn(SC)$ because the two automata are identical; the preconditions for their response actions are logically equivalent.

For $\mathcal{C}oh$ and $dyn(Coh)$, the precondition in $\mathcal{C}oh$ is implied by the precondition in $dyn(Coh)$, but the converse is not true. However, by Invariant 9.15, in any reachable state of $\mathcal{C}oh$, for each location ℓ, there is a schedule of CCG that explains $resps|_\ell$. Thus, $(CCG, \rho) \in Coh$ for some extension ρ of $resps$, which implies that the preconditions are logically equivalent in any reachable state. So, by Corollary 9.10, $\mathcal{C}oh \equiv dyn(Coh)$. ■

Translation by dyn preserves the implementation relation.

Theorem 9.24 If M implements M' then $dyn(M)$ implements $dyn(M')$.

Proof: Since $dyn(M)$ and $dyn(M')$ are identical except for the precondition of the response actions and since $M \subseteq M'$, we have $dyn(M) \subseteq dyn(M')$ by Corollary 9.10. ■

We translate client restrictions in a similar fashion, abusing notation by using the same name for the translation function. Formally, the ***translation*** of a computation-centric client restriction $CR \in 2^{\mathfrak{C}^{\mathcal{D}}_{\mathsf{A}}}$ by $dyn\colon 2^{\mathfrak{C}^{\mathcal{D}}_{\mathsf{A}}} \to \mathfrak{A}$ is a clients automaton $dyn(CR)$ that is identical to $\mathcal{GC}(\mathcal{D}, \mathsf{A})$ except for an additional clause in the precondition of the request action to ensure the client constraints graph defined is a computation allowed by the restriction.

$dyn(CR)$**: Changes from** $\mathcal{GC}$

Output request(x, P, a)
  Pre: $x \notin reqs$
    $P \subseteq reqs$
    $CCG' \in CR$
  Eff: $reqs \leftarrow reqs \cup \{x\}$
    $deps \leftarrow deps \cup \{(y, x) : y \in P\}$
    $ann(x) \leftarrow a$

The client constraints graph in any reachable state of $dyn(CR)$ is in CR.

Invariant 9.25 For $dyn(CR)$: $CCG \in CR$.

Proof: Immediate by induction on the length of an execution. ■

The clients automata $\mathcal{S}afe$ and $\mathcal{RF}$ from Section 9.4 are identical to the translations of *Safe* and *RF* by *dyn*.

Lemma 9.26 $\mathcal{S}afe = dyn(Safe)$ and $\mathcal{RF} = dyn(RF)$.

Proof: Immediate from the definitions. ■

Translation by *dyn* preserves the relative restrictiveness of clients.

Theorem 9.27 If CR is more restrictive than CR' then $dyn(CR)$ implements $dyn(CR')$.

Proof: Immediate from Corollary 9.10 and the definitions. ■

The translation of a computation-centric model and its projection under a client restriction are equivalent when composed with the translation of the client restriction.

Lemma 9.28 $dyn(M|_{CR}) \times dyn(CR) \equiv dyn(M) \times dyn(CR)$

Proof: Let $\mathcal{A} = dyn(M) \times dyn(CR)$ and $\mathcal{B} = dyn(M|_{CR}) \times dyn(CR)$. First, note that $M|_{CR} \subseteq M$, so $dyn(M|_{CR}) \subseteq dyn(M)$ by Theorem 9.24. Thus, by Theorem 9.7, $\mathcal{B} \subseteq \mathcal{A}$.

Now we prove that the identity function is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$. The start states are identical, as are the steps involving request actions, so we only need to check that if s is a reachable state of $\mathcal{A}$ and $s \xrightarrow{\mathsf{response}(x,v)}_{\mathcal{A}} s'$ then $s \xrightarrow{\mathsf{response}(x,v)}_{\mathcal{B}} s'$. By the precondition of $\mathsf{response}(x, v)$ in $\mathcal{A}$, there is an extension ρ of $s'.resps$ such that $(s.CCG, \rho) \in M$. By Invariant 9.25, $s.CCG \in CR$, so by the definition of $M|_{CR}$, $(s.CCG, \rho) \in M|_{CR}$. Since the effect clause of $\mathsf{response}(x, v)$ is the same in $\mathcal{A}$ and $\mathcal{B}$, we have $s \xrightarrow{\mathsf{response}(x,v)}_{\mathcal{B}} s'$, as required. ■

We now prove the main theorem of this section: Translation under *dyn* preserves implementation under a client restriction when the resulting memory models are composed with the translation of the client restriction.

Theorem 9.29 If M implements M' under CR then $dyn(M) \times dyn(CR) \subseteq dyn(M') \times dyn(CR)$.

Proof: By the definition of implementation under CR, $M|_{CR}$ implements $M'|_{CR}$. By Lemma 9.28 and 9.24, $dyn(M) \times dyn(CR) \equiv dyn(M|_{CR}) \times dyn(CR) \subseteq dyn(M'|_{CR}) \times dyn(CR) \equiv dyn(M') \times dyn(CR)$. ■

9.8 From Computation-Centric to Dynamic Models, Part II

Because the computation-centric framework supports a postmortem analysis of concurrent systems, the computations in a client restriction model the logical structure of finished executions and memory models specify admissible observer functions for these computations. However, neither the client restriction nor the memory model may be appropriate for modeling computations and observations in intermediate states of a program's execution. For example, there may be no way to construct the desired final computation one operation at a time without going through intermediate computations that do not satisfy the client restriction. In this section, we refine the translation from the previous section to

handle client restrictions and memory models that are not applicable to intermediate execution states. We show that the two translations are equivalent when the client restrictions and memory models are applicable to intermediate states.

We consider the translation for client restrictions first. For many client restrictions, there is no way to construct the allowed computations one operation at a time without going through intermediate computations that do not satisfy the restriction. For example, in a well-formed computation with locks, every acquire has a matching release, which is not true for computations in any state immediately following the acquisition of a lock. Informally, we want to allow the clients to request operations as long as they are building up towards a computation allowed by the client restriction.

Formally, the ***translation*** of a computation-centric client restriction CR by $dyn^*\colon 2^{\mathfrak{C}_A^{\mathcal{D}}} \to \mathfrak{A}$ is a clients automaton $dyn^*(CR)$ that is identical to $\mathcal{GC}(\mathcal{D}, A)$ except for an additional clause in the precondition of the request action that requires the client constraints graph defined to be a prefix of a computation allowed by the restriction.

$dyn^*(CR)$**: Changes from** $\mathcal{GC}$

Output request(x, P, a)
  Pre: $x \notin reqs$
    $P \subseteq reqs$
    CCG' is a prefix of some $C \in CR$
  Eff: $reqs \leftarrow reqs \cup \{x\}$
    $deps \leftarrow deps \cup \{(y, x) : y \in P\}$
    $ann(x) \leftarrow a$

Similarly, a computation-centric memory model that assumes a client restriction CR may not specify any observations for the intermediate computations generated by the clients automaton. However, the dynamic version of such a memory should still be able to respond to operations, as long as it will not "get stuck" as the computation is extended.[2] To model this, we need to specify what restriction the memory assumes. These restrictions are typically the well-formedness conditions of the system.

Formally, the ***translation*** of a computation-centric memory model M ***assuming*** WF by $dyn^*\colon \mathfrak{M}_A^{\mathcal{D}} \times 2^{\mathfrak{C}_A^{\mathcal{D}}} \to \mathfrak{A}$ is a memory automaton $dyn^*(M, WF)$ that is identical to $\mathcal{GM}$ except for an additional clause in the precondition of the response(x, v) action that restricts the return values to be consistent with some observation allowed by M *for every computation in* WF *that extends* CCG.

$dyn^*(M, WF)$**: Changes from** $\mathcal{GM}$

Output response(x, v)
  Pre: $x \in reqs - domain(resps)$
    $v = retval(x, \alpha)$ for some schedule α of CCG
    for every $C \in WF$ that extends CCG, there exists $\rho \in M[C]$ that extends $resps'$
  Eff: $resps(x) \leftarrow v$

As with the simpler translation by dyn, translation by dyn^* preserves implementation. The following theorem is analogous to Theorem 9.24.

[2] We can also define a less conservative translation in which the system can return a value for an operation as long as it is *might* not get stuck.

Theorem 9.30 If $M \subseteq M'$ then $dyn^*(M, WF) \subseteq dyn^*(M', WF)$.

Proof: Immediate from definition and Corollary 9.10. ■

Strengthening the well-formedness conditions weakens the resulting dynamic memory model.

Theorem 9.31 If $WF \subseteq WF'$ then $dyn^*(M, WF') \subseteq dyn^*(M, WF)$.

Proof: Immediate from definition and Corollary 9.10. ■

This theorem makes sense because the system with stronger well-formedness conditions does not need to guarantee consistency on as many computations; the system implementors make stronger assumptions about the behavior of the clients.

The next theorem is the analogue of Theorem 9.27: Translation by dyn^* preserves the relative restrictiveness of clients.

Theorem 9.32 If $CR \subseteq CR'$ then $dyn^*(CR)$ implements $dyn^*(CR')$.

Proof: Again, immediate from Corollary 9.10 and the definitions. ■

When translating a memory model under a client restriction by dyn^*, we do not translate the memory model and the client restriction separately, as we did when translating by dyn. Instead, the client restriction is used as the assumed well-formedness condition of the system. Thus, the next two results, analogues of Lemma 9.28 and Theorem 9.29, do not consider the composition of a memory and a clients automaton, as the results in the previous section did.

When assuming a client restriction, the translation of a memory model is the same as the translation of the memory model under that restriction.

Lemma 9.33 $dyn^*(M|_{CR}, CR) \equiv dyn^*(M, CR)$.

Proof: Because $M|_{CR}[\mathsf{C}] = M[\mathsf{C}]$ when $\mathsf{C} \in CR$, the two automata are identical. ■

Translation under dyn^* preserves implementation under a client restriction.

Theorem 9.34 If M implements M' under CR then $dyn^*(M, CR) \subseteq dyn^*(M', CR)$.

Proof: By the definition of implements under CR, $M|_{CR} \subseteq M'|_{CR}$, so by Lemma 9.33 and Theorem 9.30, $dyn^*(M, CR) \equiv dyn^*(M|_{CR}, CR) \subseteq dyn^*(M'|_{CR}, CR) \equiv dyn^*(M', CR)$. ■

We now show how to relate the two translations for memory models. It is easy to see that the new translation of client restrictions produces weaker clients automata than the old translations.

Theorem 9.35 $dyn(CR) \subseteq dyn^*(CR)$.

Proof: Immediate from the definition because any computation is a prefix of itself. ■

We cannot compare the translations of memory models so easily because the new translation has an additional parameter, the assumed well-formedness condition. However, if we do not make any well-formedness assumptions, then the translation by dyn^* is at least as strong as the translation by dyn because it must guarantee that the return values are consistent with every possible extension of the client constraints graph.

Theorem 9.36 $dyn^*(M, \mathfrak{C}_A^{\mathcal{D}}) \subseteq dyn(M)$.

Proof: Immediate from Corollary 9.10 since the precondition of $\mathsf{response}(x, v)$ in $dyn^*(M, \mathfrak{C}_A^{\mathcal{D}})$ implies the one in $dyn(M)$. ■

If the computation-centric model is constructible, the two translations are identical.

Theorem 9.37 If M is constructible then $dyn^*(M, \mathfrak{C}_A^{\mathcal{D}}) = dyn(M)$.

Proof: Immediate from Corollary 9.10 since the precondition of $\mathsf{response}(x, v)$ in $dyn(M)$ implies the one in $dyn^*(M, \mathfrak{C}_A^{\mathcal{D}})$ when M is constructible. ■

We do not have an exact condition that characterizes when the translations are equivalent for client restrictions. One sufficient condition is *prefix-closure*. A client restriction CR is ***prefix-closed*** if for all $C \in CR$, every prefix of C is also in CR. Any computation that is allowed by a prefix-closed client restriction can be constructed one operation at a time, in any order consistent with the precedence order, with the intermediate computations satisfying the restriction.

Theorem 9.38 If CR is prefix-closed then $dyn^*(CR) = dyn(CR)$.

Proof: Immediate from the definitions. ■

The translation by dyn^* of a memory model assuming a prefix-closed client restriction implements the translation of the memory model by dyn when they are composed with the translation of the client restriction.

Theorem 9.39 If CR is prefix-closed then $dyn^*(M, CR) \times dyn^*(CR) \subseteq dyn(M) \times dyn(CR)$.

Proof: By Theorem 9.38, $dyn^*(CR) = dyn(CR)$, so by Invariant 9.25, $CCG \in CR$ in any reachable state of both composite automata. Thus, in any reachable state, the precondition of the $\mathsf{response}$ action in $dyn^*(M, CR)$ implies the precondition of the $\mathsf{response}$ action in $dyn(M)$, which proves the theorem. ■

Combining the previous results, we have that the translation by dyn^* of a constructible memory model assuming a prefix-closed client restriction is equivalent to the translation of the memory model by dyn when they are composed with the translation of the client restriction.

Theorem 9.40 If CR is prefix-closed and M is constructible for CR then $dyn^*(M, CR) \times dyn^*(CR) \equiv dyn(M) \times dyn(CR)$.

Proof: By Theorems 9.37 and 9.38, $dyn(M) \times dyn(CR) = dyn^*(M, \mathfrak{C}_A^{\mathcal{D}}) \times dyn^*(CR)$. By Theorem 9.31, $dyn^*(M, \mathfrak{C}_A^{\mathcal{D}}) \subseteq dyn^*(M, CR)$ since $CR \subseteq \mathfrak{C}_A^{\mathcal{D}}$, so $dyn(M) \times dyn(CR) \subseteq dyn^*(M, CR) \times dyn^*(CR)$. Thus, by Theorem 9.38, $dyn^*(M, CR) \times dyn^*(CR) = dyn(M) \times dyn(CR)$. ■

Chapter 10

Conclusions and Future Work

In this thesis, we have presented the computation-centric framework for reasoning about memory consistency. We have demonstrated its utility by specifying and reasoning about a variety of memory models, including familiar models, such as sequential consistency and coherence, and new ones, such as weak sequential locking and internally consistent transactions. We have proven theorems that identify conditions under which systems with weak consistency guarantees appear to have strong guarantees.

However, the true value of any framework for reasoning about systems lie in its impact on real systems. In this chapter, we discuss how the work we present here may impact real systems and point out promising directions for further study.

10.1 Implications on Memory System Design and Use

There are at least three ways in which this framework may have impact on real systems. First, it may be used by programmers of these systems to reason about their programs. Second, it may be used by system designers to determine what consistency properties to guarantee and to express these guarantees precisely and unambiguously. Third, it may be used by system implementors to verify their implementations are correct. In this section, we describe why we believe the computation-centric framework could be used in these ways. Actual use, of course, can only take place over time.

Use by programmers. Several of the theorems in this thesis have been directed towards programmers, who are the main audience for the computation-centric framework. For example, the main result of Chapter 7, Theorem 7.25, says that programmers can assume sequential consistency with locks as long as they write programs that are data-race-free under locking and the system guarantees weak sequential locking. Since weak sequential locking is a very weak model, any system that provides locks should implement it. Thus, the only burden on programmers is to verify that their programs are data-race-free under locking, which is generally recommended for concurrent programming with locks.

We do not expect that programmers will always formally prove that their programs are correct using the computation-centric framework. Rather, the computation-centric framework provides a nice basis for informal "back of the envelope" reasoning, as well as for

rigorous proof. Having a precise definition for data-race-freedom under locking means that when there is ambiguity, one can resort to the formal definition.

Many relaxed memory models that have been proposed have been too difficult to understand [56]. We believe that the computation-centric framework can be readily grasped by programmers with a minimal discrete math background. The only formal background necessary is a bit of graph and set theory. More importantly, computation provide a simple visual way to understand the structure of a program's execution that we believe programmers will find easy to map to their understanding of the program.

Use by designers. One of the difficulties with memory consistency models is that any given system guarantees various consistency properties. Most models do not separate out these properties so that designers can understand the cost and benefit of each property.

For example, as we saw in Sections 5.3, 6.7 and 7.6, coherence is not necessary to guarantee sequential consistency in almost any case. Nonetheless, almost all multiprocessors guarantee coherence. In earlier work on computation-centric memory models [40], we showed that any constructible memory model that implements a particular form of *dag consistency* guarantees coherence. Because dag consistency is a very weak model, specified by ruling out anomalous behaviors, Frigo concluded that every "reasonable" memory model should guarantee coherence [39]. However, the observation above gives evidence against this conclusion. Although a system may exhibit unreasonable behavior for some computations, as long as it guarantees weak sequential locking, which does not imply coherence, it will execute any data-race-free program correctly.

Similarly, release consistency and weak ordering require all lock access to be synchronized, even accesses of different locks. However, weak sequential locking requires accesses to be synchronized only if they access the same lock. Clearly, this allows more efficient implementations. If a lock is implemented by synchronized access to a single memory location, the system need not synchronize operations performed on different locations to execute data-race-free programs correctly.

A multiprocessor that provides fine-grained synchronization can take advantage of this weak synchronization requirement to improve performance. Indeed, several shared memory multiprocessors have supported fine-grained synchronization, including HEP [102], Monsoon [92], Tera [10], the Wisconsin Multicube [48] and the MIT Alewife machine [7]. However, most modern multiprocessors memory models, including the SPARC relaxed memory ordering [107], the PowerPC [84], the Alpha [101] and release consistency [43], provide only coarse-grained synchronization that does not distinguish memory operations by location. Supporting fine-grained synchronization requires greater hardware complexity, but there is evidence that the performance advantage is worth the added complexity [65]. To the best of my knowledge, more recent literature does not address this question further.

On the other side of the memory interface, designers (and implementors) of shared memory multiprocessors seem to assume that sequential consistency is good enough, presumably because sequential consistency can be used to implement locks and other synchronization mechanisms. Although there is a rich body of literature, derived from multiprogramming, on using sequentially consistent memory, the consensus is that it is hard [70]. Books on concurrent programming teach high level constructs to make this task easier.

In fact, almost all multiprocessor systems provide hardware support for some kind of synchronization. But the implications of these mechanisms on memory consistency is only described in terms of guaranteeing sequential consistency. We want to make the power of these mechanisms directly available to the programmer through the memory model.

Use by implementors. We have not done much work on results that are directly useful to system implementors. Nonetheless, we provide a clear and precise specification of a memory system, so it is possible for implementors to verify their implementations against these specifications.

10.2 Future Work

Despite its length and breadth, this thesis presents a very preliminary investigation into how to reason about shared memory systems. The discussion sections of the chapters include many suggestions for extending the computation-centric framework. In addition, there are whole areas that we are leaving unexamined, including issues of language design and compilation. Indeed, there is much more to do than has been done, and this field is ripe for the picking. In this section, we point out some of the directions we believe will be the most fruitful to explore.

Data types. We focus on one property—independence—of data type operators. There are other properties, such as transparency and obliteration,[1] that may simplify the analysis of concurrent systems. Allowing blocking and nondeterministic operators will also be necessary to model the data types of some memory systems. Blocking operators could be used to implement locks in memory, and modeling transactions as providing abstract data types may require nondeterministic operators.

Alternative notions of operator sequence equivalence are also important to model, especially for transactional memories, which may want to "hide" the return values for internal operators.

One idea that we began to explore and looks promising is to "iterate" data types. An *iterated data type* is derived from a data type by allowing sequences of operators to be requested. In the concurrent setting, these sequences would have to be applied atomically; thus, iterated data types can be used to model the abstract data types implemented by transactions.

Computations. We need to bridge the gap better between computations and programs. This issue is discussed in Section 3.5. We model precedence dependencies specially instead of modeling them as annotations. Are there other constraints that we should treat similarly? For example, we might distinguish "external" and "internal" operations and use a notion of operator sequence equivalence that exploits this distinction.

[1] See page 29 for an informal definition of these properties.

It may also be helpful to give annotations more structure by defining annotation types, which specify the well-formedness condition and scheduling constraints in addition to the possible annotations.

Memory models. One intriguing possibility is to model the dynamic interaction between the memory and its clients directly over the computation-centric framework instead of imbedding the computations within a state machine model. One simple approach is to model a system's execution as a sequence of intermediate observations; this approach requires us to allow partial observer functions.

Our models also completely neglect liveness. It may be easy to incorporate the liveness aspects of I/O automata to study liveness in the state machine models. It would be good to have a way to study liveness directly in the computation-centric framework, perhaps indicating that some observations are "quiescent" while others require further action.

Program models. The shortcoming of the computation-centric framework in capturing the dynamic interaction between the memory and its clients is a problem in one direction: Clients, as modeled by computations, do not see the return values. Client restrictions are a crude tool for modeling clients, especially with the kinds of computations we have specified thus far.

Value synchronization, as suggested at the end of Section 5.5, is one possible way to improve the modeling of clients without jettisoning the computation-centric framework completely. With value synchronization, the values seen for earlier operations may be incorporated into the annotations of later operations.

Specific memory models and systems. The task of modeling memory systems is never finished because new systems, and new ways to program these systems, are continually being developed. An exhaustive cataloging and analysis of even those already proposed and implemented would be a herculean task,[2] and not one I suggest to undertake. However, several particular models, or relaxations of models already specified here, seem worth examining.

One useful synchronization concept that we do not model is *stability*. Informally, an operation x is *stable* when the entire system agrees on the order of the operations that precede x. This order may not explain the values returned for those operations—such constraints must be expressed by the memory model: the system may return "tentative" values for some operations—but the schedules that explain later operations will have the same prefix.

Besides mutex and shared/exclusive locks, there are other kinds of locks and other locking disciplines that are worth considering, in part because they are widely used. The simplest extension is to relax the well-formedness conditions for mutex locks, so that locked sections need not be enclosures of the computation. Other possibilities are to model the locking associated with lower degrees of isolation, or to model a set of hierarchical locks, where acquiring a lock requires the acquisition of all higher locks.

[2] Probably the slaying of the Lernean Hydra or the cleaning of the Augean stables.

Another extension relating to memories with locks is to decouple the passing of locks from the precedence dependencies. For example, a thread may spawn a new thread but not pass the locks that it holds to that thread.

As we suggest at the end of Chapter 8, transactions are a promising concept for providing a foundation for concurrent computing. Section 8.9 discusses several ways to explore transactional models further. I believe that further study into what is possible with weak transactional models, both those defined here and others as yet undefined, will be greatly rewarded.

Other issues. To avoid language design considerations, we designed computations to be flexible enough to model almost any kind of constraint that might be expressed in a concurrent programming language. However, we do not view the choice of language as a minor point to be decided when everything else has been worked out. The design or choice of proper constructs and disciplines for organizing a concurrent program may be the most important issue for concurrent systems. We focused on the memory consistency semantics because it is not clear what, if any, constructs are truly useful. If shared memory is the right way for concurrent threads to communicate, then a framework for reasoning about memory semantics is necessary to assess the various candidates.

Compilation is important because of the gap it produces between the program and the actual code executed on the system. However, program semantics should not be defined by their compilers or the systems the programs are executed on. Rather, every programming language should have a precise abstract execution model that is independent of the system. This execution model includes the memory consistency model in its specification. We believe that much of the difficulty and confusion with compiling for concurrent systems is a result of not having such a model, without which there can be no precise notion of a correct compilation of a program. Just-in-time compilation throws a new twist into the compilation quagmire, because the way code is compiled may depend on the values returned for earlier operations. Nonetheless, as long as the program execution model is invariant under changes to the compiler or the system, as it should be, reasoning about a program not be affected by compilation.

Compilation should affect memory semantics, however, in the same way that architecture affects memory semantics: The execution model for a programming language should be efficiently implementable using current compilation and architectural techniques. Understanding these techniques, and their impact on memory consistency, is essential to the design of memory models.

Another important issue that we have not discussed is performance. Good analytic performance models are rare—almost non-existent—for concurrent systems. We need more and better models to predict performance.

Finally, one of the advantages of formal methods is that we can use formal tools—theorem provers, verifiers, perhaps even automatic code generators—to help in designing, reasoning and building systems. There is a lot of work in this area for all kinds of systems, including shared memory systems.

Bibliography

[1] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the International Conference on Parallel Processing*, pages 147–150, August 1990.

[4] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th ACM Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[5] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[6] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th ACM Annual International Symposium on Computer Architecture*, pages 234–243, May 1991.

[7] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd ACM Annual International Symposium on Computer Architecture*, pages 2–13, Santa Margherita, Ligure, Italy, June 1995.

[8] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 251–260, June 1993.

[9] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[10] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, June 1990.

[11] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 241–250, July 1993.

[12] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 679–690, 1992.

[13] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall International, Inc., Englewood Cliffs, NJ, 1982.

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., Reading, MA, 1987.

[15] Brian Bershad, Matthew Zekauskas, and Wayne Sawdon. The Midway distributed shared memory system. In *Digest of Papers from the 38th IEEE Computer Society International Conference (Spring COMPCON)*, pages 528–537, San Francisco, CA, February 1993.

[16] Andrew Birrell. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation, January 1989.

[17] Andrew Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[18] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of IPPS'96, The 10th International Parallel Processing Symposium*, pages 132–141, Honolulu, Hawaii, April 1996.

[19] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[20] Thomas Bräunl. *Parallel Programming: An Introduction*. Prentice Hall, Cambridge, England, 1993.

[21] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[22] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[23] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[24] Guang-Ien Cheng. Algorithms for data-race detection in multithreaded programs. Master's thesis, Massachusetts Institute of Technology, June 1998.

[25] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, Puerto Vallarta, Mexico, June/July 1998.

[26] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1998. with Anoop Gupta.

[27] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

[28] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[29] Edsger W. Dijkstra. Solution of a problem in concurrency control. *Communications of the ACM*, 8:569, 1965.

[30] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[31] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, May 1991.

[32] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[33] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th ACM Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[34] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, pages 9–21, February 1988.

[35] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[36] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data service. *Theoretical Computer Science*, 220(1):113–156, June 1999. Special Issue on Distributed Algorithms. A preliminary version appears in PODC'96.

[37] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–11, Newport, Rhode Island, June 1997.

[38] Roy Friedman. *Consistency Conditions for Distributed Shared Memories*. PhD thesis, The Technion, Haifa, Israel, June 1994.

[39] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Massachusetts Institute of Technology, January 1998.

[40] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 240–249, Puerto Vallarta, Mexico, June/July 1998.

[41] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[42] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[43] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th ACM Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[44] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 158–168, June 1992.

[45] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 292–303, July 1991.

[46] Alex Gontmakher and Assaf Schuster. Java consistency: Non-operational characterizations for Java memory behavior. *ACM Transactions on Computer Systems*, 18(4):336–386, November 2000.

[47] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[48] James R. Goodman and Philip J. Woest. The wisconsin multicube: A new large scale cache-coherent multiprocessor. In *Proceedings of the 15th ACM Annual International Symposium on Computer Architecture*, pages 422–431, Hawaii, June 1988.

[49] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, second edition, 2000.

[50] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared database. In *Modeling in Data Base Management Systems*. Elsevier North-Holland, Amsterdam, 1976.

[51] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[52] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[53] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[54] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems*, pages 349–356, Washington, DC, October 1997.

[55] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models Part I: Definitions and comparisons. Technical Report 1998-612-03, Department of Computer Science, The University of Calgary, Calgary, Alberta, January 1998.

[56] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31(8):28–34, August 1998.

[57] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[58] IBM System/370 Principles of Operation, May 1983. Publication Number GA22-7000-9, File Number S370-01.

[59] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. *Theory of Computing Systems*, 31(4):451–473, July 1998.

[60] Joint ISCA/PODC panel and discussion, May 1996. Held at the Federated Computing Research Conference in Philadelphia, PA.

[61] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[62] Jalal Y. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems*. PhD thesis, The University of Calgary, Calgary, Alberta, January 2000.

[63] Peter J. Keheler. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 91–98, Hong Kong, May 1996.

[64] Peter J. Keheler, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Conference Proceedings*, pages 115–132, San Francisco, CA, January 1994.

[65] David Kranz, Beng-Hong Lim, Anant Agarwal, and Donald Yeung. Low-cost support for fine-grain synchronization in multiprocessors. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, 1994.

[66] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Exploiting the semantics of distributed services. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[67] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143, March 1977.

[68] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[69] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Laboratory, Palo Alto, CA, May 1989.

[70] Butler Lampson. Course notes for 6.826 (Principles of Computer Systems). Developed with William Weihl and Nancy Lynch; also taught with Alex Shvartsman and Martin Rinard. Available at `ftp://theory.lcs.mit.edu/pub/classes/6.826/www/6.826-top.html`., 1999.

[71] Dan Lenoski, James Laudon, Kouroosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[72] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, San Francisco, CA, 1995.

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

[74] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, September 1988.

[75] David B. Lomet. Process structuring, synchronization and recovery using atomic actions. *ACM SIGPLAN Notices*, 12(3):128–137, March 1977.

[76] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[77] Victor Luchangco. Precedence-based memory models. In Marios Mavronicolas and Philippas Tsigas, editors, *Distributed Algorithms: 11th International Workshop, WDAG'97, Proceedings*, volume 1320 of *Lecture Notes in Computer Science*, pages 215–229, Saarbrücken, Germany, September 1997. Springer-Verlag.

[78] Victor Luchangco. Modeling weakly consistent memories with locks. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 332–333, Crete, Greece, July 2001.

[79] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[80] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[81] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[82] Jan-Willem Maessen, Arvind, and Xiaowei Shen. Improving the Java memory model using (CRF). In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–12, Minneapolis, MN, October 2000. CSG-Memo-428.

[83] Mesaac Makpangou, Guillaume Pierre, Christian Khoury, and Neilze Dorta. Replicated directory service for weakly consistent distributed caches. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 92–100, Austin, Texas, May 1999.

[84] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[85] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 Conference on Supercomputing*, pages 24–33, Oconomowoc, Wisconsin, November 1991. IEEE Computer Society Press.

[86] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[87] David Mosberger. Memory consistency models. *ACM Operating Systems Review*, 17(1):18–26, January 1993.

[88] Robert H. B. Netzer and Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the 1992 International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[89] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 133–144, April 1991.

[90] Robert H. B. Netzer and Barton P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[91] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[92] Greg M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th ACM Annual International Symposium on Computer Architecture*, Seattle, WA, June 1990.

[93] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*, San Francisco, CA, June 1999.

[94] William Pugh and Jeremy Manson. Semantics of multithreaded Java. Available at `http://www.cs.umd.edu/~pugh/java/memoryModel/`, January 2001.

[95] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[96] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multithreaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[97] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[98] Xiaowei Shen, November 2000. Personal Communication.

[99] Xiaowei Shen. *Design and Verification of Adaptive Cache Coherence Protocols*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 2000.

[100] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A new memory model for architects and compiler writers. In *Proceedings of the 26th ACM Annual International Symposium on Computer Architecture*, pages 150–161, Atlanta, Georgia, 1999. CSG-Memo-413.

[101] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual*. Butterworth-Heinemann, second edition, 1995.

[102] B. J. Smith. The architecture of HEP. In J. S. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 41–55. MIT Press, Cambridge, MA, 1985.

[103] C. R. Snow. *Concurrent Programming*. Cambridge University Press, Cambridge, England, 1992.

[104] SPARC architecture manual, January 1991. No. 800-199-112, Version 8.

[105] Supercomputing Technologies Group. Cilk 5.3.1. Reference manual, MIT Laboratory for Computer Science, Cambridge, MA, June 1998.

[106] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, Texas, September 1994.

[107] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., 1994.

[108] R. Zucker and J.-L. Baer. A performance study of memory consistency models. In *Proceedings of the 19th ACM Annual International Symposium on Computer Architecture*, pages 2–12, May 1992.

Index