Limitations and Capabilities of Weak Memory Consistency Systems

by

Jalal Y. Kawash

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
January, 2000

© Jalal Y. Kawash 2000
Abstract

This dissertation develops and exploits a formalism for specifying memory consistency models. This formalism lays down the foundations for describing memory consistency models at various levels, and develops techniques to prove the equivalence between models defined at different levels. Two levels, called non-operational and operational, are addressed in this dissertation. The non-operational level describes these models in terms of program instructions or procedures, while the operational level describes them in terms of implementation events. Formal techniques are developed to prove the equivalence of rigorous specifications at both levels.

This formalism is then exploited to define the memory consistency models of two state-of-the-art multiprocess systems: the SPARC version 8 architecture and the Java Virtual Machine. These models are defined at both operational and non-operational levels. These operational and non-operational descriptions are proved equivalent. The SPARC models provide “reasonably” weak memory consistency models that are capable of avoiding the use of explicit synchronization primitives for certain problems. However, Java provides a consistency model that is completely dependent on synchronization primitives, without which no form of coordination between different threads is possible.

Fundamental process coordination problems have been extensively studied for traditional systems with strong memory consistency. This dissertation revisits the critical section and the producer/consumer problems in the context of weak memory consistency models. It establishes that the majority of known weak memory consistency models are incapable of supporting a solution to the critical section problem without the use of explicit synchronization primitives. Surprisingly, most of these models are capable of supporting solutions to versions of the producer/consumer problem without the use of these primitives.
This dissertation is divided into three parts, each of which occupies two consecutive chapters. The first (chapters 2 and 3) develops the formal framework and techniques required by the rest of the dissertation. The second (chapters 4 and 5) exploits this formalism to define memory consistency models of two state-of-the-art multiprocess systems. The third part (chapters 6 and 7) studies process coordination problems in the context of weak memory consistency systems, including those defined in chapters 4 and 5. Chapter 3 must be read before chapters 4 and 5. The material in Chapter 3 is never referred to in chapters 6 and 7. Chapters 6 and 7 can be read directly after Chapter 2 if the reader grasps the definitions posed in chapters 4 and 5 without delving deeper into their development details.

All the publications that resulted from or led to this work were authored jointly with Lisa Higham. A preliminary version of the framework of Chapter 2 appeared in the proceedings of the *10th International Conference on Parallel and Distributed Computing Systems* (October 1997) in a joint work with Nathaly Verwaal [41]. Some of the impossibility and possibility results of chapters 6 and 7 appeared in the proceedings of the *1997 International Symposium on Parallel Architectures, Algorithms, and Networks* (December 1997) [37]. An extended abstract based on the material of Chapter 5 and the impossibilities and possibilities pertaining to Java Consistency appeared in the proceedings of the *12th International Symposium on Distributed Computing* (September 1998) [38]. Finally, a summary of the results of chapters 6 and 7 was presented in the *13th International Symposium on Distributed Computing* (September 1999) [39] and in a poster session in the *1999 IBM Center for Advanced Studies Conference* (November 1999) [40].
Acknowledgments

My thanks go first and foremost to my supervisor, Lisa Higham. It was a great pleasure and privilege to work under her supervision. I am grateful to Lisa for posing the questions that initiated this work and for the arrangement of my financial support. She has always been available for discussion and advice. I also thank her for carefully reading the first drafts of this dissertation and for her comments that improved its presentation.

I would like to thank Brian Unger for suggesting the producer/consumer problem, and Alan Covington for suggesting the problem of Java Consistency. Markus Wittwer uncovered an error in an earlier proof by the Spin model checker, and his help with Spin saved me time and effort.

Most of the work done in this dissertation was supported in part by a post-graduate scholarship from the Natural Sciences and Engineering Research Council of Canada and by an Izaak Walton Killam Memorial Scholarship.
To the memory of Hosam,

a great brother and

an exceptional friend.
## Contents

Abstract iii  
Preface v  
Acknowledgments vii  
Dedication ix  
Contents xi  
List of Figures xv  
List of Abbreviations and Symbols xvii  

1 Introduction 1  
1.1 Motivation ........................................ 1  
1.2 Related Work ...................................... 4  
1.2.1 Defining New Models ............................ 4  
1.2.2 Developing Programming Methodologies ............. 7  
1.2.3 Formalizing Existing Implementations ............... 9  
1.2.4 Process Coordination ............................ 10  
1.2.5 Description Methods ........................... 10  
1.2.6 This Work ...................................... 11  
1.3 Organization and Overview .......................... 13  
1.4 Summary of Contributions .......................... 15  

2 Non-Operational Consistency Models 17  
2.1 The Framework ...................................... 17  
2.2 Pure Models ....................................... 22  
2.2.1 Sequential Consistency ........................... 23  
2.2.2 Linearizability ................................... 24  
2.2.3 Coherence ....................................... 25  
2.2.4 Pipelined–RAM ................................... 26  
2.2.5 Processor Consistency ............................ 27  
2.2.6 Causal Consistency ............................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>Hybrid Models</td>
<td>30</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Weak Ordering</td>
<td>30</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Other Hybrid Models</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Summary</td>
<td>32</td>
</tr>
<tr>
<td>2.5</td>
<td>Non-Terminating Computations</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>Operational Consistency Models</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Example Machine</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Machine Models</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Describing $MC$</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Relating Systems and Machines</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Example Proof</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Non-Terminating Systems</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Discussion</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>SPARC Consistency</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>Operational Description</td>
<td>51</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Object Types</td>
<td>53</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Operational Total Store Ordering</td>
<td>54</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Operational Partial Store Ordering</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Non-Operational Description</td>
<td>56</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Earlier Attempt</td>
<td>56</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Non-Operational Total Store Ordering</td>
<td>58</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Non-Operational Partial Store Ordering</td>
<td>66</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Earlier Complex Definitions</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Constructing SC from TSO and PSO</td>
<td>69</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparisons</td>
<td>70</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>75</td>
</tr>
<tr>
<td>5</td>
<td>Java Consistency</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Operational Description</td>
<td>77</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Operational Java with Base Operations</td>
<td>78</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Operational Java with Synchronization Operations</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Non-Operational Description</td>
<td>88</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Invisibility</td>
<td>88</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Non-Operational Java for Terminating Computations</td>
<td>88</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Non-Operational Java with Base Operations</td>
<td>90</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Non-Operational Java with Synchronization Operations</td>
<td>100</td>
</tr>
<tr>
<td>5.3</td>
<td>Discussion</td>
<td>105</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparisons</td>
<td>108</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Java versus Coherence</td>
<td>108</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Java versus Other Consistency Models</td>
<td>109</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>110</td>
</tr>
</tbody>
</table>
# 6 Critical Section Coordination

- **6.1 Automatic Verification** .......................... 113
- **6.2 Critical Section Problem** .......................... 114
  - 6.2.1 Problem Definition .................................. 114
  - 6.2.2 SC Solutions ........................................... 115
- **6.3 Impossibilities** ...................................... 116
  - 6.3.1 Template for Proofs .................................. 116
  - 6.3.2 Coherence .............................................. 117
  - 6.3.3 Causal Consistency .................................... 117
  - 6.3.4 P-RAM .................................................... 118
  - 6.3.5 Weak Ordering .......................................... 118
  - 6.3.6 SPARC Consistency .................................... 119
  - 6.3.7 Java Consistency ....................................... 120
- **6.4 PC-G Solutions** ....................................... 120
  - 6.4.1 Type of Variables ...................................... 121
  - 6.4.2 Number of Variables .................................... 122
  - 6.4.3 On the General Case .................................... 125
- **6.5 Dining Philosophers Problem** ....................... 131
- **6.6 Summary** .............................................. 132

# 7 Producer/Consumer Coordination

- **7.1 Producer/Consumer Problem** .......................... 133
  - 7.1.1 Problem Definition ..................................... 133
  - 7.1.2 SC Solutions ............................................ 135
  - 7.1.3 Process Self-Consistency .............................. 135
- **7.2 Impossibilities** ....................................... 136
  - 7.2.1 Template for Proofs ..................................... 136
  - 7.2.2 Coherence .............................................. 137
  - 7.2.3 Causal Consistency ..................................... 138
  - 7.2.4 P-RAM .................................................... 139
  - 7.2.5 Weak Ordering .......................................... 139
  - 7.2.6 SPARC Consistency .................................... 139
  - 7.2.7 Java Consistency ....................................... 140
  - 7.2.8 PC-G ..................................................... 141
- **7.3 Possibilities** .......................................... 142
  - 7.3.1 Single-Writer P1C1-queue Algorithm ................ 142
  - 7.3.2 Simpler Single-Writer P1C1-queue Algorithm .......... 145
  - 7.3.3 Multi-Writer P1C1-queue Algorithm .................. 149
  - 7.3.4 PmCn-set Algorithm .................................... 151
- **7.4 Summary** .............................................. 151
8 Conclusion .................................................. 153
8.1 Summary .................................................. 153
8.2 Conclusions ............................................. 155
8.3 Limitations ................................................ 157
8.4 Future Research .......................................... 158
8.5 Final Comment ............................................ 160

Bibliography .................................................. 161

A Partial and Total Orders ................................. 169
A.1 Basic Definitions ....................................... 169
A.2 Redundant Transitivity ................................ 169

B SPARC Ordering Constraints ............................ 171
B.1 Basic Definitions ....................................... 171
B.2 Total Store Ordering .................................... 172
B.3 Partial Store Ordering .................................. 172

C Java Ordering Constraints ............................... 175
C.1 Rules for One Thread ................................... 175
C.2 Rules for Thread-Main Memory Interaction ........ 176
C.3 Rules for Locks .......................................... 177
C.4 Rules for Volatiles ...................................... 178
C.5 Other Rules ............................................... 178

D CSP Algorithms ............................................ 181
D.1 Peterson’s Algorithm ................................... 181
D.2 Dekker’s Two-process Algorithm .................... 182
D.3 Dijkstra’s Algorithm .................................... 182
D.4 Knuth’s Algorithm ....................................... 183
D.5 De Bruijn’s Algorithm .................................. 183
D.6 Eisenberg and MacGuire’s Algorithm ................ 184

E SPIN Verification Suit ..................................... 185
## List of Figures

2.1 The multiprocess system \((P, J)\) ........................................... 18  
2.2 A possible timing for the \(\{p_1, p_2, p_3\}, \{S, V\}\) computation .................. 21  
2.3 Linearizable computation ............................................. 24  
2.4 SC but not linearizable computation ................................... 25  
2.5 Relationships between memory models ................................ 33  

3.1 \(M_C\), an example machine ........................................... 38  
3.2 Terms used in operational and non-operational models ................. 44  
3.3 Proving equivalence between operational and non-operational models ... 44  
3.4 Proving equivalence between operational and non-operational models for non-terminating systems (a) the provides-direction (b) the realizes-direction 48  

4.1 A two-processor SPARC architecture .................................. 52  
4.2 The SPARC atomic-load-store instruction ................................ 53  
4.3 Categorizing \(E\) of \(M_{TSO}\) and \(M_{PSO}\) ............................. 55  
4.4 Relationships between SPARC models and other models .................. 75  

5.1 A two-thread JVM architecture ......................................... 78  
5.2 JVM implementation of read and write invocations ...................... 79  
5.3 A simplified JVM architecture ......................................... 82  
5.4 Relationships between Java models and other models .................... 110  

6.1 Well known CSP algorithms for SC .................................... 115  
6.2 Burns’ CSP unfair solution ............................................. 129  
6.3 Summary of impossibilities (\(\times\)) .................................. 132  

7.1 \(A_{PC}^1\), a single-writer \(P_1C_1\)-queue algorithm ....................... 143  
7.2 \(A_{PC}^2\), a simpler single-writer \(P_1C_1\)-queue algorithm .................. 146  
7.3 \(A_{PC}^3\), a multi-writer \(P_1C_1\)-queue algorithm ....................... 149  
7.4 Summary of coordination possibilities (\(\checkmark\)) and impossibilities (\(\times\)) .... 152
# List of Abbreviations and Symbols

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Causal Consistency</td>
</tr>
<tr>
<td>CSP</td>
<td>Critical Section Problem</td>
</tr>
<tr>
<td>CSP(n)</td>
<td>Critical Section Problem with (n) processes</td>
</tr>
<tr>
<td>DPP</td>
<td>Dining Philosophers Problem</td>
</tr>
<tr>
<td>DPP(n)</td>
<td>Dining Philosophers Problem with (n) processes</td>
</tr>
<tr>
<td>Java_base</td>
<td>Java Consistency with base read/write variables only</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>PC-G</td>
<td>Processor Consistency - Goodman’s version</td>
</tr>
<tr>
<td>PCP</td>
<td>Producer/Consumer Problem</td>
</tr>
<tr>
<td>(P_mC_n)-queue</td>
<td>Producer/Consumer Queue Problem with (m) producers and (n) consumers</td>
</tr>
<tr>
<td>(P_mC_n)-set</td>
<td>Producer/Consumer Set Problem with (m) producers and (n) consumers</td>
</tr>
<tr>
<td>P-RAM</td>
<td>Pipelined-Random Access Machine</td>
</tr>
<tr>
<td>PSO</td>
<td>SPARC Partial Store Ordering</td>
</tr>
<tr>
<td>PSO_base</td>
<td>SPARC Partial Store Ordering with base read/write variables only</td>
</tr>
<tr>
<td>SC</td>
<td>Sequential Consistency</td>
</tr>
<tr>
<td>TSO</td>
<td>SPARC Total Store Ordering</td>
</tr>
<tr>
<td>TSO_base</td>
<td>SPARC Total Store Ordering with base read/write variables only</td>
</tr>
<tr>
<td>WO</td>
<td>Weak Ordering</td>
</tr>
<tr>
<td>WO_base</td>
<td>Weak Ordering with base read/write variables only</td>
</tr>
<tr>
<td>WOC</td>
<td>Coherent Weak Ordering</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>$P$</td>
<td>Set of processes</td>
</tr>
<tr>
<td>$J$</td>
<td>Set of (data) objects</td>
</tr>
<tr>
<td>$(P, J)$</td>
<td>Multiprocess system</td>
</tr>
<tr>
<td>$O$</td>
<td>Set of operations</td>
</tr>
<tr>
<td>$O</td>
<td>_x$</td>
</tr>
<tr>
<td>$O</td>
<td>_p$</td>
</tr>
<tr>
<td>$O</td>
<td>w$</td>
</tr>
<tr>
<td>$O</td>
<td>r$</td>
</tr>
<tr>
<td>$O</td>
<td>s$</td>
</tr>
<tr>
<td>$(O, \text{prog})$</td>
<td>Program Order</td>
</tr>
<tr>
<td>$(O, \text{time})$</td>
<td>Time Order</td>
</tr>
<tr>
<td>$(O, \text{wbr})$</td>
<td>Write-Before-Read Order</td>
</tr>
<tr>
<td>$(O, \text{causal})$</td>
<td>Causal Order</td>
</tr>
<tr>
<td>$(O, \text{wpo})$</td>
<td>Weak Program Order</td>
</tr>
<tr>
<td>$(O, \text{kpo})$</td>
<td>Kohli Partial Program Order</td>
</tr>
<tr>
<td>$(O, \text{tso})$</td>
<td>SPARC Total Store Ordering Partial Program Order</td>
</tr>
<tr>
<td>$(O, \text{pso})$</td>
<td>SPARC Partial Store Ordering Partial Program Order</td>
</tr>
<tr>
<td>$(O, \text{jpo})$</td>
<td>Java Partial Program Order</td>
</tr>
<tr>
<td>$(O, \text{bjpo})$</td>
<td>Base Java Partial Program Order</td>
</tr>
<tr>
<td>$w_p(x) v$ or $w(x) v$</td>
<td>a write operation of value $v$ to object $x$ by process $p$</td>
</tr>
<tr>
<td>$r_p(x) v$ or $r(x) v$</td>
<td>a read operation of value $v$ of object $x$ by process $p$</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>Set of processors</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>Space, set of (memory) locations</td>
</tr>
<tr>
<td>$N$</td>
<td>Set of operation invocations</td>
</tr>
<tr>
<td>$E$</td>
<td>Set of events</td>
</tr>
<tr>
<td>$I$</td>
<td>Implementation function ($I : N \times \Pi \to 2^E$)</td>
</tr>
<tr>
<td>$R$</td>
<td>Set of rules</td>
</tr>
<tr>
<td>$M = (\Pi, \Sigma, N, E, I, R)$</td>
<td>Multiprocess machine</td>
</tr>
<tr>
<td>$(E, \pi)$</td>
<td>Request Order</td>
</tr>
<tr>
<td>$(E, \gamma)$</td>
<td>System yield of $(P, J)$ on $M$</td>
</tr>
<tr>
<td>$M(P, J)$</td>
<td>Set of all system $(P, J)$ yields on $M$</td>
</tr>
<tr>
<td>$\Xi$</td>
<td>System $(P, J)$ run on $M$</td>
</tr>
<tr>
<td>$M\Xi(P, J)$</td>
<td>Set of all runs of $(P, J)$ on $M$</td>
</tr>
<tr>
<td>$C\Xi$</td>
<td>Induced computation of $\Xi$</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter introduces and motivates this dissertation. It also includes a literature survey.

1.1 Motivation

Weakening the memory consistency model enables improving the performance and scalability of a shared-memory multiprocess system. However, these models sacrifice programmability because they create complex models that provide fewer guarantees about the behavior of the memory system. Without the use of expensive, built-in synchronization, these models exhibit poor capabilities to support solutions for fundamental process coordination problems. This leads programmers to conservatively use these forms of synchronization, incurring additional performance burden on the system.

A multiprocess system can be viewed as a set of (distributed) processes sharing a set of (distributed) global data objects. These processes communicate with each other via this collection of shared space by performing operations on objects. A memory consistency model is a set of guarantees describing constraints on the outcome of a multiprocess program (i.e., sequences of interleaved and simultaneous operations).

Traditionally, those guarantees were expected to be many yielding what is called a strong memory consistency model. Lamport’s Sequential Consistency [49] is such a strong memory consistency model. Sequential Consistency requires that the result of any execution of the multiprocess system to be the same as if all the operations of all the processes were executed in a sequential order which is consistent with the program order of individual
processes. Although intuitive and easy to understand, strong memory consistency models, such as Sequential Consistency, are expensive to implement in large scale systems [56, 16] and in systems that do not utilize a central shared memory. In particular, strong memory consistency models are expensive to implement in distributed-shared memory (DSM) systems [63].

Weakening these guarantees admits more concurrency because it allows several optimizations, such as relaxing the blocking and atomicity requirements of strong memory consistency, write buffering, and efficient data replication [2, 27, 25, 5, 56, 11]. We refer to memory consistency models with fewer guarantees than Sequential Consistency by weak memory consistency models.\(^1\)

With enhancing performance and scalability as a target, many weak consistency models have been proposed, implemented, or both [56, 22, 5, 34, 9, 15, 10]. Moreover, many of the commercially available systems today adopt weak memory consistency models [2] such as the SPARC architecture [67, 71] and the Java programming language [35, 55].

The performance gain resulting from weakening memory consistency models had to pay the programmability price. Indeed, the resulting weak memory consistency models can be too difficult to understand and program. Moreover, these memory consistency models arise from a wide variety of sources including architecture, system, and database designers, application programmers, and theoreticians. The descriptions of memory behavior use different types and degrees of formalism. Definitions range from precise and complicated axiomatic specifications to informal and sometimes ambiguous natural language descriptions. This further complicates understanding, reasoning about, or comparing different memory consistency models. Programming for these models becomes inefficient when a new descriptive style must first be mastered for each change of model. Thus, we are motivated to provide a common formal framework that unifies all these models in one descriptive style that is easily understood by programmers.

Since synchronization primitives are a burden on performance, we are also motivated

\(^1\)We also use weak models, weak consistency models, weak memory consistency, or weak consistency.
to study the limitations and capabilities of weak memory consistency models with regard to avoiding the use of these primitives. So, we are motivated to find out which weak memory consistency models have the ability to support solutions to fundamental process coordination problems without the use of explicit synchronization. If the use of explicit synchronization is avoidable, then efficient libraries for certain classes of applications can be built, which not only ease the job of distributed application programmers, but make their applications more efficient as well. Moreover, investigations should be launched to minimize the use of synchronization where avoiding it is not possible, which would also lead to building efficient libraries for programmers. This work is a step towards bridging the gap between efficiency and scalability on one side and programmability on the other.

There exists a wide variety of memory consistency models; however, it has never been clear if there is a “best” model. Strong models are easier to understand but are inefficient to implement, while weak models are efficient to implement but complicated to program. Mark Hill, a pioneer in the area of weak memory models, has recently “reverted” to Sequential Consistency publishing a surprising article titled: “Multiprocessors Should Support Simple Memory Consistency Models” [43]. Mark Hill relied on the “lack of quantitative data on the benefits of [weak] models for compiler optimizations, [the] absence of widely used programming standards for shared-memory, and the requirement on vendors to keep their systems backward compatible.”[8]. Some recent experimental studies show that optimizing Instruction-Level Parallel processors with hardware prefetching, write buffering and speculative loads could narrow the performance gap between Sequential Consistency and weak models [60].

All these memory models are basic structures. This dissertation analyses these structures by studying their capabilities to support basic process coordination patterns, which are also in turn basic building blocks for larger programs. To accomplish this task, we define a comprehensive formal framework, which we use to derive precise definitions for the memory consistency models supported by the SPARC architecture and the Java programming language. We also redefine in this framework several widely quoted memory models
from the literature. Finally, the capabilities are studied in terms of two fundamental process coordination problems: critical section coordination and producer/consumer coordination. The majority of known weak memory consistency models are incapable of supporting a solution to the critical section problem without the use of explicit synchronization primitives. Surprisingly, most of these models are capable of supporting solutions to certain versions of the producer/consumer problem without the use of these primitives.

1.2 Related Work

Previous work related to this dissertation falls under four categories: defining new memory consistency models, formalizing the memory consistency models of existing implementations, developing programming methodologies for weak memory consistency models, and proving coordination impossibilities and possibilities for weak memory consistency models. These categories are not independent. Most papers in the literature target at least two of those categories. However, this separation is used here for clarity of presentation.

1.2.1 Defining New Models

In this section, memory consistency models are classified into two categories: strong models and weak models.

**Strong models**

*Sequential Consistency* and *Linearizability* are known to be the “traditional” consistency requirements of shared memory. Shared memories were expected to behave as such. These two models constitute what are called *strong* memory consistency models. Sequential Consistency [49], which is due to Leslie Lamport, is an extension of the uniprocessing environment to the multiprocessing one. Later Lamport defined *Atomic Memory* [51]. The memory consistency model arising from atomic memory is stronger than Sequential Consistency. This model was called Linearizability by Herlihy and Wing [36]. (Mosberger
1.2. RELATED WORK

calls it Dynamic Atomic Consistency [59].) Herlihy and Wing also studied the differences between the two models.

Attiya and Welch carried on studying those differences [16]. They compared Sequential Consistency and Linearizability in different time settings and in the context of three types of objects: read/write objects, FIFO queues, and stacks. The time settings they used are approximately synchronized and perfectly synchronized processes. In the latter case, they argue that Sequential Consistency and Linearizability cannot be distinguished. However, it is more efficient to implement Sequential Consistency than Linearizability when clocks are imperfect or absent.

Although Sequential Consistency is more cost-effective than Linearizability, it remains in general very costly to implement. This has been stated explicitly by Lipton and Sandberg [56] for the first time. Attiya and Welch’s results also show that strong consistency conditions pay the price of high latency in systems that have high message-passing delays.

Very recently, Torres-Rojas, Ahamad, and Raynal [68] defined Timed Consistency, which unifies Sequential Consistency and Linearizability. Timed Consistency requires that if a write operation is performed at some point in time $t$, it must be seen by the rest of the system components at no later than $t + \Delta t$. If $\Delta t = 0$, then Timed Consistency is Linearizability; if it is infinity, then Timed Consistency is Sequential Consistency. Timed Consistency captures all variants of consistency that fall between Linearizability and Sequential Consistency.

Weak models

Since Sequential Consistency (and any stronger model) is expensive to implement, researchers launched investigations to weaken Sequential Consistency.

Dubois, Scheurich, and Briggs [22] were the first to propose a weak memory consistency model called Weak Ordering. Weak Ordering classifies operations (alternatively, data objects) as either ordinary or synchronization. Synchronization operations are guaranteed to be strongly consistent (normally, Sequentially Consistent), while there are fewer guar-
CHAPTER 1. INTRODUCTION

antee for the orderings of ordinary operations.

Many other weak models that do not distinguish between operations have been proposed. Lipton and Sandberg proposed the Pipelined-Random Access Machine [56], a DSM system. James Goodman [34] informally distinguished between three consistency levels: Sequential Consistency (what he called strong consistency), Processor Consistency (what he called processor ordering), and Coherence (what he called cache consistency).

Goodman proposed Processor Consistency as a guideline for or a class of memory consistency models [34]. Although Goodman’s intension was never to define a particular memory consistency model, many of the proposed Processor Consistency models share Goodman’s original intentions but differ in substantial ways. Verwaal [69] distinguished between six Processor Consistency versions that were called: PC-G, PC-Gharachorloo, PC-DASH, PC-Vax, PC-Ahamad, and PC-Kohli. In this dissertation, we limit our discussion to PC-G, which was defined by Ahamad et al.[9]. PC-VAX is a formalization of the VAX 8800 memory model. All of PC-Gharachorloo, PC-DASH , PC-Ahamad, and PC-Kohli are different interpretations of the memory model supported by the Stanford DASH multiprocessor [54]. Verwaal showed that all of these versions of Processor Consistency differ from each other. Others such as Attiya and Friedman [15] did not attribute the Coherence property to Processor Consistency.

Causal Consistency, another weak model that is based on causality relations, was proposed by Ahamad et al.[10]. Causal Consistency is weaker than Sequential Consistency but stronger than Pipelined-RAM.

Following Dubois, Scheurich, and Briggs [22], Gibbons, Merritt, and Gharachorloo [30, 31] used the same idea of distinguishing operation types to define Release Consistency. This model was meant to formalize the memory consistency model of the DASH multiprocessor. While Weak Ordering classifies operations as either ordinary or synchronization, Release Consistency further classifies synchronization operations into acquire and release operations. When all acquires and releases are guaranteed to behave in a Sequentially Consistent manner, the resulting model is denoted $RC_{SC}$. When these are guaranteed
1.2. RELATED WORK

to behave according to Processor Consistency, $\text{RC}_{\text{PC}}$ denotes the resulting model.

Using similar ideas, Attiya and Friedman defined Hybrid Consistency [15]. Attiya and Friedman argue that Hybrid Consistency restricts the way a memory consistency system, the layer that implements virtual shared memory in a DSM system, should appear to the programmer, while Weak Ordering restricts the hardware implementation. Attiya and Friedman claim that their approach admits more optimizations.

1.2.2 Developing Programming Methodologies

While the new defined weak models promise more efficient implementations, they complicate programming. Understanding and using these models is not an easy job. To circumvent this loss, researchers started developing programming strategies that aim at increasing the programmability of the resulting complex systems.

The notion of data race free programs was pioneered by Adve and Hill [5, 7, 1]. A data race occurs when two ordinary operations, at least one of which is a write, access the same variable and could execute concurrently. To force a program to be data race free, the races among ordinary operations are eliminated by forcing an order among them with synchronization operations. Given that programmers prefer to reason about Sequential Consistency, Adve and Hill derived sufficient conditions for hardware to appear Sequentially Consistent to the programmer when it executes a data race free program [6].

A complementary approach was taken by Gibbons, Merritt, and Gharachorloo [30, 31, 25]. Instead of specifying system requirements to support a class of programs, they specify how a program should be in order to appear Sequentially Consistent on a certain memory consistency model. In this framework, they developed the notion of Properly Labeled (PL) programs [29, 28, 31].

The notion of PL programs realizes that memory operations generated by a given program have different roles. Some operations are meant to synchronize other operations that access data variables. So, operations are classified as either synchronization (called competing) or ordinary (called non-competing). Competing operations should be ordered with
stronger guarantees than non-competing. For instance, operations in the entry section of a mutual exclusion algorithm should be classified as competing to guarantee mutual exclusive access to the critical section. On the other hand, operations in a critical section are non-competing.

More precisely, two operations conflict if both are to the same variable and at least one of them writes that variable. These conflicting operations are competing if they could execute concurrently. For instance, a read in a synchronization loop which causes the loop to break is regarded as competing. Also, the write that causes the read to break the loop is competing. Most of the time, programmers (or compilers) can tell which operations should be competing and which should not be from the code of their programs.

When operations in a program are labeled according to their categories, the program is called PL. The underlying architecture, which implements a memory model weaker than Sequential Consistency, utilizes this additional information in order to guarantee that the outcome of a PL program is the same as that for Sequential Consistency. This eases programming for memory models that are weaker than Sequential Consistency because programmers need only reason about SC and provide proper labeling from the semantics of their programs.

As a result, two programming models PL\textsubscript{sc} and PL\textsubscript{pc} were proposed to correspond to the RC\textsubscript{sc} and RC\textsubscript{pc} memory consistency models. The similarities between data race freedom and proper labeling led them to join their efforts [26, 3].

Attiya et al.[12, 23] followed the steps of Gibbons, Merritt, and Gharachorloo in developing programming strategies for Hybrid Consistency and other models [14]. In addition to data race free programs, the programming strategies they addressed for Hybrid Consistency constitute the transformation of programs that are correct for Sequential Consistency to ones that are correct for Hybrid Consistency. This transformation consists of labeling operations in the exit and entry sections of a mutual exclusion algorithm as strong (analogous to synchronization in Weak Ordering). Everything in the critical section is labeled as weak (analogous to ordinary in Weak Ordering).
Ahamad et al. also showed that data race free programs run correctly (as if Sequentially Consistent) on Causal Consistency [10].

Leslie Lamport [52] proposed a method to transform a program that is correct for Sequential Consistency to one that is correct under any weaker memory consistency model. The method derives sufficient synchronization primitives from the proof of correctness of the program at hand. The idea is to enforce the assumptions used by the proof by augmenting the program with appropriate synchronization primitives.

1.2.3 Formalizing Existing Implementations

The description of a memory consistency model of an existing system is given either in terms of hardware implementation or in mathematical formalism that is not always useful to programmers. For this reason, several researchers attempted to formalize these models to increase their accessibility to programmers.

The Stanford DASH multiprocessor implements Release Consistency with (one version of) Processor Consistency [54]. Gharachorloo et al. [29] derived the DASH version of Processor Consistency, which later turned out to be stronger than the actual implementation [28]. Attiya and Friedman formalized Alpha Consistency, the DEC-Alpha multiprocessor memory model [14].

Sindhu, Frailong, and Cekleov [66] have given an axiomatic description of the memory models supported by the SPARC version 8 multiprocessors [67]. Kohli et al. [47] attempted a definition for one version of SPARC which turned out to be stronger than the actual implementation as discussed later in Chapter 4.

The memory consistency model of the PowerPC shared memory architecture was derived by Corella, Stone, and Barton [19].

Adve and Hill [6] defined Data-Race-Free-1 which formalizes the VAX memory model and unifies it with other models, such as Release Consistency and Weak Ordering.

A non-operational definition for a special case of Java Consistency was given by Gontmakher, Itskovitz, and Schuster [32, 33].
1.2.4 Process Coordination

When defining the Pipelined-RAM memory consistency model, Lipton and Sandberg [56] presented a centralized mutual exclusion algorithm that is correct for Pipelined-RAM. Since the algorithm is centralized it requires cooperation. Therefore, coordination with cooperation is possible in Pipelined-RAM. However, Attiya and Friedman [13, 23] showed that any Pipelined-RAM solution to mutual exclusion necessarily requires cooperation. Their result applies also to Causal Consistency.

In a careful study of the power of Processor Consistency, Ahamad et al.[9] addressed process coordination in the context of two versions of Processor Consistency: the DASH version and Goodman’s version. They showed that Peterson’s mutual exclusion algorithm [62] is correct for both versions, but Lamport’s Bakery algorithm [48] is not (for both versions as well).

1.2.5 Description Methods

The very first weak memory consistency models were described informally (Weak Ordering, Pipelined-RAM, Processor Consistency, and Coherence). Researchers realized the complexity arising from these models necessitating the formalization of these definitions.

One remarkable formalism is I/O automata. An I/O automaton [58] is an automaton (with possibly an infinite set of states) such that the transitions denote actions. I/O automata support three types of actions: input, output, and internal. Input actions are controlled by the external environment, while output and internal actions are generated by the automaton itself. I/O automata can be composed by wiring the output transitions of one set of automata to the input transitions of another, such that these input and output transitions have the same label. A wired input and output action becomes an internal action of the resulting automaton. This model has been used to specify algorithms and prove their correctness [57].

---

2 Informally, a mutual exclusion algorithm is cooperative if a process attempting to enter the critical section is required to communicate with every other process, even if they are executing in the remainder section.
Gibbons and Merritt [30] used a framework based on I/O automata. They start with specifying a base automaton that represents basic architectural assumptions. Then, a memory consistency model is defined by further restricting the actions of the base automaton. They have used this formalism to define Release Consistency and to prove that PL programs behave as Sequentially Consistent ones on Release Consistent systems.

Sindhu, Frailong, and Cekleov [66] used an axiomatic approach to specify the SPARC version 8 memory models. Their framework is based on three sets: memory operations, partial orders defined on memory operations, and axioms which are concerned with the legality of orders.

Ahamad et al.[9, 10, 47] proposed a framework that is based on partial orders on execution histories. The framework was used to define Causal Consistency as well as to “clean” different widely cited memory consistency model definitions, such as Processor Consistency and Pipelined-RAM.

Based on Collier’s formalism [18], Adve and Hill developed a formalism called Sequential Consistency Normal Form (SCNF) [6], which specifies a consistency model in terms of constraints on programs that makes the hardware appear Sequentially Consistent to those constrained programs.

The formalism developed by Attiya et al.[12, 15, 14] is also based on partial orders and was used to define Hybrid Consistency and Alpha Consistency. It was also used to develop and show correct the programming techniques for these models.

Yet another approach is that of executable specifications for analyzing and verifying memory models. The formal verification laboratory at Stanford developed Murϕ as a description language and a verification system for finite state concurrent systems [21]. Murϕ was used to verify the SPARC v9 Relaxed Memory Order [61].

1.2.6 This Work

All the above mentioned research benefited this dissertation. We believe that much of the understanding of the topic of memory consistency models is still lacking, although steadily
increasing. Every single piece of work in the area adds to our understanding of the topic. Thus, this dissertation.

It is not even clear if it is a legitimate question whether a given memory model is ultimate. Every model has its own taste with its own weakness and strength.

The formal framework presented in chapters 2 and 3 is more extensive; it incorporates mechanisms for defining relations between implementations and specifications. It states formally and explicitly how a memory consistency model, which is normally defined for terminating computations, extends in the non-terminating setting. Ignoring the non-termination issue undermines the conclusions made regarding a particular system, as is confirmed with Java Consistency (Chapter 5). The framework is also more general because it does not make any assumptions about the underlying data objects.

This careful formalism allowed us to derive non-operational definitions for two existing multiprocess systems: SPARC Consistency and Java Consistency. Unlike the axiomatic approach in [66, 67], we take a more natural approach for defining SPARC Consistency. Similar to the approach of Ahamad et al.[9, 10, 47], it describes a memory consistency model from the point of view of processes, enhancing programmability. The mechanism in our framework for associating memory models with actual implementations allows us not only to avoid errors, but to pinpoint those in other attempts as well.

Gontmakher, Itskovitz, and Schuster’s definition of Java Consistency [32, 33] assumed that a write to a thread’s local memory will eventually be written back to the main memory, an assumption that fairly simplified the resulting definition. Our definition that is addressed in Chapter 5 was given independently [38] and captures the more general case. Our extensive framework allowed us to highlight the subtleties arising from Java Consistency as shall be seen in Chapter 5.

In this dissertation, we look at fundamental process coordination patterns, which are building blocks for larger programs. The impossibility results presented in chapters 6 and 7 seem extremely simple. They are so because the extensive formalism developed in this dissertation lays down the appropriate foundation for addressing these problems.
1.3 Organization and Overview

Chapter 2
The formalism for defining memory consistency models in terms of high-level program operations, called non-operational, is presented in this chapter. A multiprocess system is mathematically defined. A memory consistency model is defined to be a set of constraints on a multiprocess system computation. The definitions of widely cited memory consistency models are re-stated using our framework. These are Coherence, Pipelined-Random Access Machine, Goodman’s Processor Consistency, Causal Consistency, and Weak Ordering. They are also compared against each other. The definitions of these memory models are given assuming the multiprocess system terminates. Extending those to non-terminating systems is discussed and formalized.

Chapter 3
This chapter extends the formalism of Chapter 2 by providing a mechanism for associating specifications with implementations. Specification models are called non-operational; implementation models are called operational. The chapter gives the basic mathematical tools needed to specify applications, whether they are software layers or hardware architectures. While processes perform operations, an actual system implements these operations by proceeding with sequences of lower-level operations, called events. It also develops the tools for proving that a certain implementation gives rise to a certain memory consistency model, and vice versa. Non-terminating systems are also discussed in this chapter.

Chapter 4
The SPARC version 8 architecture implements two memory consistency models: Total Store Ordering and Partial Store Ordering. We describe the SPARC architecture implementation for both models. SPARC “suffers” from what we call transient invisibility, where some write operations temporarily cannot be seen by some processors. Invisibility issues affect the resulting memory consistency models and must be dealt with when defining the memory model. Then, we derive the associated memory consistency definitions and prove their correctness. After establishing this, these models are compared with the rest of the
models quoted in Chapter 2.

Chapter 5
This chapter describes the Java Virtual Machine in simpler terms than those used in the Java manuals. The Java Virtual Machine also suffers from two kinds of invisibility: fixate and covert. These are more serious than transient invisibility. An explicit treatment for those is necessary to derive a correct Java Consistency model, one that describes the exact behavior of the Java Virtual Machine. The Java Consistency model is then derived and proved to capture the implementation. A discussion follows that targets a Java model that does not explicitly deal with covert invisibility. Wrong conclusions may be derived when considering non-terminating systems. In particular, this restricted Java model might give the impression that Java is Coherent, a conclusion that does not hold when dealing with non-terminating systems. Finally, Java is compared with the SPARC models as well as the models of Chapter 2.

Chapter 6
All the models mentioned in this dissertation, with the exception of Goodman’s Processor Consistency, are proved to be incapable of supporting a read/write solution to the critical section problem. Even unfair solutions are impossible. For Goodman’s Processor Consistency, we derive a tight bound for the 2-process case on the number of read/write variables needed to solve the critical section problem. It has been shown that Peterson’s algorithm is a correct solution for the critical section problem in that model. We show that Burns’ mutual exclusion algorithm is an unfair solution for the problem in Goodman’s Processor Consistency. Burns’ algorithm uses $n$ single-writer variables and one multi-writer only, while Peterson’s uses $n$ single-writers and $n - 1$ multi-writers ($n$ is the number of processes). We also specify requirements that an $n$-process solution must satisfy in order to be correct for Goodman’s Processor Consistency. These specifications are used to show that Eisenberg and MacGuire’s, De Bruijn’s, Dekker’s, Dijkstra’s, and Knuth’s algorithms are incorrect for Goodman’s Processor Consistency. We finally show that the dining philosophers problem is also impossible to solve where the critical section problem cannot be solved.
1.4 Summary of Contributions

Chapter 7

In some consistency models, some versions of the producer/consumer problem can be solved with just read/write operations even though no read/write solution for the critical section problem exists in these models. This suggests that producer/consumer coordination is a “weaker” problem than critical section coordination. Traditional solutions for the producer/consumer problem made use of the ability to solve the critical section problem. In this chapter, we distinguish between two major variants of the producer/consumer problem: the queue and set problems. In the queue version, consumers must consume items in the order in which they were produced, while in the set version order is insignificant. We show that without explicit synchronization, Java cannot support a solution to any version of the problem, even when only two processes are involved. However, we find that all the other models that cannot support a solution to the critical section problem can support a solution to the general (any number of processes) producer/consumer set problem, without the use of explicit synchronization. Furthermore, all of these support a solution to the queue problem for one consumer and one producer only. However, they become incapable of supporting a solution to the queue problem when the total number of processes exceeds two. Three different single-producer and single-consumer algorithms are presented and proved correct.

Chapter 8

This chapter summarizes and concludes the dissertation. The discussion also includes future research directions.

1.4 Summary of Contributions

The contributions of this dissertation can be summarized as follows:

- Developed a comprehensive framework that:
  
  - provides a formalism for precisely defining memory consistency models,
  
  - states mathematically how a memory consistency model must be interpreted in
the context of non-terminating systems,

– provides a formalism for describing implementations, such as multiprocessor architectures and DSM protocols, which is tailored to serve memory consistency purposes, and

– provides mechanisms for proving equivalence between a non-operational memory consistency definition and an operational memory consistency model arising from a particular implementation.

- Derived and proved correct non-operational definitions of the Total Store Ordering and Partial Store Ordering memory consistency models implemented by the SPARC version 8 architecture. The definitions are described from the point of view of processes and are, thus, a natural extension of Sequential Consistency.

- Derived and proved correct a non-operational definition for Java Consistency that does not make any assumptions that restricts the behavior of the Java Virtual Machine. This definition is also described from the point of view of processes.

- Proved that Java Consistency is incapable of achieving any form of process coordination without the use of volatile variables or synchronized constructs. This shows that Java is an intrinsically synchronization-dependent programming language.

- Proved that neither of the SPARC consistency models is capable of supporting a solution to the critical section problem without the use of expensive synchronization constructs, i.e. the atomic-load-store and store-barrier instructions.

- Developed and proved correct specialized solutions to the producer/consumer problem that use only read and write operations for a very restricted version of the queue problem, and for the general case of the set problem in both SPARC consistency models as well as other consistency models.
CHAPTER 2

Non-Operational Consistency Models

This chapter introduces the basic formal framework of the dissertation. The definitions of widely cited memory consistency models are re-stated using this framework. Relationships between these models are also summarized. The definitions of these memory consistency models assume that a given system terminates. Later in the chapter, extension to the non-terminating setting is formalized.

2.1 The Framework

Our aim is to provide a framework that is capable of describing the exact behavior of the memory of any shared-memory system of processes as it appears to the programmer. Such a memory could be centralized or distributed-shared. Furthermore, systems could utilize optimizations such as instruction buffering, caching, reordering, and overlapping. The formalism described here allows us to abstractly specify the memory behavior. This abstract specification is called non-operational because it hides the implementation (operational) details and describes the memory behavior in terms of ordering constraints on the high-level program operations that are used by the programmer.

We model a multiprocess system as a collection of processes operating on a collection of shared data objects, as is shown in Figure 2.1.

One way to define a data object is by describing the object’s initial state, the operations that can be applied to the object and the change of state that results from each applicable operation. This would give rise to a set of allowable sequences of operations for each
such object. So, we choose to define a data object by providing the set of all sequences of allowable operations together with their results (similar to that of Herlihy and Wing [36].) Specifically, an operation is a 4-tuple \((ac, obj, in, out)\) where “\(ac\)” is an action, “\(obj\)” is an object name, and “\(in\)” and “\(out\)” are sequences of parameters. The operation \((ac, obj, in, out)\) means that the action “\(ac\)” with input parameters “\(in\)” is applied to the object “\(obj\)” yielding the output parameters “\(out\)”.

A (sequential data) object is a specification of a set of sequences of operations. An arbitrary sequence of operations is valid for object \(x\) if and only if it is in the specification of \(x\). For example, we specify a shared (base) read/write variable, \(x\), by the set of all sequences \(\langle o_1, o_2, \cdots \rangle\) such that

1. each \(o_i\) is either a read operation, denoted by a four-tuple \((\text{read}, x, \lambda, (v))\)\(^1\), that returns the value \(v\) of variable \(x\), or a write operation, denoted \((\text{write}, x, (v), \lambda)\), that assigns a value \(v\) to variable \(x\), and

2. for every read operation, the value returned is the same as the value written by the most recent preceding write operation in the sequence.

Thus, any sequence of operations that satisfies 1) and 2) is valid for the read/write variable \(x\).

An operation \(o = (ac, obj, in, out)\) can be decomposed into the two matching compo-

\(^1\)\(\lambda\) denotes the empty sequence.
nents, \((ac, \text{obj, in})\), called the \textit{operation-invocation} and denoted \(\text{invoc}(o)\), and \((ac, \text{obj, out})\), called the matching \textit{operation-response} and denoted \(\text{resp}(o)\). Let \(\langle e_1, e_2, \cdots \rangle\) be a sequence consisting of operation-invocations and operation-responses. Then \(e_j\) \textit{follows} \(e_i\) if and only if \(i < j\) and \(e_j\) \textit{immediately follows} \(e_i\) if and only if \(i = j - 1\).

A \textit{timed-operation} is a 6-tuple \((ac, \text{obj, in, invoc-time, out, resp-time})\) obtained by augmenting an operation with the process’s local time for the invocation and the response of the operation. For a timed-operation \(o\), \(\text{time}(\text{invoc}(o))\) denotes the invoc-time of the timed-operation. Similarly, \(\text{time}(\text{resp}(o))\) denotes the resp-time of the timed-operation.

Informally, a process interacts with data objects by issuing a stream of invocations to some subset of them and receiving a stream of responses that are interleaved with its invocations, as shown in Figure 2.1. This is formalized as follows. A \textit{process} is a (possibly infinite) sequence of operation-invocations. A \textit{process execution} is a sequence of operation-invocations and operation-responses such that each response follows its matching invocation. A process execution is \textit{blocking} if, for each operation that has a non-empty output, the response of that operation immediately follows its matching invocation.

Whether blocking or non-blocking, the natural notion of the computation of a process is the sequence of operations that arises as its invocations are processed. Therefore, a \textit{process computation} is the sequence of operations created from a process execution by augmenting each invocation in the process sequence with its matching response. Note that operations in a process computation maintain the order of their invocation components in the process execution.

A \textit{(multiprocess) system}, \((P, J)\), is a collection \(P\) of processes and a collection \(J\) of objects, such that each operation-invocation of each process in \(P\) is applied to an object in \(J\). A \textit{(multiprocess) system execution} for a system \((P, J)\), is a collection of process executions, one for each process in \(P\). Similarly, a \textit{(multiprocess) system computation} is a collection of process computations, one for each \(p\) in \(P\).
Example

Consider the system \((P,J)\) where \(J = \{S,V\}\). The objects \(S\) and \(V\) respectively denote a stack which is initially empty and a read/write variable initialized to 0. Object \(V\) is as specified before. Object \(S\) is specified\(^2\) by the set of all sequences \(\langle o_1, o_2, \cdots \rangle\) such that

1. each \(o_i\) is either a pop operation, denoted by a four-tuple \((\text{pop},S,(v),\lambda)\), or a push operation, denoted \((\text{push},S,(v),\lambda)\), and
2. each push operation \((\text{push},S,(v),\lambda)\) appends \(v\) to the end of string \(\text{str}(S)\) which is initially set to \(\lambda\), and each pop operation \((\text{pop},S,(v),\lambda)\) returns \(v\) the last symbol in \(\text{str}(S)\) and deletes this symbol from \(\text{str}(S)\).

The set of processes \(P = \{p_1, p_2, p_3\}\) such that each process is given by the following sequences of operation invocations:

\[
\begin{align*}
p_1 &: (\text{read}, V, \lambda) (\text{push}, S, (1)) (\text{push}, S, (2)) \\
p_2 &: (\text{pop}, S, \lambda) (\text{write}, V, (2)) (\text{push}, S, (3)) \\
p_3 &: (\text{read}, V, \lambda) (\text{pop}, S, \lambda) (\text{read}, V, \lambda) (\text{pop}, S, \lambda) (\text{write}, V, (3))
\end{align*}
\]

One possible \((P,J)\) system execution is as follows, where each operation response (underlined) is added following its matching operation invocation:

\[
\begin{align*}
p_1 &: (\text{read}, V, \lambda) (\text{push}, S, (1)) (\text{read}, V, (0)) (\text{push}, S, \lambda) (\text{push}, S, (2)) (\text{push}, S, \lambda) \\
p_2 &: (\text{pop}, S, \lambda) (\text{pop}, S, (1)) (\text{write}, V, (2)) (\text{push}, S, (3)) (\text{push}, S, \lambda) (\text{write}, V, \lambda) \\
p_3 &: (\text{read}, V, \lambda) (\text{read}, V, (0)) (\text{pop}, S, \lambda) (\text{pop}, S, (2)) (\text{read}, V, \lambda) (\text{read}, V, (2)) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{push}, S, \lambda) (\text{write}, V, (3)) (\text{write}, V, \lambda) (\text{pop}, S, (3))
\end{align*}
\]

Finally the system computation corresponding to the above system execution is as follows:

\[
\begin{align*}
p_1 &: (\text{read}, V, \lambda, (0)) (\text{push}, S, (1), \lambda) (\text{push}, S, (2), \lambda) \\
p_2 &: (\text{pop}, S, \lambda, (0)) (\text{write}, V, (2), \lambda) (\text{push}, S, (3), \lambda) \\
p_3 &: (\text{read}, V, \lambda, (1)) (\text{pop}, S, \lambda, (2)) (\text{read}, V, \lambda, (2)) (\text{pop}, S, \lambda, (3)) (\text{write}, V, (3), \lambda)
\end{align*}
\]

Note that the above process executions are non-blocking and that two operation responses need not follow the order of their matching operation invocations. Figure 2.2 shows a possible timing for the system computation.

\(^2\)Other specifications are also possible. For instance, one may want to disallow performing a pop on an empty stack. However, the specification given here serves the purpose of the example.
Let \((P,J)\) be a multiprocess system, and \(O\) be all the operations in a computation of this system. \(O|p\) denotes all the operations that are in the process computation of \(p\) in \(P\). \(O|x\) denotes all the operations that are applied to object \(x\) in \(J\).

We define two (strict) partial orders\(^3\) on the operations of a system. More partial orders will be defined as needed.

**Program Order**

Define the *program order*, denoted \((O, \stackrel{\text{prog}}{\rightarrow})\), by \(o_1 \stackrel{\text{prog}}{\rightarrow} o_2\), if and only if \(\text{invoc}(o_2)\) follows \(\text{invoc}(o_1)\) in the definition of \(p\) (equivalently, \(o_2\) follows \(o_1\) in the computation of \(p\)).

The order \((O, \stackrel{\text{prog}}{\rightarrow})\) is a total order on \(O|p\). Despite this, processes need not strictly obey the “program text order”. In systems where instruction reordering and overlapping is utilized, it is always guaranteed that the outcome of the given program is the same as when the program text order is followed.

---

\(^3\)A strict partial order (or simply a partial order) is defined in Appendix A.
Time Order

Define the time order, denoted \( (O, \text{time}) \), by \( o_1 \text{time} o_2 \) if and only if \( o_1 \) and \( o_2 \) are timed-operations and \( \text{time(invoc}(o_2)) > \text{time(resp}(o_1)) \).

For the definition of some memory consistency models it is necessary to distinguish the operations that change a shared object from those that only inspect a shared object. Let \( O|w \) denote the subset of \( O \) consisting of those operations in \( O \) that update (write) a shared object, and \( O|r \) denote the subset consisting of the operations that only inspect (read) a shared object. There are also some consistency models that provide other classes of operations. These are defined as needed.

Given any collection of operations \( O \) on a set of objects \( J \), a linearization of \( O \) is a (strict) linear order\(^4\) \( (O, \xrightarrow{L}) \) such that for each object \( x \) in \( J \), the subsequence \( (O|x, \xrightarrow{L}) \) of \( (O, \xrightarrow{L}) \) is valid for \( x \).

A (memory) consistency model is a set of constraints on system computations. A finite system computation of \( (P, J) \) satisfies consistency model \( D \) if the computation meets all the constraints in \( D \). For any system \( (P, J) \) with only finite processes, \( (P, J) \) delivers memory consistency \( D \) if every computation that can arise from it satisfies \( D \). Extension of memory consistency models to non-terminating computations is defined later in the chapter.

Let \( D_1 \) and \( D_2 \) be two memory consistency models. Model \( D_1 \) is stronger than \( D_2 \) if the constraints of \( D_1 \) imply the constraints of \( D_2 \). We also say \( D_2 \) is weaker than \( D_1 \). Model \( D_1 \) is strictly stronger than \( D_2 \) if \( D_1 \) is stronger than \( D_2 \) but \( D_2 \) is not stronger than \( D_1 \). Models \( D_1 \) and \( D_2 \) are equivalent if \( D_1 \) is stronger than \( D_2 \) and \( D_2 \) is stronger than \( D_1 \). Finally, \( D_1 \) and \( D_2 \) are incomparable if \( D_1 \) is neither stronger nor weaker than \( D_2 \).

### 2.2 Pure Models

The literature describes many consistency conditions that may be considered “natural”. The strongest, Linearizability, is usually assumed by theoreticians designing distributed

---

\(^4\)A strict linear order (or simply a linear order) is defined in Appendix A
algorithms. One of the weakest, Coherence, is typically assumed to be a necessary requirement of any reasonable system. Numerous others fall between these two extremes and are often incomparable. We describe the most commonly used memory models in the following sections. These memory consistency models are defined for finite or terminating computations. Extending these definitions to the non-terminating setting is given in Section 2.5.

Pure models treat all objects “equally”. For this reason, we limit our discussion to read/write variables. However, the definitions given next could extend to any set of objects $J$. In the following example computations, $w(x)v$ denotes a write operation of value $v$ to variable $x$. Similarly, $r(x)v$ denotes a read operation of variable $x$ returning $v$. We write the process identifier as a prefix for its own process computation or computation prefix. The notation $w_p(x)v$ or $r_p(x)v$ is used to emphasize that these operations are performed by process $p$.

### 2.2.1 Sequential Consistency

Sequential Consistency (henceforth abbreviated SC), defined by Lamport [49], is the most widely used memory consistency model. According to Lamport, a multiprocessor is said to be SC if:

> “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 its program.”

Several other papers [9, 36, 34, 59] describe SC; some use a different name or a different, but equivalent, definition.

**Definition 2.2.1** Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P,J)$. Then $C$ is SC if there is a linearization $(O, \overset{L}{\rightarrow})$ such that $(O, \overset{\text{prog}}{\rightarrow}) \subseteq (O, \overset{L}{\rightarrow})$.

Dubois, Scheurich and Briggs define strong ordering as a sufficient condition for SC [22] and Goodman states that “A system that adheres to this level of consistency is said to be a strongly ordered system” [34]. However, Adve and Hill show that strong ordering and
SC are similar, but are not equivalent [5].

2.2.2 Linearizability

An even stronger memory model, Linearizability, was formalized by Herlihy and Wing [36]. For this model it is necessary to know the invocation and response time of each operation. A computation is linearizable if there is an assignment of each timed operation \( o \) to one distinct point after \( time(invoc(o)) \) and before \( time(resp(o)) \) such that the resulting sequential computation is valid for each object. Herlihy and Wing require that each process’s computation is blocking; Definition 2.2.2 extends to non-blocking computations while agreeing with the definition of Herlihy and Wing when a computation is blocking.

**Definition 2.2.2** Let \( O \) be all the timed operations of a computation \( C \) of a multiprocess system \( (P, J) \). Then \( C \) is linearizable if there is a linearization \( (O, \xrightarrow{L}) \) satisfying:

1. \((O, \xrightarrow{prog}) \subseteq (O, \xrightarrow{L})\), and
2. \((O, \xrightarrow{time}) \subseteq (O, \xrightarrow{L})\).

Figure 2.3 shows a linearizable computation [42] with the system \( P = \{p, q\} \) and \( J = \{\text{read/write variable } x\} \). Because the time interval of \( r(x) \) overlaps that of both \( w(x) \) 1 and \( w(x) \) 2 other total orderings may be considered but only the one indicated by the projected dotted lines is valid for object \( x \) (assuming \( x \neq 2 \) initially).

Figure 2.4 shows a computation that is SC but not linearizable. It is SC because there is a valid ordering of all operations that maintains program order. It is not linearizable, since
validity requires that $w(x)3$ precedes $r(x)3$ which disagrees with time order.

Torres-Rojas, Ahamad, and Raynal [68] defined Timed Consistency, which unifies SC and Linearizability. Timed Consistency requires that if a write operation is performed at some point in time $t$, it must be seen by the rest of the system processes at no later than $t + \Delta t$. If $\Delta t$ is zero, then Timed Consistency is Linearizability; if it is infinity, then Timed Consistency is SC. Timed Consistency captures all variants of consistency that requires timing and fall between Linearizability and SC.

\[ p: \begin{array}{c}
 w(x)1 \\
 w(x)2 \\
 r(x)3
\end{array} \]
\[ q: \begin{array}{c}
 r(x)2 \\
 w(x)3
\end{array} \]

Figure 2.4: SC but not linearizable computation

\[ \]

2.2.3 Coherence

Coherence, also called cache consistency [22, 34], is among the weakest consistency conditions. Goodman states that Coherence “only guarantees that accesses to a given memory location are strongly ordered” [34]. Mosberger indicates that “Coherence only requires that accesses are SC on a per-location basis” [59].

**Definition 2.2.3** Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P,J)$. Then $C$ is Coherent if for each object $x \in J$ there is some linearization $(O|_x, \xrightarrow{L_x})$ satisfying $(O|_x, \xrightarrow{\text{prog}}) \subseteq (O|_x, \xrightarrow{L_x})$.

**Computation 1** \[
\begin{align*}
 p &: \begin{array}{c}
 w(x)3 \\
 w(x)1 \\
 r(y)3
\end{array} \\
 q &: \begin{array}{c}
 w(y)3 \\
 w(y)1 \\
 r(x)3
\end{array}
\end{align*}
\]
Computation 1 is Coherent but not SC [42]. The linearizations for objects $x$ and $y$ are $(O|x, \xrightarrow{L^x}) = \langle w(x) 3, r(x) 3, w(x) 1 \rangle$ and $(O|y, \xrightarrow{L^y}) = \langle w(y) 3, r(y) 3, w(y) 1 \rangle$. However, there is no single linearization of all these operations that maintains program order because of the following cycle: $r_q(x) 3 \xrightarrow{L} w_p(x) 1 \xrightarrow{L} r_p(y) 3 \xrightarrow{L} w_q(y) 1 \xrightarrow{L} r_q(x) 3$.

Coherence has been also described as a system where “all writes to the same location are serialized in some order and are performed in that order with respect to any processor” [29] (similarly in [4, 47]). These and other definitions of Coherence are equivalent [42].

### 2.2.4 Pipelined–RAM

Lipton and Sandberg defined the Pipelined Random Access Machine (P-RAM) model of memory consistency [56]. Whereas Coherence orders operations from the point of view of each object, P-RAM orders all operations from the process’s point of view. Each process only “sees” all of its own operations and other process’s writes. In P-RAM there must be, for each process, a linear ordering of its operations and all others’ write operations that maintains program order and is valid. This does not mean that all processes will order all operations in the same way since the interleaving of the operations by different processes can be perceived differently by each process. P-RAM was introduced to capture a model where each process has a local copy of the memory. The propagation of writes to the processes could cause writes to arrive in different orders [59].

The following definition is based on that of Ahamad et al.[9] but reformulated using our framework.\(^5\)

**Definition 2.2.4** Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P,J)$. Then $C$ is P-RAM if for each process $p \in P$ there is a linearization $(O|p \cup O|w, \xrightarrow{L_p})$ satisfying $(O|p \cup O|w, \xrightarrow{\text{prog}}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$.

\(^5\)Other interpretations of the original intentions of Lipton and Sandberg are also possible [69].
Computation 2 \[
\begin{align*}
p &: w(x) 3 \ r(x) 1 \\
q &: w(x) 1 \ r(x) 3
\end{align*}
\]

Coherence and P-RAM are incomparable. Computation 1 is P-RAM as well as Coherent. However, the linearizations \( (O|p \cup O|w, \xrightarrow{L_p}) = \langle w(x) 3, w(x) 1, r(x) 1 \rangle \) and \( (O|q \cup O|w, \xrightarrow{L_q}) = \langle w(x) 1, w(x) 3, r(x) 3 \rangle \) show that Computation 2 [42] is P-RAM since they maintain program order. However, it is not Coherent because it is not possible to construct a linearization of all the operations to variable \( x \) that maintains program order.

Computation 3 \[
\begin{align*}
p &: w(x) 3 \ w(x) 1 \ w(y) 2 \\
q &: r(y) 2 \ r(x) 3
\end{align*}
\]

In Computation 3 [42], the linearizations for the objects \( x \) and \( y \) defined by \( (O|x, \xrightarrow{L_x}) = \langle w(x) 3, r(x) 3, w(x) 1 \rangle \) and \( (O|y, \xrightarrow{L_y}) = \langle w(y) 2, r(y) 2 \rangle \) both maintain program order, and show that the computation is Coherent. Since it is not possible to construct a linearization for the operations by \( q \) together with the writes by \( p \) that extends program order, the computation is not P-RAM.

Computation 3 is Coherent but not P-RAM and Computation 2 is P-RAM but not Coherent. Hence, P-RAM and Coherence are incomparable.

### 2.2.5 Processor Consistency

The term processor ordering was first used by Goodman [34] to capture a consistency condition that is stronger than Coherence but weaker than SC. Many others [9, 29, 47, 59, 28] have used the same term to define memory consistency models that have in common Goodman’s original intentions, but that differ in subtle ways. Here, our discussion is limited to the original definition by Goodman (henceforth abbreviated PC-G). In addition to Coherence, Goodman\(^6\) required that [34]:

"the order in which writes from two processes occur, as observed by themselves or a third process need not be identical, but writes issuing from any

\(^6\)Goodman uses the term weak ordering instead of Coherence. In the literature, weak ordering usually refers to a different consistency model. See Section 2.3.1.
process may not be observed in any order other than that in which they are issued.”

Goodman allows the interleaving of writes by two different processes to be viewed differently by each process, as long as program order and Coherence are maintained. The following definition is based on the interpretation of Ahamad et al. [9].

**Definition 2.2.5** Let $O$ be all the operations of a computation $C$ of a multiprocess system $\langle P, J \rangle$. Then $C$ is **PC-G** if for each process $p \in P$ there is a linearization $(O|p \cup O|w; \mathcal{L}_p)$ satisfying:

1. $(O|p \cup O|w; \mathcal{L}_p) \subseteq (O|p \cup O|w; \mathcal{L}_q)$, and
2. $\forall q \in P$ and $\forall x \in J$, $(O|w|x; \mathcal{L}_p) = (O|w|x; \mathcal{L}_q)$.

Note that in this definition, $O|w|x$ is the same set as $O|w \cap O|x$. It is easy to show that PC-G implies Coherence [69].

**Computation 4**

\[
\begin{align*}
p &: w(x)1 \ w(y)1 \ r(y)3 \\
q &: w(y)3 \ w(x)3 \ r(x)1
\end{align*}
\]

Computation 4 [42] is not PC-G because it is not possible to build the required linearizations for $p$ and $q$ that preserve program-order and agree on the ordering of write operations to the same variable. Precisely, $w_q(y)3$ must succeed $w_p(y)1$ in $p$’s view (i.e., $w_p(y)1 \xrightarrow{\mathcal{L}_p} w_q(y)3$); otherwise $w_p(y)1$ overwrites $w_q(y)3$ and $r_p(y)3$ could not have returned a 3. Similarly, $w_q(x)3 \xrightarrow{\mathcal{L}_p} w_p(x)1$. By Condition 2 of PC-G, $w_q(x)3 \xrightarrow{\mathcal{L}_p} w_p(x)1$. In addition, Condition 1 requires $w_q(y)3 \xrightarrow{\mathcal{L}_p} w_q(x)3 \xrightarrow{\mathcal{L}_p} w_p(x)1 \xrightarrow{\mathcal{L}_p} w_p(y)1$. However, this implies $w_q(y)3 \xrightarrow{\mathcal{L}_p} w_p(y)1$ giving rise to a cycle.

However, the linearizations $(O|p \cup O|w; \mathcal{L}_p) = \langle w_p(x)1, w_p(y)1, w_q(y)3, r_p(y)3, w_q(x)3 \rangle$ and $(O|q \cup O|w; \mathcal{L}_q) = \langle w_q(y)3, w_q(x)3, w_p(x)1, r_q(x)1, w_p(y)1 \rangle$ show that Computation 4 is P-RAM.

The first condition of PC-G is exactly the definition of P-RAM. Furthermore, since Computation 4 is P-RAM, PC-G is strictly stronger than P-RAM. Although PC-G implies both P-RAM and Coherence, a computation that is both Coherent and P-RAM is not neces-
2.2. Pure Models

sarily PC-G, contrary to some previous claims (see [45] for example). Computation 4 illustrates this since the linearizations \((O|x, \xrightarrow{\text{L}} x) = \langle w_q(x)3, w_p(x)1, r_q(x)1 \rangle\) and \((O|y, \xrightarrow{\text{L}} y) = \langle w_p(y)1, w_q(y)3, r_p(y)3 \rangle\) satisfy Coherence. Finally, Computation 1 is PC-G, establishing that PC-G is strictly weaker than SC.

2.2.6 Causal Consistency

Causal Consistency (CC) was introduced by Ahamad et al. [10] to capture a model that is stronger than P-RAM but weaker than SC. Ahamad et al. required:

“Causal memory requires that reads return values consistent with causally related reads and writes.”

Write-Before-Read Order

Define the write-before-read order, denoted \((O, \xrightarrow{\text{wbr}})\), by \(o_1 \xrightarrow{\text{wbr}} o_2\), if, for some \(x \in J\), \(o_1\) is a write \((write, x, (v), \lambda)\) and \(o_2\) is a read \((read, x, \lambda, (v))\); that is, \(o_2\) reads the same value written by \(o_1\).

Causal Order

Define the causal order, denoted \((O, \xrightarrow{\text{causal}})\), by the irreflexive transitive closure of the union of \((O, \xrightarrow{\text{prog}})\) and \((O, \xrightarrow{\text{wbr}})\). Namely, \((O, \xrightarrow{\text{causal}}) = ((O, \xrightarrow{\text{prog}}) \cup (O, \xrightarrow{\text{wbr}}))^+\).

Note that \((O, \xrightarrow{\text{causal}})\) could relate operations by different processes to different variables.

Definition 2.2.6 Let \(O\) be all the operations of a computation \(C\) of a multiprocess system \((P, J)\). Then \(C\) is CC if for each process \(p \in P\) there is a linearization \((O|p \cup O|w, \xrightarrow{L_p})\) satisfying \((O|p \cup O|w, \xrightarrow{\text{causal}}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})\).

It follows from definitions 2.2.6 and 2.2.4 that any CC computation is also P-RAM. However, Computation 5 [10] is P-RAM but not CC. The P-RAM linearizations for the three processes are given by \((O|p \cup O|w, \xrightarrow{L_p}) = \langle w(x)3, w(x)1, w(y)1 \rangle\), \((O|q \cup O|w, \xrightarrow{L_q}) = \langle w(x)3, w(x)1, r(x)1, w(y)1 \rangle\), and \((O|s \cup O|w, \xrightarrow{L_s}) = \langle w(x)3, w(y)1, r(y)1, r(x)3, w(x)1 \rangle\).
Computation 5 is not CC because the only possible write-before-read order is \( w(x)3 \xrightarrow{wbr} r(x)3, w(x)1 \xrightarrow{wbr} r(x)1, \) and \( w(y)1 \xrightarrow{wbr} r(y)1. \) Now, \( w(x)3 \xrightarrow{prog} w(x)1 \) and \( w(x)1 \xrightarrow{wbr} r(x)1 \) which implies that \( w(x)3 \xrightarrow{prog} w(x)1 \xrightarrow{wbr} r(x)1 \xrightarrow{prog} r(y)1 \xrightarrow{wbr} r(y)1 \xrightarrow{prog} r(x)3. \) This will result in an invalid sequence because the last read by \( s \) of \( x \) must return a 1 not a 3 to satisfy causality.

\[ \begin{align*}
    p & : w(x)3 \xrightarrow{wbr} w(x)1 \\
    q & : r(x)1 \xrightarrow{wbr} w(y)1 \\
    s & : r(y)1 \xrightarrow{wbr} r(x)3
\end{align*} \]

Computation 5 can also be shown to be PC-G. In fact, the P-RAM linearizations used above are also PC-G. Moreover, there are computations that are CC but not PC-G. For example, Computation 2 is CC but not PC-G. In summary, SC is strictly stronger than CC which is strictly stronger than P-RAM. CC and PC-G are incomparable. Finally, Computation 5 is Coherent and Computation 2 is CC but not Coherent, establishing that Coherence and CC are incomparable.

### 2.3 Hybrid Models

Hybrid models distinguish between objects. This distinction allows different treatment for each class. Certain objects are meant to be used for synchronization and satisfy stronger consistency conditions than others. This allows utilizing system optimizations while still appearing SC to the programmer.

In Chapter 5, we define one more hybrid model, Java Consistency. Here, we limit our discussion to a widely quoted model, Weak Ordering.

#### 2.3.1 Weak Ordering

Dubois, Scheurich and Briggs were the first to propose weak ordering (WO) [22] (Gharachorloo et al. call it weak consistency [29]). WO distinguishes between two classes of objects, \textit{synchronization} and \textit{base}. Base and synchronization objects are simply read/write
variables. The allowable operations for synchronization objects (called synchronization operations) are still read and write operations; however, they can be distinguished from similar operations performed on base objects (called base operations). In WO, program order for base operations can sometimes be violated. The synchronization operations must satisfy SC, and all processes must view a base operation before (respectively after) a synchronization operation if program order places it before (respectively after) the synchronization operation. Dubois, Scheurich and Briggs also state that operations to the same object must remain in program order. Let $O_s$ denote the synchronization operations (alternatively, $O_s$ denotes the operations applied to synchronization objects).

**Weak Program Order**

Define *weak program order*, denoted $(O, \longrightarrow^{wpo})$, by $o_1 \longrightarrow^{wpo} o_2$ if $o_1 \longrightarrow^{prog} o_2$ and either

1. **same-object**: $o_1$ and $o_2 \in O|x$, for some $x \in J$,
2. **synch-operation**: $o_1 \in O|s$ or $o_2 \in O|s$, or
3. **transitivity**: $\exists o' \text{ such that } o_1 \longrightarrow^{wpo} o' \longrightarrow^{wpo} o_2$.

**Definition 2.3.1** *Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P,J)$. Then $C$ is WO if for each process $p \in P$ there is some linearization $(O|p \cup O|w, \longrightarrow^{L_p})$ satisfying:

1. $(O|p \cup O|w, \longrightarrow^{wpo}) \subseteq (O|p \cup O|w, \longrightarrow^{L_p})$, and
2. $\forall q \in P (O|s, \longrightarrow^{L_q}) = (O|s, \longrightarrow^{L_q})$.***

Let $WO_{base}$ denote WO defined on base objects only. This implies that $O|s = \emptyset$. $WO_{base}$ is even weaker than P-RAM. Condition 1 of Definition 2.3.1 only guarantees maintenance of weak program order, whereas P-RAM computations must maintain program order.\(^7\) $WO_{base}$ is also weaker than Coherence. Even though $WO_{base}$ ensures that program order on a per object basis is maintained, not all processes necessarily see writes to the same

\(^7\)Just as Coherence can be viewed as SC on a per-object basis, $WO_{base}$ can be viewed as P-RAM on a per-object basis.
object in the same order, as is required in Coherent systems. Dubois, Scheurich and Briggs however, seem to imply that a system should be Coherent and satisfy the conditions of WO$_{\text{base}}$ simultaneously [22]. The following defines a weakly-ordered-Coherent memory model (WOC). With only base operations, WOC$_{\text{base}}$ becomes equivalent to Coherence.

**Definition 2.3.2** Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P,J)$. Then $C$ is WOC if for each process $p \in P$ there is some linearization $(O|p \cup O|w, \xrightarrow{L_p})$ satisfying the two conditions of Definition 2.3.1 and $\forall q \in P \land \forall x \in J \ (O|w|x, \xrightarrow{L_q}) = (O|w|x, \xrightarrow{L_q})$.

### 2.3.2 Other Hybrid Models

Following Dubois, Scheurich, and Briggs [22], Gibbons, Merritt, and Gharachorloo [30, 31] used the same idea of distinguishing object types to define Release Consistency. While Weak Ordering classifies operations as either ordinary or synchronization, Release Consistency further classifies synchronization operations into acquire and release operations. When all acquires and releases are guaranteed to behave in a SC manner, the resulting model is denoted RC$_{\text{SC}}$. When these are guaranteed to behave according to Processor Consistency$^8$, RC$_{\text{PC}}$ denotes the resulting model. Using similar ideas, Attiya and Friedman defined Hybrid Consistency [15].

### 2.4 Summary

Figure 2.4 summarizes the relationships between the memory models defined in this chapter. In this figure, an arrow from model $A$ to model $B$ indicates that $A$ is strictly stronger than $B$. If no directed path exists from $A$ to $B$, then $A$ and $B$ are incomparable.

$^8$The Processor Consistency version Gibbons, Merritt, and Gharachorloo used is different from PC-G.
2.5 Non-Terminating Computations

The above memory consistency model definitions require linearization(s) over set(s) of operations. Since these linearizations are meant to capture each system component’s “view” of the computation, we expect that, as the computation continues, processes extend their respective views, but do not “change their minds” about what happened previously.

In certain systems, as we shall see in the subsequent chapters, a write operation could be visible to a set of processes but invisible to others. Certain kinds of invisibility become problematic in non-terminating systems. Consider Computation 6. Process $q$’s $w(x)1$ is forever invisible to $p$ ($p$ is repeatedly reading 0 for $x$, never seeing that its value has changed to 1)

Computation 6 \[
\begin{align*}
    p &: [r(x)0][r(x)0][r(x)0][r(x)0]... \\
    q &: w(x)1
\end{align*}
\]

Is this computation Coherent? Since we are dealing with a non-terminating computation, only finite prefixes of the computation can be considered for consistency purposes. In
fact, any finite prefix of Computation 6 is Coherent. In particular, \([r(x)0]^i w(x)1\) for any finite \(i\) is a linearization that adheres to Definition 2.2.3.

Semantically, what is intended by the definition of Coherence is that there is a unified “view” of all the operations to the same object by all processes. The linearization \([r(x)0]^i w(x)1\) implies that \(q\)’s \(w(x)1\) could have been visible to \(p\) only after its \(i^{th}\) \(r(x)0\).

Any linearization of any finite prefix of Computation 6 that includes \(q\)’s \(w(x)1\) will have a semantic disagreement with that of any other prefix, shorter or longer. Such linearizations represent a “change in mind” by having, for example, \(q\)’s \(w(x)1\) happen after \(p\)’s \(i^{th}\) read, but later only after its \(j^{th}\). However, this does not capture the intended meaning of Coherence. Intuitively, we expect on-going computations as they are paused, resumed, and paused again to extend what has been observed at earlier pauses.

Let \(O\) be all the operations of some finite computation \(C\) of a system \((P,J)\), and let \(D\) be a memory consistency model. To establish that \(C\) satisfies consistency model \(D\), we provide a set of sequences \(S\), each composed of operations in \(O\), that satisfy the constraints of \(D\). Each sequence is meant to capture a component’s “view” of the computation, and some kind of agreement between such views. Call such an \(S\) a set of satisfying sequences for \((C,D)\).

For the memory consistency definition \(D\) to hold for a non-terminating computation, \(C\), we (informally) have three requirements. First, if \(C\) is paused, then the prefix, say \(\hat{C}\), that has been realized so far, should have satisfying sequences \(\hat{S}\) for \((\hat{C},D)\). Second, if \(C\) is resumed and paused again later, say at \(\tilde{C}\), then there are satisfying sequences \(\tilde{S}\) for \((\tilde{C},D)\) and the satisfying sequences \(\tilde{S}\) are “extensions” of the satisfying sequences \(\hat{S}\). That is, a component is prohibited from reconstructing its view of what the computation did in the past. The third requires that if a process makes some progress (by performing some operations), eventually these operations will be included in some satisfying sequences. This is a technical condition without which the first two conditions can always be vacuously satisfied. We formalize this intuition as follows.

A sequence \(s\) extends \(\hat{s}\) if \(\hat{s}\) is a prefix of \(s\). A list of sequences \(S = (s_1, \ldots, s_n)\) extends the
list of sequences $\hat{S} = (\hat{s}_1, \ldots, \hat{s}_n)$ if for each $i$, $s_i$ extends $\hat{s}_i$. Recall that a system computation $C$ is a collection of process computations. A process $p$ computation is denoted $C(p)$. Thus, $C = \bigcup_{p \in P} \{C(p)\}$.

**Definition 2.5.1** Let $D$ be a memory consistency model as defined for terminating computations. A non-terminating computation $C = \bigcup_{p \in P} \{C(p)\}$ satisfies $D$ if $\forall p \in P, \exists$ an infinite sequence $(C^p_1, S_1), (C^p_2, S_2), \cdots$, satisfying

1. each $C^p_i$ is a prefix of $C$ such that $C^p_i(p)$ contains exactly the first $i$ operations of $C(p)$ and $C^p_i$ extends $C^p_i$,
2. each $S_i$ is a set of satisfying sequences for $(C_i, D)$ and $S_{i+1}$ extends $S_i$, and
3. $\forall q \neq p, \forall \hat{\hat{C}}(q)$ a finite prefix of $C(q)$, $\exists i$ such that $\hat{\hat{C}}(q)$ is a prefix of $C^p_i(q)$.

If we apply Definition 2.5.1 to Definition 2.2.3, we confirm our intuition that Computation 6 is not Coherent. That is, any linearization of a finite prefix of the computation that contains $q$’s write and satisfies Definition 2.2.3 cannot be extended to a linearization for a longer prefix of the computation that still satisfies the definition. Instead, the write operation by $q$ would have to be moved to the new end of the linearization.

Let $C$ and $\hat{\hat{C}}$ be two computations such that $C$ extends $\hat{\hat{C}}$, and let $D$ be a memory consistency model such that both $C$ and $\hat{\hat{C}}$ satisfy $D$. Then, $(C, D)$ extends $(\hat{\hat{C}}, D)$ if the satisfying sequences $S$ for $(C, D)$ extend the satisfying sequences $\hat{S}$ for $(\hat{\hat{C}}, D)$.
CHAPTER 3

Operational Consistency Models

An additional goal of this dissertation is to associate an abstract multiprocess system providing a non-operational memory consistency model with an actual system implementation, which is normally described operationally. These implementations range from hardware architectures to software protocols and are called machines in this dissertation.

The non-operational memory consistency models are described by constraints on ordering of operations applied to simple objects, primarily read/write variables. This simplicity is maintained at the expense of complex consistency conditions. On the other hand, the operational memory consistency models are given in terms of complex objects but with simple consistency conditions, normally Sequential Consistency. The goal of this chapter is to develop the techniques that will allow us to prove the equivalence between an operational and a non-operational memory consistency model.

3.1 Example Machine

The objective of this chapter is motivated by an example. Consider the machine $M_C$ [69, 42] depicted in Figure 3.1. This machine consists of $n$ processors ($p_1$ to $p_n$), $m$ memory locations ($l_1$ to $l_m$), in addition to communication channels. Each location $l_j$ is single-ported, as represented by the switch $s_j$. Each processor $p_i$ is connected to each switch $s_j$ by a First-In-First-Out (FIFO) bi-directional channel. Processors communicate via the shared memory locations by using read and write instructions.

A processor $p_i$ in $M_C$ implements a read invocation $\langle read, l_j, \lambda \rangle$ by the following or-
ordered events, where v is a constant value:

1. \( p_i \) places a \textit{load-request}(\( p_i, l_j \)) on the channel connecting \( p_i \) to \( s_j \).

2. \( l_j \) places a \textit{load-reply}(\( p_i, l_j, v \)) on the same channel.

Similarly, a write invocation \((\text{write}, l_j, (v))\) by processor \( p_i \) is implemented by the following ordered events:

1. \( p_i \) places a \textit{store-request}(\( l_j, v \)) on the channel connecting \( p_i \) to \( s_j \).

2. \( l_j \) performs \textit{update}(\( l_j, v \)).

Although the description of \( M_C \) is simple, the memory consistency model that arises from this machine might not be so. Furthermore, existing machines are rarely as simple as \( M_C \) and their specifications are not always presented with care, further complicating our objective. To define the memory consistency model that is supported by \( M_C \), it is helpful to precisely establish the relationship between each high-level operation, such as a read or a write, and the low-level implementation in terms of machine events, such as \textit{load-request} and \textit{load-reply}.
3.2 Machine Models

Every (multiprocess) machine has several components. The interconnection network and memory ports are examples of components. Other essential components include processors and space. Space is a set of memory locations. For example, caches, write buffers, and main memory are all part of the machine space. Processors (processing units) have unique identifiers, normally from 1 to $n$. A machine implements an operation by proceeding through a sequence of events. One event of the sequence occurs at a processor and the rest occur at the various components of the machine. There may be outputs produced by one or more events in the sequence. Therefore, a machine is described by specifying the possible sequence of events for each possible operation invocation that it supports, and any restrictions that these sequences satisfy.

Processors are associated with “request” events only. Space is associated with “respond” and “update” events. For every operation invocation, there is a set of specified events that supplies an output to the processor.\(^1\)

Formally, a machine $M$ is a six-tuple $M = (\Pi, \Sigma, N, E, I, R)$ where:

- $\Pi$ is a set of (virtual) processors with unique identifiers.
- $\Sigma$, the space, is a collection of locations. $\Sigma$ is machine-dependent because machines differ in memory hierarchy architecture. For example, some machines incorporate local memories and/or write buffers, while others do not. Also, space is accessed differently in different ways in different machines. We make those distinctions when we define the events that access $\Sigma$.
- $N$ is the set of operation-invocations supported by $M$.
- $E$ is a specified set of machine events. Each $e \in E$ is represented by

  
  \[ \text{event-type}(a_1, a_2, \ldots) \]

\(^1\)For the machines described in this dissertation, at most one event supplies an output to the processor.
where the $a_i$’s are appropriate parameters. Some of these parameters are used to carry values that are assigned to memory locations, or that are returned from these locations. These are called parameter variables, and are denoted by $v$ (for a known value) or $v?$ (for an unknown uninstantiated value). The typewriter font is used to represent events. For example $\text{load-request}(p, x)$ is a load-request event by process $p$ applied to memory location $x$, and $\text{load-reply}(p, x, v?)$ is a load-reply event that returns the “not yet” instantiated value $v?$ for location $x$. The notation $v? \leftarrow v$ indicates the instantiation of $v?$ with $v$. A subset of $E$ is called the request events and comprises the events that are initiated by processors.

- $I$ is called the implementation function. It specifies, for each operation invocation in $N$, a finite set of finite sequences of events in $E$. Each sequence in this set represents a “possible implementation” of the given operation invocation. Specifically, let $E^*$ denote the set of finite sequences of events from $E$, then

$$I : N \times \Pi \rightarrow 2^E.$$  

For an operation $o$, if $I(\text{invoc}(o), p) = \{s_1 = \langle e_1^1, \cdots, e_1^1 \rangle, s_2 = \langle e_1^2, \cdots, e_1^2 \rangle, \cdots, s_l = \langle e_k^l, \cdots, e_k^l \rangle\}$, then when $\text{invoc}(o)$ by $p$ is implemented on $M$, it is converted to exactly one of the sequences $s_1$ to $s_l$, call it $s_i$. At least one element of $s_i$ is a request event.\(^2\) If $s_i = \langle e_1, e_2, \ldots \rangle \in I(\text{invoc}(o), p)$, we say $o$ (also, $\text{invoc}(o)$) corresponds to $s_i$ and to each event in the sequence. Finally, each event in the sequence $s_i$ matches every other event in the same sequence. If $\text{invoc}(o_1)$ precedes $\text{invoc}(o_2)$ in $p$ (alternatively, $o_1 \overset{\text{prog}}{\rightarrow} o_2$) and $e_1$ and $e_2$ are request events that correspond to $o_1$ and $o_2$, respectively, then we say $e_1 \overset{\pi}{\rightarrow} e_2$. The order $(E, \overset{\pi}{\rightarrow})$ is called the request order and will be dealt with more formally later in the chapter.

- $R$ is a collection of rules that restrict the possible interleavings of the sequences of events, and dictate how parameter variables are instantiated. Specifically, when a

\(^2\)The machines described in this dissertation have exactly one request event in each sequence.
system \((P,J)\) executes on \(M\), a sequence \(\Xi\) of events, called a run, results. Constraints on \(\Xi\) are given by \(R\). The subset of \(R\) that is concerned with instantiation of parameter variables is referred to as the validity rules and is denoted \(R_v\). Finally, \(R_o = R \setminus R_v\) is concerned with ordering restrictions only and is called the ordering rules.

### 3.3 Describing \(M_C\)

As mentioned earlier, when a system \((P,J)\) executes on a machine \(M\), it produces a run \(\Xi\). We will deal with \(\Xi\) more formally in the next section. The run \(\Xi\) is consistent with the program order of \((P,J)\). That is, \(\Xi\) respects the request order \((E, \rightarrow^\pi)\). The notation \(e_1 \rightarrow^\Xi e_2\) denotes that event \(e_1\) precedes event \(e_2\) in \(\Xi\). Finally, \(\Xi\{x\} \text{ and } \Xi\{p\}\) respectively denote the subsequences of \(\Xi\) that consist only of events applied to object (location) \(x\) and consequently of events initiated by process (processor) \(p\), respectively.

\[ M_C = (\Pi, \Sigma, N, E, I, R), \]

where
- \(\Pi = \{p_1, \ldots, p_n\}\).
- \(\Sigma = \{l_1, \ldots, l_m\}\).
- \(N = \bigcup_{x \in \Xi, v \in V} \{(\text{read}, x, \lambda), (\text{write}, x, (v))\}\).
- \(E = \bigcup_{p \in \Pi, x \in \Sigma, v \in V} \{\text{load-request}(p, x), \text{load-reply}(p, x, v?), \text{store-request}(x, v), \text{update}(x, v)\}\).

The request events in \(E\) are the load-request and store-request events.

- \(I\) is given as follows.
  - \(I((\text{read}, x, \lambda), p) = \{\text{load-request}(p, x), \text{load-reply}(p, x, v?)\}\)\).
  - \(I((\text{write}, x, (v)), p) = \{\text{store-request}(x, v), \text{update}(x, v)\}\)\).

- \(R = \{\rho_o, \rho_v\}\) where the rules are specified by the following:
  - \(\rho_o\): Let \(\langle e_1, e_2 \rangle \in I(\text{invoc}(o), p)\), and \(\langle e'_1, e'_2 \rangle \in I(\text{invoc}(o'), p)\), where \(o\) and \(o'\) are operations and \(p\) is a processor. If events \(e_1\) and \(e'_1\) are on the same location, then \(e_1 \rightarrow^\Xi e'_1\) if and only if \(e_2 \rightarrow^\Xi e'_2\).
– P.: The instantiation load-reply(p, x, v? ← v) takes place such that the most recent update event that precedes the reply in Ξ|x is update(x, v).

In further chapters, some more complicated machines and memory consistency models will be defined. The technique for proving their equivalence is described in the next section.

3.4 Relating Systems and Machines

The question is how to relate an abstract system (P, J) with a non-operational memory consistency model and a machine that is described operationally. For instance, how do we establish the relationship between the machine MC and SC, Coherence, or P-RAM defined in Chapter 2?

When a system (P, J) is executed on a machine M = (Π, Σ, N, E, I, R), each operation invocation o of each process p ∈ P is replaced by a sequence of events in I(invoc(o), p). The order of two request events issued by p follows the program order of their corresponding operations. Further ordering constraints are imposed by R. The run of M that results is a total order on all the resulting events that complies with I, program order, and R. Such a run is interpreted by (P, J) as a computation. This is formalized as follows.

The system (P, J) is implemented by the machine M by having a bijection between P and Π and J and Σ.

Let M = (Π, Σ, N, E, I, R) be a machine and (P, J) be a system with operation invocations in N. Machine M implements (P, J) by replacing operation invocations in each p ∈ P by sequences of events. We will overload the notation and let E represent the specific set of events occurring in a run.

---

3 A bijection between processes P and processors Π is always sufficient, even though certain systems might have more processes than processors.

The more general case would be to have an onto mapping from Σ to J. In this dissertation, we mainly deal with read/write variables and a bijection between J and Σ is sufficient.
3.4. RELATING SYSTEMS AND MACHINES

Request Order

Define the request order on request events, denoted \((E, \pi \rightarrow)\), by \(e_1 \pi \rightarrow e_2\) if and only if \(s_1 \in I(invoc(o_1), p)\) and \(s_2 \in I(invoc(o_2), p)\) for \(p \in P\) (\(invoc(o_1)\) and \(invoc(o_2)\) are both in \(p\)) and \(e_1\) is a request event in \(s_1\), \(e_2\) is a request event in \(s_2\), and \(o_1 \rightarrow prog o_2\).

A (system) yield of \((P, J)\) on \(M\) is a partial order \((E, \gamma \rightarrow)\) satisfying:

- \(e \in E\) if and only if \(e\) corresponds to an operation invocation in process \(p\),
- \((E, \gamma \rightarrow)\) obeys the order defined by \(I\) between matching events,
- \((E, \gamma \rightarrow)\) obeys \(R_o\), and
- \((E, \pi \rightarrow) \subseteq (E, \gamma \rightarrow)\).

The set of all possible yields of \((P, J)\) on \(M\) is denoted by \(M(P, J)\). A (system) run of \((P, J)\) on \(M\) is a total order \(\Xi\) that extends a yield in \(M(P, J)\) into a total order and instantiates all parameter variables according to \(R_v\). The set of all possible runs of \((P, J)\) on \(M\) is denoted by \(M^\Xi(P, J)\). Any run of \((P, J)\) on \(M, \Xi\), induces a computation of \((P, J), C_\Xi\), by instantiating the parameters variables in each operation response.

Machine \(M\) provides memory consistency model \(D\) if for every system \((P, J)\) with operation invocations in \(N\) and for every run \(\Xi \in M^\Xi(P, J), C_\Xi\) satisfies \(D\). \(M\) realizes \(D\) if for every system \((P, J)\) and for every computation \(C\) of \((P, J)\) that satisfies \(D\), there exists a run \(\Xi \in M^\Xi(P, J)\) such that \(C_\Xi = C\). Finally, machine \(M\) exactly implements \(D\) if \(M\) provides \(D\) and \(M\) realizes \(D\).

Figure 3.2 summarizes the major terms used in both the operational and non-operational models.

Template for Proofs

A system \((P, J)\) and a consistency condition \(D\) give rise to a set of possible computations as described in Chapter 2. Alternatively, a system \((P, J)\) and a machine \(M = (\Pi, \Sigma, N, E, I, R)\) give rise to a set of possible runs, each of which instantiates the variables of the operation invocations of \((P, J)\) and thus induces a computation of \((P, J)\). To show that \(M\) exactly
Chapter 3. Operational Consistency Models

<table>
<thead>
<tr>
<th>Non-operational</th>
<th>Operational</th>
</tr>
</thead>
<tbody>
<tr>
<td>(multiprocess) system with a memory consistency model</td>
<td>(multiprocess) machine</td>
</tr>
<tr>
<td>process</td>
<td>processor or thread</td>
</tr>
<tr>
<td>(data) object</td>
<td>(memory) location</td>
</tr>
<tr>
<td>operation</td>
<td>event</td>
</tr>
<tr>
<td>program order ((O, \rightarrow_{\text{prog}}))</td>
<td>request order ((E, \rightarrow_{\text{req}}))</td>
</tr>
</tbody>
</table>

Figure 3.2: Terms used in operational and non-operational models

implies \(D\), we show that these two sets of computations are equivalent.

That is, to show that a machine \(M = (\Pi, \Sigma, N, E, I, R)\) exactly implements a consistency condition \(D\), we establish that for every system \(\langle P, J \rangle\) with operation invocations in \(N\), every computation of \(\langle P, J \rangle\) that is induced by a run on \(M\) satisfies \(D\), and every computation of \(\langle P, J \rangle\) that satisfies \(D\) is the induced computation of some run of \(\langle P, J \rangle\) on \(M\). Thus, we show that the diagram in Figure 3.3 commutes.

Figure 3.3: Proving equivalence between operational and non-operational models

\(M\): machine, \(D\): memory consistency model, \(\{C\}\): set of computations, \(\Xi\): set of runs.
3.5 Example Proof

The memory consistency model arising from the machine $M_C = (\Pi, \Sigma, N, E, I, R)$ of Section 3.1 is Coherence (Definition 2.2.3). In fact, $M_C$ exactly implements Coherence. We now have the required formalism to prove this claim.

**Claim 3.5.1** $M_C$ exactly implements Coherence.

**Proof:** Let $(P, J)$ be a multiprocess system with operation invocations in $N$. Let $\Xi$ be a run of $(P, J)$ on $M_C$. That is, $\Xi \in M_C^\Xi(P, J)$.

- **$M_C$ provides Coherence**
  Let $C_\Xi$ be the computation of $(P, J)$ induced by $\Xi$. Let $O$ be all the operations that result from $C_\Xi$. We show that $C_\Xi$ satisfies Coherence.
  
  For each $x$, construct the order $(O|x, \mathcal{L}_x)$ as follows. Let $o_1$ and $o_2$ be operations in $O|x$, and let $\langle e_1, e'_1 \rangle \in I(\text{invoc}(o_1), p)$ and $\langle e_2, e'_2 \rangle \in I(\text{invoc}(o_2), p)$, then $o_1 \overset{\Xi}{\rightarrow} o_2$ if $e'_1 \overset{\Xi}{\rightarrow} e'_2$. That is, the order $(O|x, \mathcal{L}_x)$ is induced by the corresponding load-reply and update events in $\Xi$. To see that $(O|x, \mathcal{L}_x) \subseteq (O|x, \mathcal{L}_x)$, let $o_1$ and $o_2$ be as defined above. If $o_1 \overset{\text{prog}}{\rightarrow} o_2$, then $e_1 \overset{\pi}{\rightarrow} e_2$. Rule $\rho_o$ implies that $e'_1 \overset{\Xi}{\rightarrow} e'_2$, and consequently, $o_1 \overset{\mathcal{L}_x}{\rightarrow} o_2$ by construction. Finally, $(O|x, \mathcal{L}_x)$ is valid by the validity rule $\rho_v$.

- **$M_C$ realizes Coherence**
  Let $C$ be any Coherent computation of any system $(P, J)$. We show how to construct $\Xi \in M_C^\Xi(P, J)$ such that $C_\Xi = C$. Let $O$ be all the operations that result from $C$.
  
  By definition of Coherence, there is a linearization $(O|x, \mathcal{L}_x)$, for each $x$ in $J$, that extends program order. First, we build $S_1$, called the *request sequence*. Initially, $S_1$ is empty. For each process $p$, iterate through $p$’s computation considering one operation $o$ at a time. If $o$ is $(\text{read}, x, \lambda, (v))$ by $p$, append to $S_1$ load-request($p, x$). If $o$ is $(\text{write}, x, (v), \lambda)$, append to $S_1$ store-request($x, v$). Similarly, we build $S_2$, the *reply sequence*, as follows. For each object $x$, iterate through $(O|x, \mathcal{L}_x)$ considering one operation $o$ at a time. If $o$ is $(\text{read}, x, \lambda, (v))$ by $p$, append to $S_2$ load-reply($p, x, v$).
If $o$ is \((\text{write}, x, (v), \lambda)\), append to $S_2 \text{update}(x, v)$. Finally, define $\Xi$ to be $S_1||S_2$.\(^4\)

Notice that each \((\text{read}, x, \lambda, (v))\) operation in $O$ has been replaced by two matching events, \(\text{load-request}(p, x)\) and \(\text{load-reply}(p, x, v)\), such that the request event precedes the reply event. Also, each \((\text{write}, x, (v), \lambda)\) in $O$ has been replaced by corresponding \(\text{store-request}(x, v)\) and \(\text{update}(x, v)\) events, such that the request event precedes the update event. Therefore, $\Xi$ obeys $I$. Furthermore, the request events have been inserted into $\Xi$ respecting the program order of their corresponding operations, thus respecting the request order \((E, \pi)\).

To show that $\Xi$ satisfies $\rho_o$, let $o_1, o_2 \in O[x]|p$ for some variable $x$ and process $p$, such that \(\langle e_1, e_1' \rangle \in I(\text{invoc}(o_1), p)\) and \(\langle e_2, e_2' \rangle \in I(\text{invoc}(o_2), p)\). If $e_1 \xrightarrow{\pi} e_2$, then $o_1 \xrightarrow{\text{prog}} o_2$; it follows from the construction of $S_2$ that $e_1' \xrightarrow{\Xi} e_2'$ since $(O|x, \xrightarrow{\text{prog}}) \subseteq (O|x, \xrightarrow{L_x})$. For the other direction, if $e_1' \xrightarrow{\Xi} e_2'$, then it must be the case that $o_1 \xrightarrow{\text{prog}} o_2$, and consequently, $e_1 \xrightarrow{\pi} e_2$. Finally, $\Xi$ is valid (obeys $\rho_v$) because each $(O|x, \xrightarrow{L_x})$ is a linearization.

\[\blacksquare\]

In further chapters, some more complicated machines and memory consistency models will be defined and the framework illustrated here will be used to establish their equivalence.

### 3.6 Non-Terminating Systems

The proof of Claim 3.5.1 assumes that the computation $C$ and the run $\Xi$ are finite. The same proof can be adapted to apply to non-terminating computations and runs, as shall be shortly seen. In other cases, such an extension to non-terminating systems might not be as straightforward as the case of Claim 3.5.1.

To show that $M$ provides $D$, we show that any run $\Xi$ of a certain system on $M$ induces

\(^4\)|| indicates string concatenation.
3.6. Non-Terminating Systems

a computation, $C_\Xi$, that satisfies $D$. To show $C_\Xi$ satisfies $D$, the proofs provide a set of satisfying sequences for $(C_\Xi, D)$. To satisfy Definition 2.5.1, the satisfying sequences will not be allowed to disagree with the satisfying sequences corresponding to any prefix of $\Xi$.

Precisely, to show that machine $M$ provides consistency condition $D$, we show that for any non-terminating run $\Xi^\infty$, the induced finite computation $C_\hat{\Xi}$ of any finite prefix $\hat{\Xi}$ of $\Xi^\infty$ satisfies $D$ and for any prefix $\Xi$ of $\Xi^\infty$ that extends $\hat{\Xi}$, $(C_\Xi, D)$ extends $(C_\hat{\Xi}, D)$. The idea is depicted in Figure 3.4(a).

To show that $M$ realizes $D$, we show that any computation $C$ that satisfies $D$ is induced by a run $\Xi$ of some system on $M$. So when constructing a prefix of $\Xi$ that corresponds to a prefix of $C$, we should expect a longer prefix of $\Xi$ that corresponds to a longer prefix of $C$ to agree with any shorter prefix.

Specifically, to show that machine $M$ realizes consistency condition $D$, we show that for any non-terminating computation $C^\infty$ that satisfies $D$, the corresponding finite run $\hat{\Xi}$ to any finite prefix $\hat{C}$ of $C^\infty$ is a possible run on $M$ and for any prefix $C$ of $C^\infty$ that extends $\hat{C}$, the corresponding run $\Xi$ of $C$ extends $\hat{\Xi}$. The procedure is depicted in Figure 3.4(b).

**Revisiting Claim 3.5.1**

- **$M_C$ provides Coherence**
  
  The construction ignores the load-request and store-request events. As the run $\Xi$ continues, the same construction procedure guarantees that the new linearizations preserve the old ones completely.

- **$M_C$ realizes Coherence**
  
  If $C$ is non-terminating, the prefix considered so far induces the run $S_1||S_2$. When $C$ is resumed and paused again, the new induced run will be $S_1||S_2||\hat{S}_1||\hat{S}_2$, where $\hat{S}_1$ and $\hat{S}_2$ are respectively the request and reply sequences corresponding to operations that have not been considered in the previous prefix (those that have just been added when $C$ is resumed). It is trivial that $S_1||S_2||\hat{S}_1||\hat{S}_2$ extends $S_1||S_2$. 

Determined Operations

Let $\Xi$ be a finite prefix of a non-terminating run. The run $\Xi$ might include some events whose matching events have not occurred yet. An operation corresponding to such events (the operation implementation is not completed yet) is called undetermined, otherwise it is called determined.

In $M_C$, the extension of $(C_\hat{\Xi}, D)$ to $(C_\Xi, D)$ is straightforward. Since the construction in the provides-direction considers only reply and update events, all the operations considered are determined. In other systems, not all operations under consideration are necessarily determined. These undetermined operations require special treatment as argued below.

In certain systems, the extension of $(C_\hat{\Xi}, D)$ to $(C_\Xi, D)$ is not straightforward. Consider a machine where processors utilize private caches. An update to a local cache will be immediately visible to the updating processor, but the update might be delayed before it is
made visible to other processors in the machine. Assume that the run being considered, \( \hat{\Xi} \), contains the local update but the global update(s) did not take place yet. Even though the implementation of the corresponding write operation is not “completed” yet, it might not be possible to exclude such an operation from \( C_{\hat{\Xi}} \), the computation induced by \( \hat{\Xi} \). For instance, \( \hat{\Xi} \) might include events corresponding to a read operation that returns the value written by the undetermined write. Since the implementation of such a read could be completed, the read must be included in \( C_{\hat{\Xi}} \). Thus, the undetermined write operation must also be included in \( C_{\hat{\Xi}} \). When \( \hat{\Xi} \) is extended to some \( \Xi \) that includes the global update corresponding to the write operation, the satisfying sequences for \( (C_{\Xi}, D) \) might not literally extend those for \( (C_{\hat{\Xi}}, D) \). This is because the undetermined write might be reordered in some satisfying sequences to capture the “time” of its completion. Therefore, when we say \( (C_{\Xi}, D) \) extends \( (C_{\hat{\Xi}}, D) \), we allow the undetermined operations to be reordered in the satisfying sequences.

Call a determined satisfying sequence the subsequence of a satisfying sequence \( S \) that consists entirely of all the determined operations in \( S \). The notion of \( (C, D) \) extending \( (\hat{C}, D) \) introduced in Chapter 2 is revised as follows.

**Definition 3.6.1** Let \( C \) and \( \hat{C} \) be two computations such that \( C \) extends \( \hat{C} \), and let \( D \) be a memory consistency model such that both \( C \) and \( \hat{C} \) satisfy \( D \). Then, \( (C, D) \) extends \( (\hat{C}, D) \) if the determined satisfying sequences \( \hat{S} \) for \( (C, D) \) extend the determined satisfying sequences \( \hat{S} \) for \( (\hat{C}, D) \).

The realizes-direction does not need a similar treatment. When we are given a computation, all the operations are determined.

### 3.7 Discussion

Another approach to define machines would be defining another multiprocess system \((\Pi, \Sigma)\), where the objects in \( \Sigma \) are locations with the allowable operations that we called events in Section 3.2. The implementation of a system \((P, J)\) with a (non-operational) memory consistency model by a system \((\Pi, \Sigma)\) with an (operational) memory consistency model is a
mapping from $P$ to $\Pi$ and from $J$ to $\Sigma$.

Although natural, this approach gives rise to highly complicated objects especially in complicated machines, which in turn complicates our proofs. To simplify this task, we defined a machine by separating the operations on objects (events) from these objects (memory locations) and by considering the implementation functions to be part of the machine itself. This approach simplified the proofs in the following chapters.
CHAPTER 4

SPARC Consistency

The next chapters define memory consistency models associated with state-of-the-art multiprocess systems. Two SPARC consistency models are defined in this chapter. First, we give the operational description of each. Next, we derive the non-operational definitions and prove them equivalent to the operational descriptions. Finally, these models are compared with the models discussed in Chapter 2.

4.1 Operational Description

Figure 4.1 depicts a two-processor SPARC architecture [67, 71]. An axiomatic description of the operational behavior is quoted from the manual [67] in Appendix B. In SPARC, there is one store-buffer associated with each processor in the system. The main memory is single ported with a non-deterministic switch providing one memory access at a time.

Each buffer operates in parallel with the processor. When a processor performs a store\(^1\), it does not wait for it to be committed to main memory. Instead, the store is sent to the store buffer, which is responsible for committing the pending stores to main memory.

When a load is issued by a processor, the associated store-buffer is checked for any pending stores to the same location. If there is any such store, the value “to be written” by the last such store is returned. In this case, the load completes without accessing main memory. The load accesses main memory in the normal manner when there are no pending

\(^1\)A store updates a memory locations and a load investigates a location.
stores to the same location in the associated buffer.

The order in which the store-buffer commits the stores to main memory differentiates between two memory consistency models adopted by the SUN SPARC v8. If the buffer is FIFO, the resulting model is called Total Store Ordering and abbreviated TSO. In some implementations, the buffer is guaranteed to be FIFO only on a per-location basis. In this case, the resulting memory consistency model is called Partial Store Ordering and abbreviated PSO.

SPARC supports a kind of atomic read-modify-write instruction called atomic-load-store. As the name suggests, this instruction is both a load and a store. Therefore, it is sent to the buffer like any other store. Since loads are blocking an atomic-load-store is also blocking. It is specified in Figure 4.2.

Since TSO buffers are FIFO, an atomic-load-store flushes the store-buffer before the processor is allowed to proceed. However, in PSO an atomic-load-store might over-take other buffered stores and atomic-load-stores because the PSO buffers are only partially FIFO. To force stronger ordering restrictions, an additional fence instruction called a store-barrier is included in the PSO model. A store-barrier does not access memory but is sent to the buffer. It enforces that no store issued after the barrier can be committed to main
memory before any store that is issued before the barrier.

### 4.1.1 Object Types

In order to formally define the SPARC operational models, we need to specify the objects that are supported by a high level system \((P, J)\) that runs on a SPARC machine. In fact, SPARC provides just one kind of variable, which we call a *SPARC variable*. The difference between a SPARC variable and a read/write variable is that the first allows more operations. A SPARC variable, \(x\), is specified by the set of all sequences of operations \(<o_1, o_2, \cdots>\) such that

1. each \(o_i\) is either \((\text{read}, x, \lambda, (v))\) that returns the value \(v\) of variable \(x\), \((\text{write}, x, (v), \lambda)\) that assigns a value \(v\) to \(x\), or \((\text{swap}, x, (v_1), (v_2))\) that returns the value \(v_2\) of \(x\) and assigns the value \(v_1\) to \(x\), and

2. for every read or swap operation, the value returned is the same as the value written by the most recent preceding write or swap operation in the sequence.

Both TSO and PSO use the same variable type, SPARC variables. SPARC’s load, store, atomic-load-store, and store-barrier instructions correspond to the read, write, swap, and barrier operations, respectively.

---

**Figure 4.2**: The SPARC atomic-load-store instruction

\[
\text{atomic-load-store}(x, v) : \\
\begin{array}{l}
\text{begin[blocking]} \\
\quad t \leftarrow x \\
\quad x \leftarrow v \\
\quad \text{return } t \\
\text{end[blocking]}
\end{array}
\]

\(x\): shared location in main memory; \(v\): constant value; \(t\): temporary local location
4.1.2 Operational Total Store Ordering

The machine $M_{TSO} = (\Pi, \Sigma, E, N, I, R)$ is defined as follows. The events of $E$ are of six types: load-request, load-reply, buffer-store, memory-store, buffer-swap, and memory-swap. A load-request$(p, x)$ is a request by processor $p$ to load the value stored in location $x$. A load-reply$(p, x, v?)$ returns a value, $v?$, of $x$ to $p$ either from main memory or from the buffer. A buffer-store$(p, x, v)$ is a store request by $p$ to $x$ of a value $v$; the request is placed in the store-buffer associated with $p$. Similarly, memory-store$(p, x, v)$ commits $p$'s request, buffer-store$(p, x, v)$, by removing the request from the buffer and applying it to main memory. The effect of a buffer-swap$(p, x, v_1)$ is similar to that of a buffer-store$(p, x, v_1)$ combined with a load-request$(p, x)$, and the event memory-swap$(p, x, v_1, v_2?)$ has the same effect of a load-reply$(p, x, v_2?)$ followed by memory-store$(p, x, v_1)$. These events are categorized in Figure 4.3, and will be referred to according to their categories where appropriate. The buffer and load-request events together comprise the request events. That is, $(E, \xrightarrow{\pi})$ is defined on these events only. We specify the $I$ and $R$ components of $M_{TSO}$; the rest of the components of $M_{TSO}$ can be inferred from those.

The function $I$ of $M_{TSO}$ is defined as follows:

- A write invocation is implemented in $M_{TSO}$ by the ordered pair that represents placing the write in the buffer and later committing it to main memory.
  $I((\text{write}, x, (v)), p) = \{ \text{buffer-store}(p, x, v), \text{memory-store}(p, x, v) \}$. 

- A read operation is implemented by the ordered pair that represents requesting the value of a variable and later receiving the value.
  $I((\text{read}, x, \lambda), p) = \{ \text{load-request}(p, x), \text{load-reply}(p, x, v?) \}$. 

- A swap is implemented by the ordered pair that represents placing the swap in the buffer and then accessing main memory.
  $I((\text{swap}, x, (v_1)), p) = \{ \text{buffer-swap}(p, x, v_1), \text{memory-swap}(p, x, v_1, v_2) \}$. 

After a processor requests a load, $v?$ in the load-reply event will be instantiated either
Figure 4.3: Categorizing $E$ of $M_{TSO}$ and $M_{PSO}$

according to the value written by a preceding buffer event or a preceding memory event. This is imposed by $R$. The set $R$ specifies three rules: (1) the buffers are FIFO, (2) loads and swaps are blocking, and (3) how parameter variables are instantiated. Precisely, $R = \{\rho_1, \rho_2, \rho_3\}$, and is defined as follows:

- $\rho_1$: Let $e_1$ and $e_2$ be buffer events by the same processor. Also, let $e'_1$ and $e'_2$ be the matching memory events of $e_1$ and $e_2$, respectively. Then, $e_1 \xrightarrow{\pi} e_2$ if and only if $e'_1 \xrightarrow{\Xi} e'_2$.

- $\rho_2$: Let $e_1$ represent a load-request (respectively, buffer-swap), and $e_2$ represent its matching load-reply (respectively, memory-swap) event. If there is an event $e$ by the same processor such that $e_1 \xrightarrow{\Xi} e \xrightarrow{\Xi} e_2$, then $e$ is necessarily a memory-store event.

- $\rho_3$: A load-reply$(p, x, v?)$ event $e$ instantiates $v?$ as follows. If the most recent buffer-store that precedes $e$ in $\Xi|p|x$ is buffer-store$(p, x, v)$ such that its matching memory-store$(p, x, v)$ follows $e$ in $\Xi|x$, then load-reply$(p, x, v? \leftarrow v)$. Otherwise, load-reply$(p, x, v? \leftarrow v)$ where the most recent memory event that precedes $e$ in $\Xi|x$ is either memory-swap$(q, x, v, \_\_)$ or memory-store$(q, x, v)$.

A memory-swap$(p, x, \_, v?)$ event $e$ instantiates $v?$ as follows. Let memory-store$(q,
x, v) or memory-\text{swap}(q, x, v, -) be the most recent memory event that precedes e in 
\Xi|x, then memory-\text{swap}(p, x, -, v? \leftarrow v).

4.1.3 Operational Partial Store Ordering

The description of \( M_{PSO} \) is very similar to that of \( M_{TSO} \). There are two essential differences:
(1) the buffers’ behavior and (2) the support for a barrier invocation \((\text{barrier}, \lambda, \lambda)\). The request events in \( M_{PSO} \) are the same as those for \( M_{TSO} \), in addition to the store-barrier events.

The machine \( M_{PSO} = (\Pi, \Sigma, E, N, I, R) \) where \( I((\text{read}, x, \lambda), p), I((\text{write}, x, (v)), p) \), and \( I((\text{swap}, x, (v_1)), p) \) are the same as for \( M_{TSO} \), and

- \( I((\text{barrier}, \lambda, \lambda), p) = \{ (\text{store-barrier}(p) ) \} \).
- \( R = \{ \rho_1, \rho_2, \rho_3, \rho_4 \} \), where each \( \rho_i \) is given by the following:
  - \( \rho_1' \): Let \( e_1 \) and \( e_2 \) be buffer events by the same processor on the same location. Also, let \( e_1' \) and \( e_2' \) be the matching memory events of \( e_1 \) and \( e_2 \), respectively. Then \( e_1 \xrightarrow{\pi} e_2 \) if and only if \( e_1' \xrightarrow{\Xi} e_2' \). Note that this is the same as \( \rho_1 \) for \( M_{TSO} \) except that it restricts \( e_1 \) and \( e_2 \) to be on the same location.
  - \( \rho_2 \): Same as \( M_{TSO} \).
  - \( \rho_3 \): Same as \( M_{TSO} \).
  - \( \rho_4 \): Let \( e \) be a store-barrier event by processor \( p \). For any two buffer events \( e_1 \) and \( e_1' \) by \( p \) such that \( e_2 \) and \( e_2' \) are respectively their matching memory events, if \( e_1 \xrightarrow{\pi} e \xrightarrow{\pi} e_1' \) then \( e_2 \xrightarrow{\Xi} e_2' \).

4.2 Non-Operational Description

4.2.1 Earlier Attempt

Kohli et al. attempted to provide a non-operational definition for TSO [47]. Their definition ignores the TSO swap operation. However, considering a swap to be both a read and a write
operation, the definition extends to include swaps naturally. We refer to their definition by TSO-K and we argue that it fails to capture the operational semantics described in Section 4.1.

Kohli Partial Program Order

Define the Kohli partial program order, denoted \( (O, \overset{kpo}{\rightarrow}) \), by \( o_1 \overset{kpo}{\rightarrow} o_2 \) if \( o_1 \overset{prog}{\rightarrow} o_2 \) and one of the following holds:

1. **two-write**: \( o_1, o_2 \in O|w \),
2. **two-reads**: \( o_1, o_2 \in O|r \),
3. **read-before-write**: \( o_1 \in O|r \) and \( o_2 \in O|w \),
4. **same-object**: \( o_1, o_2 \in O|x \), for some \( x \in J \), or
5. **transitivity**: there is an operation \( o' \) such that \( o_1 \overset{kpo}{\rightarrow} o' \overset{kpo}{\rightarrow} o_2 \).

**Definition 4.2.1** Let \( O \) be all the operations of a computation \( C \) of a multiprocess system \( (P,J) \). Then \( C \) is TSO-K if there exists a total order \( (O|w, \overset{\text{writes}}{\rightarrow}) \) and \( \forall p \in P \) there is a linearization \( (O|p \cup O|w, \overset{L_p}{\rightarrow}) \), satisfying:

1. \( (O|p \cup O|w, \overset{kpo}{\rightarrow}) \subseteq (O|p \cup O|w, \overset{L_p}{\rightarrow}) \), and
2. \( (O|w, \overset{L_p}{\rightarrow}) = (O|w, \overset{\text{writes}}{\rightarrow}) \).

Definition 4.2.1 can be rewritten as follows. The proof is straightforward (see that of Lemma 5.2.3 as an example).

**Definition 4.2.2** Let \( O \) be all the operations of a computation \( C \) of a multiprocess system \( (P,J) \). Then \( C \) is TSO-K if there exists a linearization \( (O, \overset{L}{\rightarrow}) \) such that \( (O, \overset{kpo}{\rightarrow}) \subseteq (O, \overset{L}{\rightarrow}) \).

This definition does not capture the TSO semantics. Consider the following computation.

**Computation 7**\[
\begin{align*}
p & : w(x)2 w(x)1 r(x)1 r(y)2 \\
q & : w(y)2 w(y)3 r(y)3 r(x)2
\end{align*}
\]
One possible scenario of this computation on a TSO machine is captured by the following sequence of events:

\[
\text{buffer-store}(p, x, 2), \text{memory-store}(p, x, 2), \text{buffer-store}(q, y, 2), \\
\text{memory-store}(q, y, 2), \text{buffer-store}(p, x, 1), \text{buffer-store}(q, y, 3), \\
\text{load-request}(p, x), \text{load-repy}(p, x, 1), \text{load-request}(q, y), \\
\text{load-repy}(q, y, 3), \text{load-request}(p, y), \text{load-repy}(p, y, 2), \\
\text{load-request}(q, x) \text{ load-repy}(q, x, 2), \text{memory-store}(p, x, 1), \\
\text{memory-store}(q, y, 3).
\]

Observe that in the above scenario, the events \(\text{load-repy}(p, x, 1)\) and \(\text{load-repy}(q, y, 3)\) return values from the buffers, while \(\text{load-repy}(p, y, 2)\) and \(\text{load-repy}(q, x, 2)\) access main memory. However, Computation 7 does not satisfy TSO-K. First of all, note that in Computation 7, \((O, \xrightarrow{kpo}) = (O, \xrightarrow{prog})\). To construct the linearization \((O|p \cup O|w; \xrightarrow{Lp})\), we must have \(r_p(y)2 \xrightarrow{Lp} w_q(y)3\) for validity. Since transitivity implies \(w_p(x)1 \xrightarrow{kpo} r_p(y)2\), we must have \(w_p(x)1 \xrightarrow{Lp} w_q(y)3\). When constructing \((O|q \cup O|w; \xrightarrow{Lq})\), we must maintain \(r_q(x)2 \xrightarrow{Lq} w_p(x)1\), but since \(w_q(y)3 \xrightarrow{kpo} r_q(x)2\), we must have \(w_q(y)3 \xrightarrow{Lq} w_p(x)1\). This violates Condition 2 of Definition 4.2.1, which requires a mutual agreement between \(p\) and \(q\) on the order of the writes.

While TSO-K does not capture the the \(M_{TSO}\) semantics, we shall see later that TSO-K captures something stronger.

### 4.2.2 Non-Operational Total Store Ordering

A *domestic* read operation by process \(p\) returns the value written by a write operation also by process \(p\); otherwise, the read is called *foreign*. Also, swap operations are defined to be foreign reads. Recall that a swap is both a read and a write.
TSO Partial Program Order

Define the TSO partial program order, denoted \((O, \overset{tso}{\rightarrow})\), by \(o_1 \overset{tso}{\rightarrow} o_2\) if \(o_1^{\text{prog}} \overset{\text{prog}}{\rightarrow} o_2\) and one of the following holds:

1. **Two-writes**: \(o_1, o_2 \in O|w,\)
2. **Two-reads**: \(o_1, o_2 \in O|r\) and \(o_1\) is foreign,
3. **Read-before-write**: \(o_1 \overset{r}{\rightarrow} o_2 \overset{w}{\rightarrow} \),
4. **Same-object**: \(o_1, o_2 \overset{x}{\rightarrow} O|\), for some \(x \in J\), or
5. **Transitivity**: there is an operation \(o'\) such that \(o_1 \overset{tso}{\rightarrow} o' \overset{tso}{\rightarrow} o_2\).

**Definition 4.2.3** Let \(O\) be all the operations of a computation \(C\) of a multiprocess system \((P, J)\). Then \(C\) is TSO if there exists a linearization \((O, \overset{L}{\rightarrow})\) such that \((O, \overset{tso}{\rightarrow}) \subseteq (O, \overset{L}{\rightarrow})\).

It follows from the definitions of \((O, \overset{tso}{\rightarrow})\) and \((O, \overset{kpo}{\rightarrow})\) that TSO-K is strictly stronger than TSO. The partial order \((O, \overset{kpo}{\rightarrow})\) defined by Kohli et al.[47] proved too strong. It requires program order to be maintained between any two read operations. In Computation 7, we have seen that a read-before-read program order must be relaxed. Namely, we had to relax either \(r_q(y)3^{\text{prog}} \overset{\text{prog}}{\rightarrow} r_q(x)2\) or \(r_p(x)1^{\text{prog}} \overset{\text{prog}}{\rightarrow} r_p(y)2\). Notice that \(r_p(x)1\) and \(r_q(y)3\) are domestic reads and \(r_q(x)2\) and \(r_p(y)2\) are foreign. For this reason, \((O, \overset{tso}{\rightarrow})\) relaxes the domestic read to foreign read program order. Computation 7 satisfies TSO as shown by the following linearization:

\[
(O, \overset{L}{\rightarrow}) = \langle w_p(x)2, r_q(x)2, w_p(x)1, r_p(x)1, w_q(y)2, r_p(y)2, w_q(y)3, r_q(y)3 \rangle
\]

Another observation is that program order between two domestic reads must be also relaxed, although this may seem counter-intuitive. Consider the following computation where both \(p\)'s reads are domestic.

**Computation 8**

\[
\begin{align*}
p &: w(x)1 \quad w(y)1 \quad r(y)1 \quad r(x)1 \\
qu &: r(x)1 \quad w(x)2 \quad w(y)2 \quad r(y)1
\end{align*}
\]
Computation 8 is a possible TSO computation as shown by the following sequence of events:

\[ \text{buffer-store}(p, x, 1), \text{buffer-store}(p, y, 1), \text{load-request}(p, y), \]
\[ \text{load-reply}(p, y, 1), \text{load-request}(p, x), \text{load-reply}(p, x, 1), \]
\[ \text{memory-store}(p, x, 1), \text{load-request}(q, x), \text{load-reply}(q, x, 1), \]
\[ \text{buffer-store}(q, x, 2), \text{memory-store}(q, x, 2), \text{buffer-store}(q, y, 2), \]
\[ \text{memory-store}(q, y, 2), \text{memory-store}(p, y, 1), \text{load-request}(q, y), \]
\[ \text{load-reply}(q, y, 1). \]

Note again that \((O, \overset{kpo}{\rightarrow}) = (O, \overset{prog}{\rightarrow})\) in Computation 8, and a linearization that maintains program order cannot be found because the following contains a cycle:

\[ w_p(x) 1 \overset{L}{\rightarrow} w_p(y) 1 \overset{L}{\rightarrow} r_p(y) 1 \overset{L}{\rightarrow} w_p(x) 1 \overset{L}{\rightarrow} w_q(y) 2 \overset{L}{\rightarrow} w_p(y) 1 \overset{L}{\rightarrow} r_p(y) 1 \]

Nevertheless, the desired TSO linearization is as follows:

\[ (O, \overset{L}{\rightarrow}) = \{ w_p(x) 1, r_p(x) 1, r_q(x) 1, w_q(x) 2, w_q(y) 2, w_p(y) 1, r_p(y) 1, r_q(y) 1 \} \]

Finally, note that \((O, \overset{tso}{\rightarrow})\) maintains the program order of a foreign read before a domestic read. For example, Computation 9 in Section 4.4 will be shown to be PSO but not TSO. If the foreign read to domestic read program order is relaxed, then this computation would satisfy this new relaxed order.

**Theorem 4.2.4** \(M_{TSO}\) exactly implements TSO.

The next two lemmas establish Theorem 4.2.4. Let \((P, J)\) be a multiprocess system with operation invocations in \{\text{\textit{read}, \textit{read}}, \text{\textit{write}, \textit{write}}, \text{\textit{swap}, \textit{swap}}\}. Let \(\Xi\) be a run of \((P, J)\) on \(M_{TSO}\). That is, \(\Xi \in M^2_{TSO}(P, J)\). If a read \(r\) returns the value written by a write \(w\), \(r\) and \(w\) are said to be \textit{causally-matching}.

**Lemma 4.2.5** \(M_{TSO}\) provides TSO.
4.2. Non-Operational Description

Proof: Let $C_\Xi$ be the computation of $(P, J)$ induced by $\Xi$, and $O$ be all the operations that result from $C_\Xi$. We show that $C_\Xi$ satisfies TSO.

Construct a sequence of operations $X$ from $\Xi$ as follows.

Stage 1: Initially, $X$ is empty. Iterate through $\Xi$ considering one event $e$ at a time. If $e$ is a load-request, buffer-store, or buffer-swap, ignore it. If $e$ is a load-reply ($-, x, v$), memory-store ($-, x, \lambda$), or a memory-swap ($-, x, v_1, v_2$) event, then respectively append to $X$ a $(read, x, \lambda, (v))$, $(write, x, (v), \lambda)$, or $(swap, x, (v_1), (v_2))$ operation.

For a read that completed at main memory, the corresponding load-reply instantiates its parameter variable according to a preceding memory-store. So, the construction of $X$ at Stage 1 guarantees that such a read (domestic or foreign) follows its causally-related write. This does not hold for a read that corresponds to a load-reply that completed at the buffer level. At this point, such a read precedes its causally matching write in $X$, violating both validity and TSO. It violates TSO because its causally matching write is necessarily applied to the same variable as the read. Thus, the program order of such a write followed by that read must be maintained in TSO (same-object condition). So, the next stage adjusts $X$ so that the resulting sequence is valid and extends TSO.

Stage 2: Now, we adjust $X$ by moving the domestic reads that violate validity. Initially, mark every domestic read operation in $X$ as unvisited. Iterate through $X$ examining each unvisited domestic read operation $o$ in turn. Let $e$ be $o$’s corresponding load-reply event in $\Xi$. Let $e'$ be the memory event in $\Xi$ such that $e$ returns the value written by $e'$. By construction of $X$, $e'$ has a corresponding write operation $o'$ in $X$. If $o'$ precedes $o$ in $X$, then mark $o$ as visited and continue with the first unvisited read in $X$. If $o'$ follows $o$ in $X$, then move $o$ in $X$ such that it immediately follows the latest of $o'$ or the last moved read event. Mark $o$ as visited, and continue with the first unvisited read event in $X$.

Let the adjusted $X$ be $\hat{X}$. Define $(O, \longrightarrow)$ to be exactly $\hat{X}$. Now, we show that $(O, \longrightarrow)$ is a linearization that extends $(O, \longrightarrow)$. 

- $(O, \longrightarrow)$ is a linearization:

  In the construction of $\hat{X}$ from $X$, we only had to move reads that violate valid-
ity. Furthermore, each moved domestic \((\text{read}, x, \lambda, (v))\) has been inserted after a \((\text{write}, x, (v), \lambda)\) or \((\text{swap}, x, (v), (v'))\) without any intervening write or swap operations. The reads that were not moved do not violate validity because each such read or swap operation in \(\hat{X}\) (with a corresponding load-reply or memory-swap event in \(\Xi\)) returns the value written to the same variable in main memory by the most recent write or swap in \(\hat{X}\) (with a corresponding memory-store or memory-swap event in \(\Xi\)). Therefore, \((O, \xrightarrow{L})\) is a linearization.

- \((O, \xrightarrow{\text{tso}}) \subseteq (O, \xrightarrow{L})\):

  We first show that the construction of \(X\) at Stage 1 satisfies \(\xrightarrow{\text{tso}}\) except for some domestic reads. Given two operations \(o_1\) and \(o_2\) by some process \(p\) such that \(\langle e'_1, e_1 \rangle \in I(\text{invoc}(o_1), p)\) and \(\langle e'_2, e_2 \rangle \in I(\text{invoc}(o_2), p)\), we prove that if \(o_1 \xrightarrow{\text{tso}} o_2\), then \(e_1 \xrightarrow{\Xi} e_2\) and, hence, \(o_1\) precedes \(o_2\) in \(X\).

1. **two-writes**: If \(o_1\) and \(o_2\) are write operations, then, \(e_1\) and \(e_2\) are memory events (store or swap, whichever applies). Since \(o_1 \xrightarrow{\text{prog}} o_2\), \(e'_1 \xrightarrow{\pi} e'_2\). Rule \(\rho_1\) of \(M_{\text{TSO}}\) implies that \(e_1 \xrightarrow{\Xi} e_2\). Therefore, \(o_1\) precedes \(o_2\) in \(X\) by construction.

2. **read-before-write**: If \(o_1\) and \(o_2\) are respectively read and write operations, then, \(e_1\) is a load-reply and \(e_2\) is a memory-store. Since \(o_1 \xrightarrow{\text{prog}} o_2\), \(e'_1 \xrightarrow{\pi} e'_2\). By \(\rho_2\), \(e'_2\) cannot intervene between \(e'_1\) and \(e_1\). Thus \(e'_1 \xrightarrow{\Xi} e_1 \xrightarrow{\Xi} e'_2 \xrightarrow{\Xi} e_2\), and therefore, \(o_1\) precedes \(o_2\) in \(X\) by construction.

   If \(o_1\) is a read and \(o_2\) is a swap, then \(e_1\) is a load-reply and \(e_2\) is a memory-swap. Also by \(\rho_2\), \(e'_1 \xrightarrow{\Xi} e_1 \xrightarrow{\Xi} e'_2 \xrightarrow{\Xi} e_2\). Therefore, \(o_1 \xrightarrow{L} o_2\) by construction. The cases where \(o_1\) and \(o_2\) are both swaps or \(o_1\) is a swap and \(o_2\) is a write are exactly the same as the two-writes case.

3. **two-reads**: If \(o_1\) and \(o_2\) are both read operations, then \(e_1\) and \(e_2\) are both load-reply events. Since \(o_1 \xrightarrow{\text{prog}} o_2\), \(e'_1 \xrightarrow{\pi} e'_2\). \(\rho_2\) implies that \(e'_1 \xrightarrow{\Xi} e_1 \xrightarrow{\Xi} e'_2 \xrightarrow{\Xi} e_2\). Therefore, \(o_1\) precedes \(o_2\) in \(X\) by construction.

   The cases where \(o_1\) and/or \(o_2\) is a swap are also straightforward.
4. same-object: If \( o_1 \) is a write and \( o_2 \) is a read applied to the same object \( x \), then, \( e_1 \) is a memory-store and \( e_2 \) is a load-reply. Since \( o_1^{prov} o_2 \), \( e_1 \rightarrow e_2 \). If \( e_2 \) instantiates the parameter \( v \) according to \( e_1 \), then it follows from \( \rho_3 \) that \( e_1 \rightarrow e_2 \). Otherwise, \( e_2 \) instantiates the parameter \( v \) according to \( e_1 \). It follows that \( e_2 \rightarrow e_1 \) and in this case \( o_2 \) precedes \( o_1 \) in the construction of \( X \) at Stage 1, and \( o_2 \) is a domestic read.

Therefore only some of the domestic reads are violating \( TSO \) after Stage 1 (in \( \hat{X} \)). These are exactly the reads that are moved forward in \( \hat{X} \) (Stage 2). So, we show that this movement does not violate any conditions and restores \( TSO \).

Let \( o_1 \) be a moved read of \( x \), such that it bypassed \( o_2 \) where \( o_1 \rightarrow o_2 \). Also let \( \langle e_1', e_1 \rangle \in I(invoc(o_1), p) \) for some \( p \), and \( \langle e_2', e_2 \rangle \in I(invoc(o_2), p) \). Recall that \( o_1 \) was moved because \( e_1 \) falls between a buffer-store \( e_3' \) and a memory-store \( e_3 \) to the same location as \( e_1 \). Rule \( \rho_2 \) implies that \( e_1' \) also falls between \( e_3' \) and \( e_3 \). We argue that \( o_2 \) must be a read of \( y \neq x \). If \( o_2 \) were a write, then for \( o_1 \) to bypass \( o_2 \), we must have \( e_2 \) intervening between \( e_1 \) and \( e_3 \). However, rule \( \rho_1 \) indicates that \( e_2' \rightarrow e_3' \) because \( e_2 \rightarrow e_3 \). That is, \( e_2' \rightarrow e_1' \) and consequently, \( e_2' \rightarrow e_1 \). This means that \( o_2 \rightarrow o_1 \), a contradiction. Finally, if \( o_1 \) and \( o_2 \) are both reads of the same object, then both must have been moved without violating \( TSO \) by construction. The case where both \( o_1 \) and \( o_2 \) are reads does not need a special treatment. Since \( o_1 \) is a domestic read and \( o_2 \) is a read of a different variable from \( o_1 \), \( TSO \) does not require program order to be maintained between \( o_1 \) and \( o_2 \).

\[\text{Lemma 4.2.6} \ M_{TSO} \text{ realizes } TSO.\]

**Proof:** Let \( C \) be any TSO computation of any \((P, J)\) system. We show how to construct a run \( \Xi \in M_{TSO}^\Xi(P, J) \) such that \( C_\Xi = C \). Let \( O \) be all the operations resulting from \( C \).

Given the linearization \( (O, \rightarrow_L) \) guaranteed by Definition 4.2.3, construct a sequence \( X \) of operations as follows. Initially \( X \) is empty. For each process \( p \) computation of \( C \), maintain a pointer \( \downarrow_p \) that initially points at the first operation in \( p \)'s computation. If \( \downarrow_p \) points at \( o \), then we say \( \downarrow_p = o \). Similarly, we maintain the pointer \( \downarrow_L \) to operations in
Initially, $\downarrow_L$ points at the first operation in $(O, \rightarrow_L)$. When there are no more operations to consider in a sequence $s$, we say $\downarrow_s = \bot$. Also, advancing $\downarrow_s$ means the pointer is incremented to point at the next operation in $s$. Initially, all operations are unmarked.

Repeat until $\downarrow_L = \bot$:

1. If $\downarrow_L = o \in O|p$, for some $p$, and $\downarrow_p = o$, then append $o$ to $X$. Advance both $\downarrow_L$ and $\downarrow_p$.

2. If $\downarrow_L = o \in O|p$ and $o$ is unmarked but $\downarrow_p = o' \neq o$, then append $o'$ to $X$. If $o'$ is a write mark the occurrence of $o'$ in $(O, \rightarrow_L)$ as $\hat{o'}$. Advance $\downarrow_p$ only.

3. If $\downarrow_L = o$ and $o$ is marked ($\hat{o}$), then append $\hat{o}$ to $X$. Advance $\downarrow_L$.

Notice that some writes are duplicated in $X$ with the unmarked copy preceding the marked one. At the end (when $\downarrow_L = \bot$), if there are write operations that have not been duplicated in $X$, then for each such write $o$ insert a marked (“hatted”) copy $\hat{o}$ immediately after $o$.

Now, we construct $\Xi$ as follows. Iterating through $X$, consider each operation $o$. If $o$ is a $(\text{read}, x, \lambda, (v))$ or $(\text{swap}, x, (v_1), (v_2))$ by $p$, then append to $\Xi$ load-request $(p, x)$ immediately followed by load-reply $(p, x, v)$ or buffer-swap $(p, x, v_1)$ immediately followed by a memory-swap $(p, x, v_1, v_2)$, respectively. If $o$ is an unmarked $(\text{write}, x, (v), \lambda)$ by $p$, append to $\Xi$ buffer-store $(p, x, v)$. If $o$ is marked, then append to $\Xi$ memory-store $(p, x, v)$.

1. Complying with $I$:

$\Xi$ complies with $I$ by construction. Each load-request event precedes its matching load-reply event, and each buffer-swap precedes its matching memory-swap event. Furthermore, the construction guarantees that each “hatted” write operation follows its unmarked copy in $X$. Therefore, each buffer-store precedes its matching memory-store in $\Xi$.

2. Complying with program order:

It follows from the construction that if $o_1 \xrightarrow{\text{prog}} o_2$ such that $\langle e_1, e'_1 \rangle \in I(\text{invoc}(o_1), p)$ and $\langle e_2, e'_2 \rangle \in I(\text{invoc}(o_2), p)$, then $e_1 \xrightarrow{\Xi} e_2$. 
3. **Complying with R:**

To show that $\Xi$ satisfies $\rho_1$, let $o_1$ and $o_2$ be two operations in $O|w|p$ such that $\langle e_1, e'_1 \rangle \in I(\text{invoc}(o_1), p)$ and $\langle e_2, e'_2 \rangle \in I(\text{invoc}(o_2), p)$. If $e_1 \overset{\pi}{\rightarrow} e_2$, then $o_1 \overset{\text{prog}}{\rightarrow} o_2$. Therefore, $e'_1 \overset{\pi}{\rightarrow} e'_2$ by construction. If $e'_1 \overset{\pi}{\rightarrow} e'_2$, then $o_1 \overset{\text{prog}}{\rightarrow} o_2$ because $o_1 \overset{\text{tso}}{\rightarrow} o_2$. Therefore, $e_1 \overset{\pi}{\rightarrow} e_2$.

$\Xi$ satisfies $\rho_2$ because the construction guarantees that there are no intervening events between a matching load-request, load-reply pair. Also, there are no intervening events between a matching buffer-swap, memory-swap pair of events.

Finally, we show that $\Xi$ satisfies $\rho_3$. Since $(O, \overset{L}{\rightarrow})$ is a linearization, for any $x \in J$ each operation $o \in O|x|\tau$ returns the value written by the most recent operation $o' \in O|x|w$ that precedes $o$ in $(O|x, \overset{L}{\rightarrow})$. Let $\langle e_1, e_2 \rangle \in I(\text{invoc}(o), p)$ and $\langle e'_1, e'_2 \rangle \in I(\text{invoc}(o'), p)$.

If both $o$ and $o'$ are by the same process, then $o' \overset{\text{tso}}{\rightarrow} o$ and, thus, $o' \overset{\text{prog}}{\rightarrow} o$. The construction guarantees that $e'_1$ is the most recent buffer-store by the same process that precedes $e_2$. Otherwise ($o$ and $o'$ are by different processes), the most recent event that precedes $e_2$ in $\Xi|x$ is $e'_2$. Since $o'$ is the most recent operation in $O|x|w$ that precedes $o$ in $(O|x, \overset{L}{\rightarrow})$, then $e'_2$ is the most recent memory-store event that precedes $e_2$ in $\Xi$ by construction. Moreover, there could not be any intervening buffer-store events by the same process between $e'_2$ and $e_2$. To see this, assume to the contrary that a buffer-store $e''$ to $x$ intervenes between $e'_2$ and $e_2$. This means that $o' \overset{\text{prog}}{\rightarrow} o \overset{\text{pro}}{\rightarrow} o$ where $e''$ corresponds to $e''$. However, $o$ returns the value written by $o'$, a contradiction.

The argument for memory-swaps is also straightforward.
Non-Terminating Computations

The proof of Theorem 4.2.4 extends to the non-terminating case naturally. For the provides direction, when considering a prefix $\Xi$ of a non-terminating run, not all events in $\Xi$ correspond to completed implementations. For instance, $\Xi$ might include a buffer-store event without its matching memory-store event. The operations corresponding to such incomplete implementations are not included in $C_{\Xi}$, the computation induced by the prefix $\Xi$. Also, $C_{\Xi}$ does not include the read operations that are causally related to a write that has been excluded from $C_{\Xi}$. As $\Xi$ extends to a longer prefix, the same construction procedure described in Lemma 4.2.6 is used. The extension to non-terminating computations in the realizes direction is trivial.

A corollary follows after noting that TSO-K is strictly stronger than TSO, because $(O, \stackrel{tsop}{\rightarrow}) \subseteq (O, \stackrel{kpos}{\rightarrow})$.

**Corollary 4.2.7** $MTSO$ realizes TSO-$K$.

### 4.2.3 Non-Operational Partial Store Ordering

PSO is defined similarly to TSO. The only difference is that two writes necessarily maintain their program order only if they are to the same object or they are separated by a barrier operation. Let $O|b$ denote all the barrier operations in $O$.

**PSO Partial Program Order**

Define the PSO *partial program order*, denoted $(O, \stackrel{psop}{\rightarrow})$, by $o_1 \stackrel{psop}{\rightarrow} o_2$ if $o_1 \stackrel{prog}{\rightarrow} o_2$ and one of the following holds:

1. **two-writes** $o_1 \in O|b$ and $o_2 \in O|w$, or $o_1 \in O|w$ and $o_2 \in O|b$,
2. **two-reads**: $o_1, o_2 \in O|r$ and $o_1$ is foreign,
3. **read-before-write**: $o_1 \in O|r$ and $o_2 \in O|w$ and $o_1$ is foreign,
4. **same-object**: $o_1, o_2 \in O|x$, for some $x \in J$, or
5. **transitivity**: there is an operation $o'$ such that $o_1 \stackrel{psop}{\rightarrow} o' \stackrel{psop}{\rightarrow} o_2$. 
Definition 4.2.8 Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P, J)$. Then $C$ is PSO if there exists a linearization $(O, \xrightarrow{L})$ such that $(O, \xrightarrow{\text{pso}}) \subseteq (O, \xrightarrow{L})$.

The proof of the following theorem is very similar to that of Theorem 4.2.4. As a matter of fact, the construction procedures used for both of lemmas 4.2.5 and 4.2.6 apply here.

Theorem 4.2.9 $M_{PSO}$ exactly implements PSO.

4.2.4 Earlier Complex Definitions

In this section, we give earlier definitions for TSO and PSO that we have used in some earlier papers [41, 37, 38, 39]. The complexity of these definitions makes them less useful to programmers than definitions 4.2.3 and 4.2.8. Also, they could prove difficult to use with an automated verifier. Whether they are truly non-operational is also questionable.

Besides its own events, each processor $p$ can “see” only another processor $q$’s memory events. A write by $p$ is visible to $p$ (as a buffer event) before it is visible to other processors. If for some buffer-store$(q, x, v’)$ with a matching memory-store$(q, x, v’)$, buffer-store$(p, x, v) \xrightarrow{\Xi} \text{memory-store}(q, x, v’)$, then the write of $v’$ to $x$ by $q$ is invisible to $p$. This is because $R$ forces a load by $p$ on $x$ to return a value from $p$’s buffer as long as a buffer-store by $p$ to $x$ is still pending. Furthermore, when memory-store$(p, x, v)$ is applied, this memory-store overwrites memory-store$(q, x, v’)$.

The definitions presented in this section distinguish between writes that are only visible to the writer (corresponding to values that are temporarily stored in the writer’s buffer) and writes that are visible to other processes (corresponding to values stored in main memory). Let $A \uplus B$ denote the disjoint union of $A$ and $B$, and if $x \in A \cap B$ then the copy of $x$ in $A$ is denoted $x_A$ and the copy of $x$ in $B$ is denoted $x_B$. The copies $x_A$ and $x_B$ are said to be matching. The disjoint union $(O|p \uplus O|w)$ duplicates the write operations by process $p$ where the copy in $O|p$, $w_{O|p}$, represents a write that is only visible to $p$, and the copy in $O|w$, $w_{O|w}$, represents the matching write that might be visible to other processes. The following definition requires a total order on all the writes and swaps $(O|w, \xrightarrow{\text{writes}})$, called the writes order. A process’s view is defined by the total order $((O|p \uplus O|w), \xrightarrow{\text{merge}})$, which
obeys \( p \)'s program order, the writes order, in addition to the fact that a copy of a write in \( O|p \) precedes its matching copy in \( O|w \). In this way, the set of invisible write operations to a process \( p \) can be defined as:

\[
O_{\text{inv}p} = \{ w \mid \exists x \in J, w \in (O|x|w\backslash O|p), \land \exists w' \in O|x|p|w \land w'_p \xrightarrow{\text{merge}_p} w \xrightarrow{\text{merge}_p} w'|w \}
\]

Let \( O|s \) denote all the swap operations. By convention, \( O|s \subseteq O|w \). Similarly, the set of \( p \)'s memory writes can be defined as follows

\[
O_{\text{mw}p} = \{ w|O|w \mid w \in O|p|w\backslash O|s \}
\]

**Definition 4.2.10** Let \( O \) be all the operations of a computation \( C \) of a multiprocess system \( P,J \). Then \( C \) is TSO\(_{\text{semi}} \) if there exists a total order \( (O|w, \xrightarrow{\text{writes}}) \) such that \( (O|w, \xrightarrow{\text{prog}}) \subseteq (O|w, \xrightarrow{\text{writes}}) \) and \( \forall p \in P \) there is a total order \( (O|p \uplus O|w, \xrightarrow{\text{merge}_p}) \), satisfying:

1. \( (O|p, \xrightarrow{\text{prog}}) = (O|p, \xrightarrow{\text{merge}_p}) \),
2. \( (O|w, \xrightarrow{\text{writes}}) = (O|w, \xrightarrow{\text{merge}_p}) \),
3. if \( w \in (O|p|w) \) then \( w|O|w \xrightarrow{\text{merge}_p} w|O|w \),
4. \(((O|p \uplus O|w) \setminus (O_{\text{inv}p} \cup O_{\text{mw}p})) > (\text{merge}_p) \) is a linearization, where
   \[
   O_{\text{inv}p} = \{ w \mid \exists x \in J, w \in (O|x|w\backslash O|p), \land \exists w' \in (O|x|p|w) \land w'_p \xrightarrow{\text{merge}_p} w \xrightarrow{\text{merge}_p} w'|w \}
   \]
   \[
   O_{\text{mw}p} = \{ w|O|w \mid w \in (O|p|w\backslash O|s) \},
   \]
5. let \( w \in (O|p|w) \) and \( s \in (O|p|s) \), if \( s \xrightarrow{\text{prog}} w \), then \( s|O|w \xrightarrow{\text{merge}_p} w|O|p \) and
6. if \( s \in (O|p|s) \), then there is no operation \( o \in O|p \) such that \( s|O|p \xrightarrow{\text{merge}_p} o \xrightarrow{\text{merge}_p} s|O|w \).

The proof of the following theorem is elsewhere [46].

**Theorem 4.2.11** \( M_{\text{TSO}} \) exactly implements TSO\(_{\text{semi}} \).

The following corollary is immediate from theorems 4.2.4 and 4.2.11

**Corollary 4.2.12** TSO is equivalent to TSO\(_{\text{semi}} \).

\( \text{PSO}_{\text{semi}} \) can be defined in a similar way.
**Definition 4.2.13** Let $O$ be all the operations resulting from a computation $C$ of a multi-process system $(P, J)$. Then $C$ is PSO semi if there exists a total order $(O|w, \overset{\text{writes}}{\longrightarrow})$ such that \( \forall x, (O|w|x, \overset{\text{writes}}{\longrightarrow}) \subseteq (O|w|x, \overset{\text{writes}}{\longrightarrow}) \) and \( \forall p \in P \) there is a total order $(O|p \uplus O|w, \overset{\text{merge}_{p}}{\longrightarrow})$, satisfying items 1 through 6 of TSO semi and

7. if \( b \in (O|p|b) \) and \( w, u \in (O|p|w) \) and \( w \overset{\text{prog}}{\rightarrow} b \) and \( b \overset{\text{prog}}{\rightarrow} u \), then \( w \overset{\text{merge}_{p}}{\rightarrow} uO|w \).

**Theorem 4.2.14** $M_{PSO}$ exactly implements PSO semi.

**Corollary 4.2.15** PSO is equivalent to PSO semi.

Even though the definitions of TSO semi and PSO semi are correct, they are not truly non-operational and are complex. The disjoint union $(O|p \uplus O|w)$ duplicates the write operations to mimic buffer-stores and memory-stores, and for this reason its claimed non-operational characteristic is questionable. The need to identify $O_{inv_p}$ resulted in resorting to this joint union. However, it turned out, as we have seen in Section 4.2, that dealing with this kind of invisibility (transient invisibility) can be circumvented by defining the appropriate relaxed program orders, namely $(O, \overset{\text{tso}}{\rightarrow})$ and $(O, \overset{\text{pso}}{\rightarrow})$.

### 4.3 Constructing SC from TSO and PSO

Any TSO computation is also PSO. In fact, TSO is a special case of PSO where the buffer is always FIFO. Moreover, if every two write invocations in a program are separated by a barrier in PSO, then the execution of the altered program on PSO will have the same outcome of the original program on TSO.

**Observation 4.3.1** Let $\mathcal{P}$ denote a program, and let $\mathcal{P}'$ denote $\mathcal{P}$ altered by separating every two write invocations with a barrier invocation. The outcome of $\mathcal{P}'$ on PSO is exactly the same as the outcome of $\mathcal{P}$ on TSO.

TSO with the swap invocation can be made to guarantee SC. In fact, if all writes in a given program are replaced by swaps, this results in behaving as if the buffers do not exist. Call a write invocation *update-equivalent* to a swap invocation if both write the same value to the same variable.
Observation 4.3.2 Let $P$ denote a program, and let $P'$ denote $P$ altered by replacing every write invocation with an update-equivalent swap invocation. The outcome of $P'$ on TSO is exactly the same as the outcome of $P$ on SC.

Finally, the above two observations imply the following:

Observation 4.3.3 Let $P$ denote a program, and let $P'$ denote $P$ altered by replacing every write invocation with an update-equivalent swap invocation, and by separating every pair of swap invocations by a barrier invocation. The outcome of $P'$ on PSO is exactly the same as the outcome of $P$ on SC.

4.4 Comparisons

To compare TSO and PSO with other models, we ignore the swap and barrier operations because the other pure models only include read and write operations. Furthermore, hybrid models include further kinds of objects that may not be comparable with the SPARC objects. Use $TSO_{base}$ and $PSO_{base}$ to refer to TSO and PSO without the swap and barrier operations.

Clearly any $TSO_{base}$ computation is also $PSO_{base}$. However, not all $PSO_{base}$ computations are $TSO_{base}$. From the definitions of $(O, tso)$ and $(O, pso)$, we can see that the first orders every two writes by the same process, while writes can only be ordered by the latter if these are to the same object.

Consider the following $PSO_{base}$ computation, where $w_p(x) 1$ was buffered before $w_p(y) 2$; however, $w_p(y) 2$ was committed to main memory before $w_p(x) 1$. This is not permitted by $TSO_{base}$.

**Computation 9**

\[
\begin{align*}
p &: w(x) 3 \ w(x) 1 \ w(y) 2 \\
q &: r(y) 2 \ r(x) 3
\end{align*}
\]

Computation 9 is $PSO_{base}$ as confirmed by the following linearization:

\[ (O, \rightarrow_L) = \{ w_p(y) 2, \ r_q(y) 2, \ w_p(x) 3, \ r_q(x) 3, \ w_p(x) 1 \} \]
Because of the FIFO store buffer, TSO$_{\text{base}}$ requires program order to be maintained on all writes. In Computation 9, $(O, \xrightarrow{\text{tso}}) = (O, \xrightarrow{\text{prog}})$ because $p$’s computation consists of writes only and $q$’s computation consists of foreign reads. A linearization that adheres to Definition 4.2.3 does not exist because of the following cycle:

$$r_q(x) \xrightarrow{L} w_p(x) 1 \xrightarrow{L} w_p(y) 2 \xrightarrow{L} r_q(y) 2 \xrightarrow{L} r_q(x) 3$$

**Observation 4.4.1** TSO$_{\text{base}}$ is strictly stronger than PSO$_{\text{base}}$.

**Theorem 4.4.2** PSO$_{\text{base}}$ and TSO$_{\text{base}}$ are each incomparable with each of P-RAM, PC-G, and CC.

Each direction of Theorem 4.4.2 is established with a separate lemma.

**Lemma 4.4.3** PSO$_{\text{base}}$ and TSO$_{\text{base}}$ are each not stronger than each of P-RAM, PC-G, and CC.

**Proof:** We show that Computation 10 is PC-G and CC but not PSO$_{\text{base}}$.

**Computation 10**

$$\begin{align*}
p &: w(x) 1 \\
q &: w(y) 1 \\
s &: w(x) 2 \ r(x) 1 \ r(y) 2 \\
t &: w(y) 2 \ r(y) 1 \ r(x) 2
\end{align*}$$

The PC-G linearizations are given by

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle w_s(x) 2, w_p(x) 1, w_r(y) 2, w_q(y) 1 \rangle$$

$$(O|r \cup O|w, \xrightarrow{L_q}) = \langle w_s(x) 2, w_p(x) 1, r_s(x) 1, r_q(y) 1 \rangle$$

$$(O|s \cup O|w, \xrightarrow{L_s}) = \langle w_s(x) 2, w_t(y) 2, w_q(y) 1 \rangle$$

Each linearization maintains the order $\langle w_s(x) 2, w_p(x) 1 \rangle$ on the writes to $x$ and the order $\langle w_t(y) 2, w_q(y) 1 \rangle$ on the writes to $y$, establishing Condition 2 of Definition 2.2.5. The same linearizations are also CC satisfying Definition 2.2.6.

Since the computation is PC-G, it must be also P-RAM. However, Computation 10 is not PSO$_{\text{base}}$. 
Informally, \( r_s(x) 1 \) must have accessed main memory. Since \( r_s(x) 1 \) is preceded by \( w_s(x) 2 \) in \( r \)'s program, it must be the case that \( w_s(x) 2 \) was not pending in the store-buffer when \( r_s(x) 1 \) is performed. This leaves \( s \) with an empty buffer, and means that \( w_p(x) 1 \) overwrites \( w_s(x) 2 \) before \( r_s(x) 1 \) is performed. Similarly, \( r_t(y) 1 \) accesses main memory and \( w_q(y) 1 \) overwrites \( w_t(y) 2 \). Since \( r_s(y) 2 \) also accesses main memory, \( r_s(y) 2 \) must have been performed before \( r_t(y) 1 \). For similar reasons, \( r_s(x) 1 \) must have been performed before \( r_t(x) 2 \). That is, \( w_p(x) 1 \) overwrites \( w_s(x) 2 \) before \( s \) performs \( r_t(x) 2 \) which is impossible.

Precisely, \( (O, \stackrel{pso}{\longrightarrow}) = (O, \stackrel{prog}{\longrightarrow}) \) in Computation 10. This is because \( w_s(x) 2 \stackrel{pso}{\rightarrow} r_s(x) 1 \) (same object), and \( r_s(x) 1 \stackrel{pso}{\rightarrow} r_s(y) 2 \) (\( r_s(x) 1 \) is foreign). Similar reasoning applies to \( t \).

So for this computation to be PSO\textsubscript{base} it must be SC. A linearization that satisfies SC is impossible because of the following. Operation \( w_q(y) 1 \) (respectively, \( w_p(x) 1 \)) must intervene between \( w_t(y) 2 \) (respectively, \( w_s(x) 2 \)) and \( r_t(y) 1 \) (respectively, \( r_s(x) 1 \)) for \( (O, \stackrel{L}{\longrightarrow}) \) to be valid. Furthermore, \( r_s(y) 2 \) (respectively, \( r_t(x) 2 \)) must precede \( w_q(y) 1 \) (respectively, \( w_p(x) 1 \)) because \( w_q(y) 1 \) (respectively, \( w_p(x) 1 \)) follows \( w_t(y) 2 \) (respectively, \( w_s(x) 2 \)). Therefore,

\[
w_t(y) 2 \stackrel{L}{\rightarrow} r_s(y) 2 \stackrel{L}{\rightarrow} w_q(y) 1 \stackrel{L}{\rightarrow} r_t(y) 1 \stackrel{L}{\rightarrow} r_t(x) 2, \text{ and}
\]

\[
w_s(x) 2 \stackrel{L}{\rightarrow} r_t(x) 2 \stackrel{L}{\rightarrow} w_p(x) 1 \stackrel{L}{\rightarrow} r_s(x) 1 \stackrel{L}{\rightarrow} r_s(y) 2
\]

imply the cycle: \( r_s(y) 2 \stackrel{L}{\rightarrow} r_t(x) 2 \stackrel{L}{\rightarrow} r_s(y) 2 \).

Therefore, Computation 10 is not PSO\textsubscript{base}, and consequently it is not TSO\textsubscript{base}.

\[\blacksquare\]

**Lemma 4.4.4** \( P\text{-RAM}, PC\text{-G}, \) and \( CC \) are each not stronger than each of PSO\textsubscript{base} and TSO\textsubscript{base}.

**Proof:** We show that Computation 11 is TSO\textsubscript{base} but not P-RAM.

**Computation 11**

\[
\begin{align*}
p : & w(x) 4 \ r(y) 1 \ r(y) 2 \ r(y) 3 \ r(x) 4 \\
q : & w(y) 1 \ w(y) 2 \ w(x) 6 \ w(y) 3
\end{align*}
\]

If Computation 11 were P-RAM, then there is a linearization for \( p \), \( (O|p \cup O|w, \stackrel{L_p}{\longrightarrow}) \),
that satisfies Definition 2.2.4. That is, it must maintain \( q \)'s program order:

\[
\begin{align*}
\text{w}_q(y) & \xrightarrow{L_p} \text{w}_q(y) 2 \xrightarrow{L_p} \text{w}_q(x) 6 \xrightarrow{L_p} \text{w}_q(y) 3 \\
\end{align*}
\]

Also, validity requires:

\[
\begin{align*}
\text{w}_q(y) & \xrightarrow{L_p} \text{r}_p(y) 1 \xrightarrow{L_p} \text{w}_q(y) 2 \xrightarrow{L_p} \text{r}_p(y) 2 \xrightarrow{L_p} \text{w}_q(y) 3 \xrightarrow{L_p} \text{r}_p(y) 3 \\
\end{align*}
\]

This necessitates that \( \text{w}_p(x) 4 \xrightarrow{L_p} \text{w}_q(x) 6 \), and subsequently, \( \text{r}_p(x) 4 \xrightarrow{L_p} \text{w}_q(x) 6 \) to maintain validity. However, since \( \text{w}_q(y) 3 \xrightarrow{L_p} \text{r}_p(y) 3, \text{w}_q(y) 3 \xrightarrow{L_p} \text{r}_p(x) 4 \). Thus, \( \text{w}_q(x) 6 \xrightarrow{L_p} \text{r}_p(x) 4 \).

Therefore, \( (O|p \cup O|w, \xrightarrow{L_p}) \) contains the following cycle

\[
\begin{align*}
\text{r}_p(x) 4 \xrightarrow{L_p} \text{w}_q(x) 6 \xrightarrow{L_p} \text{r}_p(x) 4 \\
\end{align*}
\]

We conclude that Computation 11 is not P-RAM.

Since P-RAM is strictly weaker than CC and PC-G, Computation 11 is neither CC nor PC-G.

However, Computation 11 is TSO\textsubscript{base}. First, note that \( (O|q, \xrightarrow{\text{tso}}) = (O|q, \xrightarrow{\text{prog}}) \) in this computation. However, for \( (O|p, \xrightarrow{\text{tso}}) \), we only have \( \text{w}_p(x) 4 \xrightarrow{\text{tso}} \text{r}_p(x) 4 \) and \( \text{r}_p(y) 1 \xrightarrow{\text{tso}} \text{r}_p(y) 2 \xrightarrow{\text{tso}} \text{r}_p(y) 3 \xrightarrow{\text{tso}} \text{r}_p(x) 4 \). Therefore, the following is the required linearization:

\[
(O, \xrightarrow{L}) = (\text{w}_q(y) 1, \text{r}_p(y) 1, \text{w}_q(y) 2, \text{r}_p(y) 2, \text{w}_q(x) 6, \text{w}_q(y) 3, \text{r}_p(y) 3, \text{w}_p(x) 4, \text{r}_p(x) 4)
\]

Computation 11 is TSO\textsubscript{base}, and therefore, PSO\textsubscript{base}.

\[\blacksquare\]

**Theorem 4.4.5** \( \text{PSO}_{\text{base}} \) is strictly stronger than Coherence.

**Proof:** First we argue that the following computation is Coherent but not PSO\textsubscript{base}.

**Computation 12** \[\]
\[
\begin{align*}
p &: \text{w}(y) 2 \ x(y) 1 \ x(w) 1 \\
q &: \text{w}(x) 2 \ y(x) 1 \ y(w) 1 \\
\end{align*}
\]

The required Coherence linearizations for \( x \) and \( y \) are as follows.
In PSO base, the operation \( r_p(y) 1 \) accesses main memory, which means at the time of performing \( r_p(y) 1 \), \( w_p(y) 2 \) must have been overwritten by \( w_q(y) 1 \). This in turn implies that \( w_p(x) 1 \) was not even buffered when \( w_q(y) 1 \) was committed to main memory. So, how could \( r_q(x) 1 \) return a 1? More precisely in Computation 12, \( (O, \xrightarrow{pso}) = (O, \xrightarrow{prog}) \) and the cycle in the following sequence prohibits forming a PSO base linearization:

\[
\begin{align*}
w_p(y) 2 &\xrightarrow{L} w_q(y) 1 \xrightarrow{L} r_p(y) 1 \xrightarrow{L} w_p(x) 1 \xrightarrow{L} r_q(x) 1 \xrightarrow{L} w_q(y) 1
\end{align*}
\]

However, every PSO base computation is Coherent because \( (O, \xrightarrow{pso}) = (O, \xrightarrow{prog}) \), by definition.

Since Coherence is strictly stronger than WO base, we infer the following corollary.

**Corollary 4.4.6** PSO base is strictly stronger than WO base.

Theorem 4.4.1 establishes that TSO base is strictly stronger PSO base. Thus, the following corollary.

**Corollary 4.4.7** TSO base is strictly stronger than Coherence and WO base.

Finally we establish that TSO base (consequently, PSO base) is strictly weaker than SC (consequently, Linearizability).

**Theorem 4.4.8** SC is strictly stronger than TSO base.

**Proof:** Theorem 4.4.2 shows that Computation 11 is TSO base but not PC-G. Since it is not PC-G, it cannot be SC. Moreover, every SC computation is also TSO base. This situation occurs when TSO base behaves as if the buffers do not exist. Formally, every SC computation is also TSO base because \( (O, \xrightarrow{tso}) \subset (O, \xrightarrow{prog}) \).

\[
\qed
\]
4.5. Summary

We have derived non-operational definitions for the TSO and PSO memory consistency models, which are supported by the SPARC v8 architecture. We have shown that with the use of explicit synchronization, TSO and PSO can be used to guarantee SC. However, the TSO and PSO models without synchronization primitives (TSO\textsubscript{base} and PSO\textsubscript{base}) are incomparable with most of the memory models defined in Chapter 2. The relationships are summarized in the Hasse diagram of Figure 4.4.

Figure 4.4: Relationships between SPARC models and other models

\textit{A} \rightarrow \textit{B} indicates that model \textit{A} is strictly stronger than model \textit{B}.
One of the major advantages of Java is that it includes concurrency as part of the programming language itself [53]. The memory consistency model that results from multi-threaded Java computations on a Java Virtual Machine is formally derived in this chapter. First, we infer a simpler description of the Java Virtual Machine than that described in the Java manuals [55, 35]. Then, we derive the non-operational formal description. Finally, we compare Java Consistency with the models defined in Chapter 2 as well as the SPARC TSO and PSO models of Chapter 4. Our definition is also compared with another proposed modification to Java Consistency. Process coordination problems under Java Consistency are discussed in chapters 6 and 7.

### 5.1 Operational Description

The Java Virtual Machine (JVM) [55] is an abstract machine introduced by SUN Microsystems to support the Java programming language [35]. Its behavior for multi-threaded execution is specified in the Java manuals [35, 55] and is quoted in Appendix C. These constraints will be referred to in this chapter according to their numbers in the appendix. This section provides a simple, precise alternative but equivalent description.

The components and events of JVM are depicted in Figure 5.1 for a two-thread machine. The *working memory* is local to a thread and is accessible by *use* and *assign* events. A *use* investigates (reads) the working memory and an *assign* updates (writes) it. The *main memory* is accessible to a thread by *load*, *store*, *read*, and *write* events. Main memory
is also accessible directly from the thread through pairs of `lock` and `unlock` events.

To commit an `assign` to main memory, the working memory performs a `store` event. Later, the main memory performs a subsequent `write` event, which updates main memory. Before a thread attempts to use a location that is not in its working memory, main memory performs a `read` event. Later, the working memory issues a `load` bringing a copy of the location to the working memory. A component called the `waiting area` is introduced to model the delay between `stores` and `writes` and between `reads` and `loads`.

5.1.1 Operational Java with Base Operations

Java is a hybrid model that distinguishes between three types of objects: read/write variables, volatile variables, and lock variables. To simplify presentation, we will first describe JVM with events that correspond to base read and write operations. The resulting machine is denoted $M_{JVM}^b$. Synchronization operations will be added in the next section. The base read/write variables are as defined in Chapter 2.

For memory consistency concerns, a Java thread is considered to be a sequence of read
and write\textsuperscript{1} invocations, which are implemented in the JVM machine by the algorithms indicated in Figure 5.2.

Algorithm (A.1) captures constraints C.1.4 and C.1.5. It indicates that a thread attempts to read a location locally. If the location has no local copy, then it must be “loaded” from main memory. Otherwise, the thread non-deterministically chooses to load it or not. Similarly, Algorithm (A.2) indicates that a write invocation is performed locally and again, the thread non-deterministically chooses to store it or not. With base variables, the only constraint that forces the choice function in Algorithm (A.2) to succeed is Constraint C.1.2, which indicates that an assign is forced to be written back (stored) if there is a following load by the same thread to the same location. The get and put functions are derived from constraints C.2.1 and C.2.2, respectively.

Let $M_{JVM}^b = (\Pi, \Sigma, E^b, N^b, I^b, R^b)$. The events of $E^b$ are of six types: use, assign, load, store, read, and write. The use and the assign events are the request events. That is, $(E^b, \rightarrow)$ is defined on uses and assigns.

There are two base operation types that access memory: read and write. A read invocation by thread $t$ is implemented by a use that accesses $t$’s working memory. The working

\textsuperscript{1}The font distinguishes read and write events from read and write operations.

\begin{align*}
(A.1) & \quad \text{read}, x, \lambda) \text{ by thread } t: \\
& \quad \begin{cases} 
\text{if } x \text{ is not in } t's \text{ working memory} \\
\text{then } \text{get}(t, x, v) \\
\text{else choice} \{ \text{get}(t, x, v) \}
\end{cases} \\
& \quad \text{use}(t, x, v)
\end{align*}

\begin{align*}
(A.2) & \quad \text{write}, x, (v) \text{ by thread } t: \\
& \quad \begin{cases} 
\text{assign}(t, x, v) \\
\text{choice} \{ \text{put}(t, x, v) \}
\end{cases} \\
& \quad \text{write}(t, x, v)
\end{align*}

Figure 5.2: JVM implementation of read and write invocations

t: thread, x: object, v: constant value, choice\{f\}: non-deterministic choice to perform f or not.
copy of a location might be brought from main memory before the use, by a read followed by a load. Similarly, a write invocation is implemented by an assign that updates the working copy. Sometimes a working copy is written back to main memory, by a store followed by a write. Thus,

- \( I^b((\text{read}, x, \lambda), t) = \{ \langle \text{use}(t, x, v') \rangle, \langle \text{read}(t, x, v'), \text{load}(t, x, v'), \text{use}(t, x, v') \rangle \} \).
- \( I^b((\text{write}, x, (v)), t) = \{ \langle \text{assign}(t, x, v) \rangle, \langle \text{assign}(t, x, v), \text{store}(t, x, v), \text{write}(t, x, v) \rangle \} \).

The nondeterministic implementation of \( I^b \) for read and write invocations is further restricted by \( R^b \), which specifies five rules. The first specifies that, in general, read and write invocations are implemented according to algorithms (A.1) and (A.2) capturing constraints C.1.4, C.1.5, C.2.1, and C.2.2. The second and third rules capture constraints C.1.2 and C.1.3, respectively. Constraint C.2.3 implies that the order of read and write events to a particular location follows the order of their matching load and store events. Since the order of load and store events to a particular location follows that of their matching use and assign events, the fourth rule specifies that for a given location, the order of the request events determines that of the rest of the matching events. The last rule (\( \rho_5^b \)) captures validity which is implicit in the Java specifications (constraints C.5.1 and C.5.2). Precisely, \( R^b = \{ \rho_i | 1 \leq i \leq 5 \} \), where

- \( \rho^b_1 \): \( I^b \) implements read and write invocations according to algorithms (A.1) and (A.2).
- \( \rho^b_2 \): If assign\((t, x, v) \xrightarrow{e} \text{load}(t, x, v?)\), then there exists a store\((t, x, v)\) satisfying assign\((t, x, v) \xrightarrow{e} \text{store}(t, x, v) \xrightarrow{e} \text{load}(t, x, v?)\).
- \( \rho^b_3 \): If store\((t, x, v) \xrightarrow{e} \text{store}(t, x, u)\), then there exists an assign\((t, x, u)\) satisfying store\((t, x, v) \xrightarrow{e} \text{assign}(t, x, u) \xrightarrow{e} \text{store}(t, x, u)\).
- \( \rho^b_4 \): Let \( e_1, e_2, \) and \( e_r \) be matching events by thread \( t \) to location \( x \), and let \( e_1', e_2', \) and \( e_r' \) also be matching events by \( t \) to \( x \). If \( e_r \) and \( e_r' \) are the request events and \( e_r \xrightarrow{e} e_r' \), then \( e_1 \xrightarrow{e} e_1' \) and \( e_2 \xrightarrow{e} e_2' \).
5.1. Operational Description

- $\rho_b^3$: $\text{read}(t, x, v? \leftarrow v)$ where $\text{write}(s, x, v)$ is the most recent $\text{write}$ event that precedes the $\text{read}$ in $\Xi|x$. Also, $\text{load}(t, x, v? \leftarrow v)$ such that $\text{read}(t, x, v? \leftarrow v)$ is the most recent $\text{read}$ by $t$ that precedes the $\text{load}$ in $\Xi|x$. Finally, $\text{use}(t, x, v? \leftarrow v)$ such that the most recent $\text{load}$ or $\text{assign}$ event by $t$ that precedes the $\text{use}$ in $\Xi|x$ is $\text{assign}(t, x, v)$ or $\text{load}(t, x, v? \leftarrow v)$.

Unlike Constraint C.1.3, $\rho_b^3$ is restricted to a $\text{store}$ followed by another $\text{store}$. The case where a $\text{load}$ of $x$ is followed by a $\text{store}$ of $x$ is captured by $\rho_2^b$ combined with the implementation function. Consider a situation where $\text{load}(t, x, v) \rightarrow \text{store}(t, x, u)$. By $I_b^b$, there must be a matching $\text{assign}$ to the $\text{store}$ that precedes it. By $\rho_2^b$, this $\text{assign}$ cannot precede the $\text{load}$, otherwise the $\text{store}$ must precede the $\text{load}$ as well. Thus, the $\text{assign}$ must intervene between the $\text{load}$ and the $\text{store}$.

Implicit JVM Constraints

The definition of $I_b^b$ on read and write operations implies that

- every $\text{load}$ has exactly one matching $\text{use}$, and
- every $\text{store}$ has exactly one matching $\text{assign}$.

Although implicitly assumed, the first constraint is never stated explicitly in the manuals. The Java Virtual Machine Specifications [55] states that:

"A single Java thread issues a stream of $\text{use}$ [and] $\text{assign}$ [...] operations as dictated by the semantics of the Java program it is executing. The underlying Java implementation is then required additionally to perform appropriate $\text{load}$, $\text{store}$, $\text{read}$, and $\text{write}$ operations [...]"

The term "appropriate" in the above quote is ambiguous for $\text{loads}$. None of the Java constraints (see Appendix C) prevents a $\text{load}$ to be unmatched with a $\text{use}$. However, even if we want to assume that certain floating, never-used $\text{loads}$ exist in a Java execution, these can be ignored since they do not affect the outcome of the execution. What really matters is a $\text{load}$ that has a matching $\text{use}$.
On the other hand, it is clear from constraints C.1.3, C.1.4, and C.1.5 that every store has a matching assign.

**Simplified Description of JVM**

The memory consistency model arising from the $M_{JVM}^b$ is equivalent to that of an even simpler machine. The simpler machine treats the get and put events of Algorithms (A.1) and (A.2) as indivisible events. Consequently, load, read, store, and write events are replaced by get and put events which, in turn, results in a simpler set of constraints. The corresponding simplified machine is depicted in Figure 5.3.

Define $M_{JVM}^s = (\Pi, \Sigma, E^s, N^b, I, R^s)$. The set $E^s$ contains four types of events: use, assign, get, and put. The use and assign events are the request events. The function $I$ for $N^b$ is given below:

- $I((\text{read}, x, \lambda), t) = \{\text{use}(t, x, v?), \text{get}(t, x, v?), \text{use}(t, x, v?)\}$.
- $I((\text{write}, x, (v)), t) = \{\text{assign}(t, x, v), \text{assign}(t, x, v), \text{put}(t, x, v)\}$.

The set of rules $R^s$ is revised as follows:\footnote{Rules 4 to 10 are reserved for synchronization events and are given in the next section.}
5.1. Operational Description

- $\rho_1$: $I$ implements read and write invocations according to algorithms (A.1) and (A.2), where the get and put are indivisible events.
- $\rho_2$: If $\text{assign}(t, x, v) \xrightarrow{\Xi} \text{get}(t, x, v?)$, then there is a $\text{put}(t, x, v)$ satisfying $\text{assign}(t, x, v) \xrightarrow{\Xi} \text{put}(t, x, v) \xrightarrow{\Xi} \text{get}(t, x, v?)$.
- $\rho_3$: Let $e$ and $e_r$ be matching events by thread $t$ to location $x$, and let $e'$ and $e_r'$ also be matching events by $t$ to $x$. If $e_r$ and $e_r'$ are the request events and $e_r \xrightarrow{\pi} e_r'$, then $e \xrightarrow{\Xi} e'$.
- $\rho_{11}$: $\text{get}(t, x, v? \leftarrow v)$ where $\text{put}(t, x, v)$ is the most recent put event that precedes the get in $\Xi|x$. Also, $\text{use}(t, x, v? \leftarrow v)$ such that the most recent get or assign event by $t$ that precedes the use in $\Xi|x$ is $\text{assign}(t, x, v)$ or $\text{get}(t, x, v? \leftarrow v)$.

Let $\Xi^b$ be a run of some system $(P, J)$ with operation invocations in $N^b$ on $M_{JVM}^b$. Similarly, let $\Xi^s$ be a run of the same system $(P, J)$ on $M_s^b$. We now show that the simplified description is equivalent to the original description. That is, we show that the computation induced by $\Xi^b$ has the same outcome as that induced by $\Xi^s$ and vice-versa.

Note that in the simplified machine there is no corresponding rule to $\rho^b_3$. While $\rho^b_3$ requires an assign to intervene between any two consecutive stores to the same location by the same thread, the simplified machine allows two consecutive puts to the same location by the same thread without any intervening assigns.

**Theorem 5.1.1** $M_s^b$ is equivalent to $M_{JVM}^b$.

**Proof:** The first direction constitutes showing that any run $\Xi^b$ of $M_{JVM}^b$ can be converted into a run $\Xi^s$ of $M_s^b$, such that the computations induced by $\Xi^s$ and $\Xi^b$ are equivalent. Given $\Xi^b$, construct $\Xi^s$ by deleting all load and store events and renaming reads as gets and writes as puts. In the following discussion, $x$ represents a variable and $t$ a thread.

It is easily verified that the sequence $\Xi^s$ complies with $I$. Also, $\rho_1$ is trivially satisfied. To see that rule $\rho_2$ is satisfied consider an assign that is followed by a get in $\Xi^s|x|t$. Then, the assign is followed by a read in $\Xi^b|x|t$. By $I^b$, the load and use that match this read must follow the assign in $\Xi^b|x|t$. By $\rho^b_2$ and $\rho^b_3$, there must be an intervening store between the assign and the load, and the matching write to this store must precede the
read because all these events are to the same location and the assign precedes the use. In Ξ', this write was renamed put and the read was renamed get. Therefore, the put intervenes between the assign and the get in Ξ', satisfying ρ2.

Rule ρ3 is trivially satisfied. For ρ11, a get instantiates a value according to a put because reads and writes have been named to gets and puts, respectively. This follows from ρ3b. For a use that instantiates a parameter according to an assign, the conclusion still holds in Ξ'. We need to show that if the most recent event that precedes the use in Ξ[x:t] is a get, then the use returns the value returned by the get. Since in Ξb[x:t] the load is the most recent load event that precedes the use, it must be the case that the matching read to the load is the most recent read event that precedes the use in Ξb[x:t], by ρ4b. Therefore, ρ11 is satisfied.

For the other direction, we show how to construct Ξb from Ξs, each assign that has a matching put is replaced by an assign-store pair, and each put is replaced by a write. Finally, each get is replaced by a read-load pair.

The constructed sequence complies with I b, and rule ρ1b is trivially satisfied. To see that ρ2b is satisfied, note that ρ2 requires a put(t, x, v) to intervene between an assign(t, x, v) and a following get(t, x, v?). This means that such assign was either replaced by an assign-store pair or is followed by another assign to the same location that was replaced by an assign-store pair. Since the get was replaced by a read-load pair in Ξb, the store intervenes between the assign and the load, establishing ρ2b. Since stores are matched always with immediately preceding assigns, ρ3b is satisfied. Rules ρ4b and ρ5b are easily checked.

In the subsequent sections, we will only refer to the simplified JVM.

5.1.2 Operational Java with Synchronization Operations

There are two kinds of synchronization operations in Java: operations performed on lock locations (simply locks) and operations performed on volatile locations (simply volatiles).
5.1. Operational Description

Locks

A *Java lock variable* \( l_x \) is specified by the set of all sequences \( \langle o_1, o_2, \cdots \rangle \) such that

1. each \( o_i \) is either \( \text{lock}(l_x, \lambda, \lambda) \) or \( \text{unlock}(l_x, \lambda, \lambda) \),

2. there is a one-to-one correspondence between lock and unlock operations,

3. each pair of corresponding lock and unlock operations are performed by the same process,

4. each lock operation precedes its corresponding unlock operation,

5. for each \( \text{lock}(l_x, \lambda, \lambda) \) by process \( p \), the number of preceding \( \text{lock}(l_x, \lambda, \lambda) \) operations by each \( q \neq p \) is equal to the number of preceding \( \text{unlock}(l_x, \lambda, \lambda) \) operations by \( q \neq p \), and

6. for each \( \text{unlock}(l_x, \lambda, \lambda) \) operation, the number of preceding \( \text{lock}(l_x, \lambda, \lambda) \) operations is strictly larger than the number of preceding \( \text{unlock}(l_x, \lambda, \lambda) \) operations.

When methods or statements are declared “synchronized”, the Java compiler forces the executing thread to acquire a lock before they are executed and release the lock after execution. Acquiring and releasing locks is supported by the JVM through \text{lock} and \text{unlock} events, respectively. So, every \text{lock} event has exactly one matching \text{unlock} event, and the \text{unlock} event always follows its matching \text{lock} event in the execution.

Access to a lock is mutually exclusive. That is, only one thread can acquire a given lock at a time. Furthermore, a lock acquired by a thread can only be released by the same thread. It is also possible for a thread to acquire many locks before releasing any. Thus, a JVM execution may consist of nested \text{lock} and \text{unlock} events.

A \text{lock} forces a thread to start with a “clean” working memory (the thread’s working memory is invalidated), and an \text{unlock} forces the “dirty” part of the working memory of that thread to be put to main memory.

Formally, in addition to the support of read and write invocations, \( M_{JVM} \) supports \( \text{lock}(l_x, \lambda) \) and \( \text{unlock}(l_x, \lambda) \) invocations. By the preceding discussion, the \text{lock} and \text{unlock} events can be regarded as fence events that do not access main memory. The
implementation of lock and unlock operations is given below.

- \( I((\text{lock}, l_x, \lambda), t) = \{ \langle \text{lock}(t, l_x) \rangle \} \).

- \( I((\text{unlock}, l_x, \lambda), t) = \{ \langle \text{unlock}(t, l_x) \rangle \} \).

The rules \( R \) include further components. The first, \( \rho_4 \), captures constraints C.3.1, C.3.2, and C.3.3. Rule \( \rho_5 \) captures Constraint C.3.4 and rules \( \rho_6 \) and \( \rho_7 \) capture Constraint C.3.5. Let \( \sigma \subset \Sigma \) denote the collection of lock locations. Then, \( R \) is extended below.

- \( \rho_4 \): For a given lock location \( l_x \in \sigma \), there is a total order on all lock and unlock events applied to \( l_x \) that satisfies the following:
  - the total order is consistent with the threads request order \( (E, \pi) \),
  - for each lock applied to \( l_x \) by thread \( t \), the number of preceding lock events applied to \( l_x \) by every other thread is equal to the number of preceding unlock events applied to \( l_x \) by every other thread.
  - for each unlock applied to \( l_x \), the number of preceding lock events applied to \( l_x \) is greater than the number of preceding unlock events applied to \( l_x \).

- \( \rho_5 \): Let \( e \) be an unlock event by thread \( t \) and let \( e_1 \) be an assign event by \( t \) to \( x \). If \( e_1 \xrightarrow{\pi} e \), then there must be a put event \( e_2 \) by \( t \) to \( x \) such that \( e_1 \xrightarrow{\Xi} e_2 \xrightarrow{\Xi} e \).

- \( \rho_6 \): Let \( e \) be a lock event by thread \( t \) and let \( e_1 \) be a put event by \( t \) to \( x \). If \( e \xrightarrow{\pi} e_1 \), then there must be an assign \( e_2 \) by \( t \) to \( x \) such that \( e \xrightarrow{\Xi} e_2 \xrightarrow{\Xi} e_1 \).

- \( \rho_7 \): Let \( e \) be a lock event by thread \( t \) and let \( e_1 \) be a use event by \( t \) to \( x \). If \( e \xrightarrow{\pi} e_1 \), then there must be a get or an assign \( e_2 \) by \( t \) to \( x \) such that \( e \xrightarrow{\Xi} e_2 \xrightarrow{\Xi} e_1 \).

**Volatile**

Volatile variables allow the same operations allowed by read/write variables; however, they restrict their JVM implementations. For this reason, a Java volatile variable is a read/write variable marked with a “tag”, which distinguishes it from a base variable.

For base locations, a matching get (respectively, put) for a use (respectively, assign) could be optional. This non-deterministic behavior is prohibited with volatiles. When a
location is declared “volatile”, a use (respectively, assign) is forced to be paired with a get (respectively, put) all the time. Furthermore, those paired events must not have between them any intervening events to the same location by the same thread.

Let $\nu \subseteq \Sigma$ denote the collection of volatile locations. The operations on these locations can be implemented by the same events as used for read/write variables. However, the implementation of any operation invocation to a variable in $\nu$ must consist of an ordered pair of events, and their interleaving is further restricted by the rules $\rho_8$ to $\rho_{10}$, which capture constraints C.4.1 to C.4.3.

- $\rho_8$: For a given volatile variable in $\nu$, read and write invocations are always implemented by pairs of events (The choice function in algorithms (A.1) and (A.2) always succeeds). That is, a read invocation is implemented by a get followed by a use, and a write invocation is implemented by an assign followed by a put.

- $\rho_9$: Let $e_1$ and $e_2$ be two matching events applied to a volatile location $v_x$ by thread $t$, then there are no intervening events by $t$ to $v_x$ between $e_1$ and $e_2$.

- $\rho_{10}$: Let $e$ and $e_r$ be two matching events by thread $t$ applied to volatile location $v_x$ and let $e'$ and $e'_r$ be two matching events by $t$ applied to volatile location $v'_x$. If $e_r \xrightarrow{\pi} e'_r$, then $e \xrightarrow{\Xi} e'$.

Summary

In summary, $M_{JVM} = (\Pi, \Sigma, E, N, I, R)$, where $E$ supports six types of events: use, assign, get, put, lock and unlock. All of these events are request events except get and put events. The operation invocations $N$ contains all the read, write, lock, and unlock invocations. The function $I$ is the same as that defined for $M_{JVM}^s$ in addition to lock and unlock invocation implementations. Finally, $R = \{\rho_i|1 \leq i \leq 11\}$. 
5.2 Non-Operational Description

5.2.1 Invisibility

The rules of JVM that determine the inter-operation between working memories and main memory permit a process’s write operation to be invisible to another process. This is highlighted by the appearance of the choice function in algorithms (A.1) and (A.2). JVM gives rise to two kinds of invisibilities. First, certain puts are optional, which makes some assigns visible to the process that issued them, but invisible to others. This invisibility is covert. Second, a get is optional when a process already has a value for the required variable recorded in its working memory, which can cause a use to retrieve a stale value rather than seeing a value newly written by another process. This is fixate invisibility.

To define the memory consistency model of Java, the obstacles that arise from covert and fixate invisibilities can be cleanly and elegantly finessed as long as computations are finite as shown next by the definition of Gontmakher et al. [32, 33]. Those ideas, however, do not suffice for non-terminating Java computations. We provide a new definition of consistency that is correct for both terminating and non-terminating Java computations.

5.2.2 Non-Operational Java for Terminating Computations

Gontmakher et al. [32, 33] gave non-operational definitions for Java memory behavior. We use Java1 for their “programmer’s view” characterization, after translation to our framework. Recall that a read by process $p$ that returns the value written by process $q \neq p$ is called foreign; otherwise, it is domestic.

**Base Java Partial Program Order:**

Define the *base Java partial program order*, denoted $(O, \rightarrow_{bjpo})$, by $o_1 \rightarrow_{bjpo} o_2$ if $o_1 \rightarrow_{prog} o_2$ and one of the following holds:

1. **same-object**: $o_1, o_2 \in O[x$ for some $x \in J$,
2. **read-before-write:** \( o_1 \in O|r, o_2 \in O|w, \) and \( o_1 \) is foreign, or

3. **transitivity:** there is an operation \( o' \) such that \( o_1 \xrightarrow{bipo} o' \xrightarrow{bipo} o_2. \)

**Definition 5.2.1** [32] Let \( O \) be all the operations of a computation \( C \) of a multiprocess system \( (P, J) \). Then \( C \) is Java\(_1\) if there is some linearization \( (O, \xrightarrow{L}) \) satisfying \( (O, \xrightarrow{bipo}) \subseteq (O, \xrightarrow{L}). \)

Notice that this definition requires one linearization for all operations. Gontmakher et al. [32, 33] prove that their definition does capture exactly all terminating Java computations.

There are two essential ideas in forming the linearization \( (O, \xrightarrow{L}) \):

- certain covert writes can be moved to the end of the linearization so that these writes are never read by any other process and hence do not negate validity.

- fixate reads could be moved earlier in the linearization to precede the writes that are invisible to the reader so that the stale value returned is valid.

Furthermore, Java partial program order is just enough to permit these writes and reads to be moved as described.

A problem arises with Definition 5.2.1 when a system is non-terminating for two reasons. First, the end of the computation is not defined. Second, it is not always possible to tell whether a write operation has completed or not. Precisely, some write operations that have been implemented by an `assign` event might still be followed by a `put`. In certain situations, it is impossible to determine if such a `put` is going to take place in the future or not. Recall the following computation given in Chapter 2.

**Computation 13**

\[
\begin{align*}
p & : [r(x)0], [r(x)0], [r(x)0], [r(x)0], ... \\
q & : w(x)1
\end{align*}
\]

In Computation 13, process \( p \) continues to read 0 for \( x \) even though \( q \) at some point writes 1 to \( x \). This could arise as a JVM computation either 1) from a fixate invisibility of \( p \) to the updated value of \( x \) because (with the exception of the very first `use`) none of \( p \)’s `uses` is preceded by a matching `get`, or 2) from a covert invisibility of \( x \) because \( q \)’s `assign` was not succeeded by a `put`.  


Does Computation 13 satisfy Definition 5.2.1? Certainly for any finite prefix of \( p \)'s computation, say after \( i \) reads by \( p \), it is Java\( _1 \), since the linearization \([r(x)0]^i w(x)1\) satisfies the definition. However the linearization(s) required by a consistency model are meant to capture each system component’s “view” of the computation. For Java, the given linearization means that \([r(x)0]^i w(x)1\) is consistent with each process’s view. We expect that, as the computation continues, processes extend their respective views, but do not “change their minds” about what happened earlier.

Indeed, if we apply Definition 2.5.1 to Definition 5.2.1, Computation 13 is not Java\( _1 \). That is, any linearization of a finite prefix of the computation that contains \( q \)'s write and satisfies Definition 5.2.1 cannot be extended to a linearization for a longer prefix of the computation that still satisfies the definition. Instead, the write operation by \( q \) would have to be moved to the new end of the linearization. However, Computation 13 is a possible computation arising from JVM implying that the definition must be adjusted for not-terminating computations.

### 5.2.3 Non-Operational Java with Base Operations

To simplify presentation, we will first define the Java memory consistency model ignoring locks and volatiles. Use Java\(_{\text{base}}\) to refer to the resulting memory model. The Java Consistency definition that includes locks and volatiles will be given in the next section.

To develop the Java\(_{\text{base}}\) definition, we first define Java\(_2\) a model that is equivalent to Java\(_1\) but is described from the point of view of processes.

**Definition 5.2.2** Let \( O \) be all the operations of a computation \( C \) of a multiprocess system \((P, J)\). Then \( C \) is Java\(_2\) if there is a total order \( (O|w, \text{write}) \) and \( \forall p \in P \) there is a linearization \( (O|p \cup O|w, \text{write}_{L_p}) \) satisfying:

1. \( (O|p \cup O|w, \text{write}_{L_p}) \subseteq (O|p \cup O|w, \text{write}_{L_p}), \) and
2. \( (O|w, \text{write}_{L_p}) = (O|w, \text{write}_{L_p}). \)

**Lemma 5.2.3** Java\(_1\) is equivalent to Java\(_2\).
5.2. NON-OPERATIONAL DESCRIPTION

Proof: Java$_1$ is stronger than Java$_2$. Given $(O, \rightarrow)$ guaranteed by Definition 5.2.1, the total order $(O|w, \xrightarrow{\text{writes}})$ is built by projecting $(O, \rightarrow)$ to write operations. That is, $(O|w, \xrightarrow{\text{writes}}) = (O|w, \xrightarrow{L})$. Similarly, $(O|p \cup O|w, \xrightarrow{L_p})$ is built from $(O, \xrightarrow{L})$ by deleting all read operations by $q \neq p$. The argument that Definition 5.2.2 holds is trivial.

Java$_2$ is stronger than Java$_1$. By Condition 2 of Definition 5.2.2, for any process $p$ the order of $O|w$ in $\xrightarrow{L_p}$ is the same as that in $\xrightarrow{\text{writes}}$. Let $(O|w, \xrightarrow{\text{writes}})$ be $w_1, w_2, \ldots, w_n$. Then, $(O|p \cup O|w, \xrightarrow{L_p}) = S_0^p, w_1, S_1^p, \ldots, S_{n-1}^p, w_n, S_n^p$ where the $S_i^p$’s contain only $p$’s read operations. Construct $(O, \xrightarrow{L})$ by adding the $S_i^p$’s to $(O|w, \xrightarrow{\text{writes}})$ each after $w_i$ and before $w_{i+1}$, for each $p$. Since each $(O|p \cup O|w, \xrightarrow{L_p})$ is a linearization and since only read operations are added, it follows immediately that $(O, \xrightarrow{L})$ is a linearization. It is also obvious that $(O, \xrightarrow{\text{bijpo}}) \subseteq (O, \xrightarrow{L})$.

We further adjust Definition 5.2.2 to cope with invisibility and hence capture both terminating and non-terminating Java computations.

Recall that Java$_2$ requires all processes to agree on the order of all the writes in a computation, $(O|w, \xrightarrow{\text{writes}})$. In Java$_{\text{base}}$, the guarantees on visibility are fewer; moreover, “extreme” non-determinism in making operations visible further complicates the issue. It is delicate in Java to define the set of guaranteed visible operations, or alternatively the set of all invisible operations. Because of these invisibilities, a mutual agreement on $(O|w, \xrightarrow{\text{writes}})$ in Java$_{\text{base}}$ is not generally possible.

**Guaranteed Visibility**

The Java$_{\text{base}}$ definition requires a minimum agreement between processes on the order of some of the writes. Since it is optional to put an `assign`, it is not always possible to determine whether a write operation has completed or not. A write operation that has been implemented by an `assign-put` pair is known to be complete (also called a determined write). However, if the write has been implemented by an `assign` only, there are certain situations where it is impossible to tell from a prefix of a non-terminating computation whether the write is determined or there is a `put` yet to follow.
A write is in \( \text{DIRECT-VIS}(p) \) if it is causally related to at least one read by \( p \). For \( p \in P \), define \( \text{DIRECT-VIS}(p) = \{ w \mid w \in O|w \text{ and } \exists r \in O|p \wedge w \xrightarrow{\text{wbr}} r \} \).

If process \( q \) performs a write \( w \) and later another write \( w' \) both to \( x \), and \( p \) sees \( w' \), then we can assume that \( w \) is visible to \( p \). For \( p \), define \( \text{INDUCED-VIS}(p) \) for some \( x \in J \), \( w \xrightarrow{\text{prog}} w' \wedge w' \xrightarrow{\text{wbr}} r \).

The set \( O_{\text{vis}} \) includes the write operations that are “so far” visible to process \( p \). Formally, \( O_{\text{vis}} = \text{DIRECT-VIS}(p) \cup \text{INDUCED-VIS}(p) \cup O|w|p \). Note that \( O|w = \cup_{p \in P} O_{\text{vis}} \).

Synchronization operations may enlarge \( O_{\text{vis}} \) as will be seen in the next section.

**Definition 5.2.4** Let \( O \) be all the operations of a computation \( C \) of a multiprocess system \((P,J)\). Then \( C \) is \( \text{Java}_{\text{base}} \) if there is a total order \( (O|w, \xrightarrow{\text{writes}}) \) and \( \forall p \in P \) there is a linearization \( (O|p \cup O_{\text{vis}}, \xrightarrow{L_p}) \) satisfying:

1. \( (O|p \cup O_{\text{vis}}, \xrightarrow{L_p}) \subseteq (O|p \cup O_{\text{vis}}, \xrightarrow{\text{write}}) \), and
2. \( (O_{\text{vis}}, \xrightarrow{L_p}) = (O_{\text{vis}}, \xrightarrow{\text{write}}) \).

**Theorem 5.2.5** \( \text{M}_{JVM} \) exactly implements \( \text{Java}_{\text{base}} \).

The following two lemmas prove Theorem 5.2.5. We first introduce some terminology that simplifies the proofs. If \( e \) is an assign (respectively, use) that does not have a matching put (respectively, get) and \( e' \) is the most recent put (respectively, get) that follows (respectively, precedes) \( e \) in \( \Xi|x \), then \( e \) and \( e' \) are said to be semi-matching. If a use (respectively, get) \( e \) returns the value of an assign (respectively, put) \( e' \) in \( \Xi \), then \( e \) and \( e' \) are said to be causally-matching. An assign is visible to \( p \) if it is by \( p \) or it semi-matches some put that causally-matches a get by \( p \).

**Lemma 5.2.6** \( \text{M}_{JVM}^s \) provides \( \text{Java}_{\text{base}} \).

**Proof:** Let \((P,J)\) be a system with operation invocations in \( N^b \). Let \( \Xi \) be a run of \((P,J)\) on \( \text{M}_{JVM}^s \) and \( C_\Xi \) be the computation of \((P,J)\) induced by \( \Xi \). Let \( O \) be all the operations that result from \( C_\Xi \). We show that \( C_\Xi \) satisfies \( \text{Java}_{\text{base}} \).

First, construct the sequence \( \widehat{\Xi} \) from \( \Xi \) as follows. Initially, \( \widehat{\Xi} = \Xi \). Iterating through \( \Xi \), consider every event \( e \). If \( e \) is an assign such that it has a matching or semi-matching put
move \( e \) such that it immediately precedes \( e' \). The assigns to location \( x \) that have not been moved in \( \hat{\Xi} \) belong to a sequence of assigns that is not terminated by a \( \text{put} \) in \( \Xi|x \).

By Algorithm (A.1), each \( \text{use} \) in \( \hat{\Xi} \) either has a matching or semi-matching \( \text{get} \) or is preceded by a causally-matching \( \text{assign} \). Iterating through \( \hat{\Xi} \), consider each \( \text{use} \) event \( e \). Let \( e' \) denote the matching or semi-matching \( \text{get} \) or causally-matching \( \text{assign} \) to \( e \). Move \( e \) so that it immediately follows \( e' \). If there are more than one such \( e \), each subsequent \( \text{use} \) event is moved such that it immediately follows the last moved \( \text{use} \) event.

Construct the order \( (O|w, \text{writes}) \) from \( \hat{\Xi} \) as follows. Let \( o \) be in \( O|w|p \) for some \( p \) and \( \langle e, e' \rangle \in I(\text{invoc}(o), p) \) or \( \langle e \rangle \in I(\text{invoc}(o), p) \). Then, \( o \)'s order in \( (O|w, \text{writes}) \) is induced by the order of \( e \) in \( \hat{\Xi} \). That is, the order \( (O|w, \text{writes}) \) is induced by the order of corresponding assigns in \( \hat{\Xi} \).

The set \( O_{vis_p} \) includes each write by \( q \neq p \) where its corresponding \( \text{assign} \) has a matching or semi-matching \( \text{put} \) that causally-matches a \( \text{get} \) by \( p \), in addition to the writes performed by \( p \) itself \((O|w|p)\). It is obvious from this construction that \( O_{vis_p} = \text{DIRECT-VIS}(p) \cup \text{INDUCED-VIS}(p) \cup O|w|p \).

The order \( (O|p \cup O_{vis_p}, \overset{L_p}{\longrightarrow}) \) is induced by the corresponding \( \text{uses} \) and \( \text{assigns} \) in \( \hat{\Xi} \). So by construction, \( (O|vis, \overset{L_p}{\longrightarrow}) = (O_{vis}, \overset{\text{writes}}{\longrightarrow}) \) establishing Condition 2 of Definition 5.2.4. We show that \( (O|p \cup O_{vis_p}, \overset{L_p}{\longrightarrow}) \) is a linearization by showing that every \( \text{use} \) returns the value of the most recent visible \( \text{assign} \) that precedes it in \( \hat{\Xi} \). Moreover, we show \( (O|p \cup O_{vis_p}, \overset{\text{bjo}}{\longrightarrow}) \subseteq (O|p \cup O_{vis_p}, \overset{L_p}{\longrightarrow}) \) by showing that assigns and \( \text{uses} \) in \( \hat{\Xi} \) preserve \( \overset{\text{bjo}}{\longrightarrow} \).

\( (O|p \cup O_{vis_p}, \overset{L_p}{\longrightarrow}) \) is a linearization:

If \( w \in (O|p \cup O_{vis_p}) \), then it is either by \( p \) or has a matching or semi-matching \( \text{put} \). Thus, \( w \)'s corresponding \( \text{assign} \) is visible. Showing that every \( \text{use} \) event by \( p \) returns the value of the most recent visible \( \text{assign} \) event that precedes it in \( \hat{\Xi}|x \) implies that \( (O|p \cup O_{vis_p}, \overset{L_p}{\longrightarrow}) \) is a linearization.

---

\( ^3 \) assigns and \( \text{uses} \) are events while \( \overset{\text{bjo}}{\longrightarrow} \) is defined on operations. The definition of \( \overset{\text{bjo}}{\longrightarrow} \) could be extended to \( \text{use} \) and \( \text{assign} \) events in a natural way.
If the use was inserted into \( \hat{\Xi} \) immediately after its causally-matching assign, then this is true by construction. If the use was inserted immediately after its matching or semi-matching get, then by validity of \( \Xi (\rho_{11}) \), the get must return the value of the most recent put event that precedes it in \( \Xi|x \). There could be an assign by a different thread intervening between the put and the get. This assign must correspond to an invisible write to \( p \); otherwise, it must have been moved to immediately precede a matching or a semi-matching put. So, a use returns the value of the most recent put event that precedes it in \( \hat{\Xi}|x \), and thus, the value of the most recent visible assign event that precedes it in \( \hat{\Xi}|x \). 

\[
(O|p \cup O_{\text{visp}}, \frac{b_{\text{pp}}}{L_p}) \subseteq (O|p \cup O_{\text{visp}}, \frac{L_p}{\text{visp}}) \leq (O|p \cup O_{\text{visp}}, \frac{L_p}{\text{visp}}).
\]

This will follow after showing that the use and assign events in \( \hat{\Xi} \) satisfy \( \frac{b_{\text{pp}}}{L_p} \). Let \( e_1 \) and \( e_2 \) be two events by the same thread, we show that if \( o_1 \xrightarrow{b_{\text{pp}}} o_2 \), then \( e_1 \) precedes \( e_2 \) in \( \hat{\Xi} \), where \( e_1 \) and \( e_2 \) are the corresponding request events to \( o_1 \) and \( o_2 \), respectively. The events \( e_1 \) and \( e_2 \) are ordered by \( \frac{b_{\text{pp}}}{L_p} \) in \( \hat{\Xi} \) if both are to the same location or \( e_1 \) is a use and \( e_2 \) is an assign. We consider each possibility separately.

**e_1 and e_2 are to the same location:**

1. both \( e_1 \) and \( e_2 \) are assigns:

   If both \( e_1 \) and \( e_2 \) do not have matching or semi-matching puts, then the construction guarantees that these events maintained their original order in \( \Xi \) because they have never been moved. If \( e_1 \) has a matching or semi-matching put, but \( e_2 \) does not, then this put must precede \( e_2 \) in \( \Xi \). In \( \hat{\Xi} \), \( e_1 \) precedes the put by construction and, consequently, it precedes \( e_2 \). If both \( e_1 \) and \( e_2 \) have the same matching or semi-matching put, then the construction guarantees that the original order in \( \Xi \) of \( e_1 \) and \( e_2 \) has been maintained. We are left with the case where \( e_1 \) and \( e_2 \) have different matching or semi-matching puts. Let \( e'_1 \) and \( e'_2 \) be the matching or semi-matching puts to \( e_1 \) and \( e_2 \), respectively. Since \( e_1 \xrightarrow{\pi} e_2 \), it follows from \( \rho_3 \) that \( e'_1 \xrightarrow{\Xi} e'_2 \). Moreover, \( e_1 \) and \( e_2 \) were inserted into \( \hat{\Xi} \) before \( e'_1 \) and \( e'_2 \), respectively. Therefore, \( e_1 \) precedes \( e_2 \) in \( \hat{\Xi} \).

2. \( e_1 \) is a use and \( e_2 \) is an assign:
It cannot be the case that $e_1$ returns the value of $e_2$ because $e_1 \not\rightarrow \pi e_2$. So, $e_1$ must return the value of some $e' \neq e_2$ and $e_1$ was inserted immediately after $e'$.

The event $e'$ could be a get or an assign.

If $e'$ is an assign, then it must be the case that $e' \not\to \pi e_2$. By an analysis similar to Case 1, $e'$ precedes $e_2$ in $\hat{\Xi}$. Since, $e_1$ was inserted after $e'$ without any intervening assigns, $e_1$ precedes $e_2$ in $\hat{\Xi}$.

If $e'$ is a get, then two cases are considered depending on whether $e_2$ has a matching or semi-matching put or not. If $e_2$ does not have a matching or semi-matching put, then $e_2$ could not have been moved. If $e_1$ was moved, it was moved backwards because $e'$ precedes $e_1$ in $\Xi$. Thus, $e_1$ precedes $e_2$ in $\hat{\Xi}$. Finally, if $e_2$ has a matching or semi-matching put $e''$, then $e' \not\rightarrow \Xi e''$ because of rule $\rho_3$ and $e_1 \not\rightarrow \pi e_2$. Since $e'$ precedes $e''$ in $\hat{\Xi}$, the construction guarantees that $e_1$ also precedes $e''$ in $\hat{\Xi}$.

3. $e_1$ is an assign and $e_2$ is a use:

If $e_2$ and $e_1$ are causally-matching, then $e_1$ precedes $e_2$ in $\hat{\Xi}$ by construction. So, let $e_2$ causally-match or match some other event $e'$. Event $e'$ could be a get or an assign and $e_2$ follows $e'$ in $\hat{\Xi}$.

First, assume that event $e_1$ was inserted to $\hat{\Xi}$ before a matching or a semi-matching put, $e''$. For $e_2$ to return a value different from that of $e_1$, $e_1 \rightarrow \Xi e' \rightarrow \Xi e_2$. If $e'$ is a get, then $e''$ must also precede $e'$ in $\Xi$ by $\rho_2$. Since $e'' \rightarrow \Xi e'$, $e_1$ precedes $e_2$ in $\hat{\Xi}$ by construction. If $e'$ is an assign, by an analysis similar to Case 1, $e_1 \rightarrow \Xi e'$. Therefore, $e_1$ precedes $e_2$ in $\hat{\Xi}$ also by construction.

If $e_1$ has no matching or semi-matching put, then $e_2$ must causally-match an assign $e'$ such that $e_1 \rightarrow \Xi e' \rightarrow \Xi e_2$. Since $e_1$ does not have a semi-matching put, then $e'$ also does not have a semi-matching put. So, $e_1$ and $e'$ maintained in $\hat{\Xi}$ their original relative order in $\Xi$, and since $e_2$ was inserted after $e'$, $e_1$ precedes $e_2$ in $\hat{\Xi}$.

4. both $e_1$ and $e_2$ are uses:

If both $e_1$ and $e_2$ have the same causally-matching assign or the same matching or
semi-matching get, then the construction guarantees that $e_1$ precedes $e_2$ in $\hat{\Xi}$. So, assume $e_1$ and $e_2$ were inserted after two different events $e'$ and $e''$, respectively.

If $e'$ and $e''$ are both assigns, then since $e_1 \rightarrow e_2$, $e' \rightarrow e_1 \rightarrow e'' \rightarrow e_2$. Thus, $e' \rightarrow e''$ and by a similar analysis to Case 1, $e'$ precedes $e''$ in $\hat{\Xi}$. Therefore, the construction guarantees that $e_1$ precedes $e_2$ in $\hat{\Xi}$.

If $e'$ and $e''$ are both gets, then since $e_1 \rightarrow e_2$, it follows from $\rho_3$ that $e' \rightarrow e''$. So, $e_1$ precedes $e_2$.

If $e'$ is a get and $e''$ is an assign, then $e' \rightarrow e_1 \rightarrow e'' \rightarrow e_2$ since $e_1 \rightarrow e_2$. This means that $e'$ precedes $e''$ in $\hat{\Xi}$ because if $e''$ was moved, then it must have been moved forward in $\hat{\Xi}$. Therefore, the construction guarantees that $e_1$ precedes $e_2$ in $\hat{\Xi}$.

If $e'$ is an assign and $e''$ is a get, then $e' \rightarrow e_1 \rightarrow e_2$. Rule $\rho_2$ implies that there is a matching put event to $e'$ that intervenes in $\Xi$ between $e'$ and $e''$. The construction guarantees that $e'$ precedes $e''$ in $\hat{\Xi}$ and that $e_1$ precedes $e_2$ in $\hat{\Xi}$.

### Lemma 5.2.7

$M_{JV,M}^f$ realizes Java$^\text{base}$.

**Proof:** Let $C$ be any Java$^\text{base}$ computation of any $(P,J)$ system with operation invocation in $N^b$. We show how to construct a $\Xi \in M_{JV,M}^f(P,J)$, such that the computation induced by $\Xi$ is equivalent to $C$.

By Condition 2 of Definition 5.2.4, for any $p$ the order of $O_{vis_p}$ in $(O|p \cup O_{vis_p}, \overrightarrow{L_p})$ is the same as that in $(O|w, \overrightarrow{write})$. Recall that $O|w = \bigcup_{p \in P} O_{vis_p}$. Let $(O|p \cup O_{vis_p}, \overrightarrow{L_p}) = S_0, w_1, S_1, \ldots, S_n, w_n, S_n$ where each $w_i \in O|w$ and the $S_i$'s contain only $p$'s read operations. Then, $(O|w, \overrightarrow{write})$ is $w'_1, w'_2, \ldots, w'_m$, such that there is a one-to-one function $f_p$ from the $w'_i$'s to the $w'_j$'s.

A covert write is a write whose corresponding assign does not have a matching or
semi-matching put. For each $p$, mark each covert write $o$ by $\hat{o}$. Fixate reads are identified and marked as follows. Since $(O[p \cup O_{\text{vis}, p}, L_p^o])$ is a linearization, $(O[x[p], L_x^o]) = S_0', S_1', \ldots, S_s'$, where the reads in each subsequence $S_i'$ return the same value $v_i$ which is different from $v_j$, $j \neq i$, the value returned by the reads in $S_j'$. That is, the reads in $S_i'$ are all causally related to exactly one write $w$ and all the reads causally related to $w$ are in $S_i'$. Let $S_i' = r_{i1}, r_{i2}, \ldots, r_{is}$, mark each $r_{ij}$, $j \neq 1$, by a hat. Also, mark $r_{i1}$ if it is domestic. Some unmarked reads in $(O[p], L_p^o)$ are moved backwards. Each foreign unmarked read $o$ is moved so that it immediately follows the write $o'$ by $q \neq p$ such that $o' \xrightarrow{\text{wbr}} o$. The claim is that this distorted $(O[p \cup O_{\text{vis}, p}, L_p^o])$ is still a linearization that satisfies conditions (1) and (2) of Definition 5.2.4. Obviously, it is still a linearization. Furthermore, these read movements do not violate $\xrightarrow{\text{bjpo}}$ because in the interval from $o'$ to $o$ there cannot be any writes to $x$. Moreover, all the operations in this interval that are applied to $x$ must be reads that return the value written by $o'$. Since these are foreign reads, moving them backwards does not violate $\xrightarrow{\text{bjpo}}$. Next, these distorted linearizations are used to construct a total order on all the operations $(O, L_p^o)$.

Construct $(O, L_p^o)$ by adding the $S_i$’s to $(O[w', \xrightarrow{\text{writes}}])$ each between $w'_j$ and $w'_{j+1}$, where $f_p(w_i) = w'_j$. Note that this construction guarantees that $(O, \xrightarrow{\text{bjpo}}) \subseteq (O, L^o_p)$.

Given $(O, L_p^o)$, construct $\Xi$ as follows. Initially $\Xi$ is empty. Iterating through $(O, L_p^o)$, consider one operation $o$ at a time. If $o$ is a marked operation, ignore it. Otherwise, append to $\Xi$ a corresponding put or get if $o$ is a write or a read, respectively. Next, add the assign and use events as follows.

For each process $p$, iterate through $p$’s computation considering one operation $o$ at a time.

1. If $o$ is a marked operation (read or write), insert into $\Xi$ a corresponding use or assign to $o$ immediately after the last inserted event by $p$ (initially, at the beginning of $\Xi$).

2. If $o$ is an unmarked write, insert into $\Xi$ a corresponding assign to $o$ immediately after the last inserted event by $p$.

3. If $o$ is an unmarked read, insert into $\Xi$ a corresponding use to $o$ immediately after
the latest of its matching or semi-matching get or the last inserted event by \( p \).

Now, we show that the constructed \( \Xi \) complies with \( I \), program order, and \( R \).

1. **Complying with \( I \):

   Each marked read has been replaced by a single use and each marked write by a single assign. So, marked operations comply with \( I \). Each unmarked write was either replaced by a single assign or by an assign-put pair.

   To see that an assign \( e \) always precedes its matching put \( e' \), let \( e'' \) be the last inserted event by \( p \) before \( e \) is inserted. Assume for the sake of contradiction that \( e'' \) was inserted after \( e' \). Since all events are immediately inserted after the last inserted event by \( p \) except for some unmarked reads, there must be a get that follows \( e' \) and the get matches or semi-matches \( e'' \). If the get matches \( e'' \), then the corresponding read to \( e'' \) must have been a foreign read. Since \( (O,\mathcal{L}) \) satisfies bjpo, the get must precede \( e' \) in \( \Xi \), a contradiction. If \( e'' \) semi-matches the get, then there must be a use that matches the get and precedes \( e'' \). The corresponding read to this use is an unmarked foreign read. Similar to the previous case, the get must precede \( e' \) in \( \Xi \). Therefore, unmarked operations also comply with \( I \).

2. **Complying with program order:

   The construction ensures that every use and assign event is inserted after the previous inserted event. Since the operations are considered incrementally in the program order sequence, \( \Xi \) extends \( (E,\pi) \).

3. **Complying with \( R \):

   First, we show that \( \Xi \) complies with \( \rho_1 \). Each assign was either inserted alone, or paired with a matching put respecting Algorithm (A.2). If a use \( e \) corresponds to a domestic read \( r \), then there must have been a write \( w \) by the same process such that \( w \xrightarrow{\text{wbr}} r \). So, it must be the case that \( w \xrightarrow{\text{prog}} r \). Thus, an assign corresponding to \( w \) must have been inserted before \( e \), complying with Algorithm (A.1). If \( e \) corresponds to a foreign read, then the construction guarantees that there is an unmarked read leading a sequence of foreign fixate reads. This unmarked read was translated into a
get, also complying with Algorithm (A.1).

The following shows that $\Xi$ complies with $\rho_2$. Let $e_1$ be $\text{assign}(t, x, v)$ and $e_2$ be $\text{get}(t, x, v?)$ such that $e_1 \xrightarrow{\Xi} e_2$. This implies that $e_1 \xrightarrow{\pi} e'_2$ where $e'_2$ is the matching use of $e_2$. So, if $o_1$ and $o_2$ are the operations that respectively correspond to $e_1$ and $e_2$, we conclude that $o_1 \xrightarrow{\text{prog}} o_2$. Since both operations are applied to $x$, $o_1 \xrightarrow{\text{bjpo}} o_2$ and their program order must be maintained in $(O, \xrightarrow{L})$. Thus in $\Xi$, the corresponding put to $e_1, e'_1$, precedes $e_2$. Since $\Xi$ complies with $I$, $e_1 \xrightarrow{\Xi} e'_1 \xrightarrow{\Xi} e_2$.

Let $o_1$ and $o_2$ be operations and $e_1$ and $e_2$ be corresponding put and get events to $o_1$ and $o_2$ respectively. Since $(O, \xrightarrow{L})$ extends $(O, \xrightarrow{\text{bjpo}})$, the construction guarantees that if $o_1 \xrightarrow{\text{bjpo}} o_2$, then $e_1 \xrightarrow{\Xi} e_2$. In particular, $\rho_3$ is satisfied.

Finally, rule $\rho_{11}$ is satisfied as shown by the following argument. First, we show that every instantiation $\text{get}(t, x, v? \leftarrow v)$ takes place because the most recent put event that precedes the get in $\Xi|x$ is put$(s, x, v)$. The corresponding operations to the mentioned get and put events, $o_1$ and $o_2$ respectively, must be unmarked. Furthermore, the get immediately follows the put because $o_2$ was moved backwards to immediately follow $o_1$ before the construction of $(O, \xrightarrow{L})$.

For a use that corresponds to a local marked read $o_1 \in O|p|x|w$, the most recent write $o_2 \in O|p|x|r$ that precedes $o_1$ in $(O, \xrightarrow{L})$ is such that $o_2 \xrightarrow{\text{wbr}} o_1$. It follows from the construction that this use instantiates its value with that written by the most recent assign that precedes the use in $\Xi|x|p$. The argument is also trivial for a use that matches or semi-matches a get.

### Non-Terminating Computations

The construction in the proof of Lemma 5.2.6 takes into consideration non-terminating systems. When the run $\Xi$ is paused, it may include assigns that do not yet have matching or semi-matching puts. The write operation corresponding to an assign that is “so far”
covert is included in \((O|w, \overset{writes}{\rightarrow})\). When \(\Xi\) is resumed and paused again, the matching or semi-matching put for that assign might occur and the assign must be moved forward, according to the construction procedure in the proof of Lemma 5.2.6. Consequently, the corresponding write in \((O|w, \overset{writes}{\rightarrow})\) must be moved to cope with this change. Although this movement will make the new writes order inconsistent with the old one, the satisfying sequences still adhere to Definition 3.6.1 because the moved write was an incomplete operation and was not part of the determined satisfying sequences until the put occurred. Note also that the order of such a write in a process’s view \((O|p \cup O_{vis}, \overset{L_p}{\rightarrow})\) will be also updated such that Condition 2 of Definition 5.2.4 still holds. For the rest of the operations (complete or determined operations), the construction procedure ensures that Definition 3.6.1 is satisfied. Finally, the construction in the realizes direction applies to non-terminating computations as well.

### 5.2.4 Non-Operational Java with Synchronization Operations

Java programmers use the synchronized constructs which are translated by the compiler into lock and unlock events. When a method or a sequence of statements is declared synchronized, the compiler requires acquiring a lock (represented by the lock event) before executing those “guarded” statements. The lock is released (represented by the unlock event) after the execution of these statements is completed. Therefore, for each lock there is a unique matching unlock.

Let \(O|l\) and \(O|u\) be the subsets of \(O\) denoting lock and unlock operations\(^4\), respectively. The collection of both lock and unlock operations is called the set of synchronization operations and denoted \(O|s = O|l \cup O|u\). By convention, \(O|s \subseteq O|w\). Let \(O|v\) denote the subset of \(O\) applied to volatile variables. Since operations on volatile variables are reads and writes, \((O, \overset{bfpq}{\rightarrow})\) applies to \(O|v\). A read operation which is applied to a volatile variable is called a volatile read.

\(^4\)Again, the font distinguishes lock and unlock events from lock and unlock operations. Furthermore, it is always clear from the context if the reference is made to a lock variable or lock operation.
**Java Partial Program Order:**

Define the *Java partial program order*, denoted \((O, \overset{jpo}{\rightarrow})\) by \(o_1 \overset{jpo}{\rightarrow} o_2\) if \(o_1 \overset{prog}{\rightarrow} o_2\) and one of the following holds:

1. **same-object**: \(o_1, o_2 \in O|x|,\) for some \(x \in J,\)

2. **read-before-write**: \(o_1\) is a foreign or volatile read and \(o_2 \in O|w|,\)

3. **synch-before**: \(o_1 \in O|l|\) or \((o_1 \in O|u|\) and \(o_2 \in O|w|),\)

4. **synch-after**: \(o_2 \in O|u|\) or \((o_1 \in O|r|\) and \(o_2 \in O|l|),\)

5. **two-volatiles**: \(o_1, o_2 \in O|v|,\) or

6. **transitivity**: there is an operation \(d'\) such that \(o_1 \overset{jpo}{\rightarrow} d' \overset{jpo}{\rightarrow} o_2.\)

The same-object and read-before-write conditions of \((O, \overset{jpo}{\rightarrow})\) are similar to those of \((O, \overset{bipo}{\rightarrow})\). However, \((O, \overset{jpo}{\rightarrow})\) enforces further orderings on a volatile read before a write. The intuition behind this is that like foreign reads, volatile reads are required to get values from main memory as indicated by \(\rho_8\). The synch-before condition requires a lock followed by any operation in program order to maintain this order in capturing rule \(\rho_6\). The second part of synch-before relaxes the unlock before read program order. This is because the get of the read might still be issued before the unlock is performed. However, this situation cannot happen if the subsequent operation to the unlock is a write because the put corresponding to the write must come after the assign which comes after the unlock. Similarly, synch-after requires all operations preceding an unlock in the program to maintain this order in \(\overset{jpo}{\rightarrow}\). Rule \(\rho_5\) forces any write to complete before the unlock is performed. For a read, the get, if it exists, must precede the use which precedes the lock. The second part of synch-after relaxes the write before lock program order. The reason is that even though a lock flushes a thread’s working memory, the writes that are “in transit” to main memory might complete after the lock is acquired. However, this is not possible if the preceding operation to the lock is a read. Rule \(\rho_4\) requires a total order on all the lock and unlock events which is consistent with program order. This is captured by the combination
of synch-after and synch-before conditions. Finally, the volatile-before-volatile condition captures $\rho_{10}$.

Guaranteed Visibility

As indicated in the previous section, synchronization operations enlarge the set of visible write operations. Unlike base write operations, it can always be determined when lock, unlock, and write operations applied to volatile variables have completed. Define $\text{SYNCH-VIS} = O|s$ and $\text{VOLATILE-VIS} = O|v|w$. Define $O_{vis_p} = \text{DIRECT-VIS}(p) \cup \text{INDUCED-VIS}(p) \cup \text{SYNCH-VIS} \cup \text{VOLATILE-VIS} \cup O|w|p$.

**Definition 5.2.8** Let $O$ be all the operations of a computation $C$ of a multiprocess system $(P, J)$. Then $C$ is Java if there is a total order $(O|w, \text{writes} \rightarrow)$ and $\forall p \in P$ there is a linearization $(O|p \cup O_{vis_p}, L_p \rightarrow)$ satisfying:

1. $(O|p \cup O_{vis_p}, i_{po} \rightarrow) \subseteq (O|p \cup O_{vis_p}, L_p \rightarrow)$, and
2. $(O_{vis_p}, L_p \rightarrow) = (O_{vis_p}, \text{writes} \rightarrow)$.

**Theorem 5.2.9** $M_{JVM}$ exactly implements Java.

We prove this theorem by proving each direction in a separate lemma. The details that are much like those of the proof of Theorem 5.2.5 are omitted.

**Lemma 5.2.10** $M_{JVM}$ provides Java.

**Proof:** Let $C_\Xi$ be the computation of $(P, J)$ induced by $\Xi$. Let $O$ be all the operations that result from $C_\Xi$. We show that $C_\Xi$ satisfies Java.

The sequence $\hat{\Xi}$ is constructed from $\Xi$ exactly as we did for Lemma 5.2.7.

The order $(O|w, \text{writes} \rightarrow)$ is induced by the corresponding assign, lock, and unlock event in $\hat{\Xi}$.

The set $O_{vis_p}$ is defined similarly to Lemma 5.2.7, but also contains VOLATILE-VIS and SYNCH-VIS. The order $(O|p \cup O_{vis_p}, L_p \rightarrow)$ is induced by the corresponding use, assign, lock, and unlock events in $\hat{\Xi}$. So, $(O_{vis_p}, L_p \rightarrow) = (O_{vis_p}, \text{writes} \rightarrow)$ by construction.

---

5The rest of the requirements of $\rho_4$ has been embedded in the definition of the Java lock variables.
What remains to show is that \((O|p \cup O_{\text{vis}}p, \xrightarrow{L_p})\) is a linearization that extends \(j_{po}\). We will omit in the following discussion the relative order of base operations with respect to each other (this has been proved in Lemma 5.2.7). Instead, we concentrate on the order of \(O|s\) and \(O|v\) in relation to each other as well as to base operations.

\((O|p \cup O_{\text{vis}}p, \xrightarrow{L_p})\) is a linearization:

The validity of lock and unlock operations follows trivially from \(\rho_4\). For read and write operations the argument is similar to that in Lemma 5.2.7.

\((O|p \cup O_{\text{vis}}p, \xrightarrow{j_{po}}) \subseteq (O|p \cup O_{\text{vis}}p, \xrightarrow{L_p})\):

The argument for the same-object and read-before-write conditions is the same as for Lemma 5.2.7.

The first part of the synch-before condition \((o_1 \in O|l)\) follows from rule \(\rho_6\). The second part \((o_1 \in O|u\) and \(o_2 \in O|w)\) follows because the issue order is maintained between the corresponding unlock and assign events in \(\Xi\). Since the matching get follows the assign, this order is still maintained even if the assign was moved forward. The synch-after condition follows from \(\rho_5\) and a similar argument to the synch-before case.

Finally, the two-volatiles condition follows from rules \(\rho_8\) and \(\rho_9\).

\[\text{Lemma 5.2.11} \ M_{JV M} \text{ realizes Java.}\]

\[\text{Proof:} \ \text{Let } C \text{ be any Java computation of any } (P,J) \text{ system. We show how to construct a } \Xi \in M_{JV M}(P,J), \text{ such that the computation induced by } \Xi, C_\Xi, \text{ is equivalent to } C.\]

By Condition 2 of Definition 5.2.8, for any \(p\) the order of \(O_{\text{vis}}p\) in \((O|p \cup O_{\text{vis}}p, \xrightarrow{L_p})\) is the same as that in \((O|w, \xrightarrow{\text{write}})\). Recall that \(O|w = \cup_{p \in p} O_{\text{vis}}p\). Let \((O|p \cup O_{\text{vis}}p, \xrightarrow{\text{write}})\) = \(S_0, s_1, S_1, \ldots, S_n, s_n, S_n\) where each \(s_i \in O|s\) and the \(S_i\)'s contain only \(p\)'s read and write operations. Then, \((O|w, \xrightarrow{\text{write}})\) is \(S'_0, s_1, S'_1, \ldots, S'_n, s_n, S'_n\), where the \(S'_i\)'s only contain write operations. Precisely, \(S'_i = S_i|w\).

Operations are marked as in Lemma 5.2.7, with the following difference: (1) an operation that is applied to a volatile variable is unmarked, and (2) if \(w \in O|w|p\) and \(w \xrightarrow{\text{wbr}} r\) where \(r \in O|p\) and there is an \(l \in O|l|p\) such that \(w \xrightarrow{\text{prog} l prog} r\), then \(w\) is unmarked and \(r\) is treated as a foreign read. The construction of \((O, \xrightarrow{L})\) is similar to Lemma 5.2.7. The only
additional requirement is that the order of $s_i$’s in $(O|w_{\text{writes}})$ is maintained in $(O, \rightarrow^L)$.

Given $(O, \rightarrow^L)$, construct $\Xi$ as follows. Initially $\Xi$ is empty. Iterating through $(O, \rightarrow^L)$, consider one operation $o$ at a time. If $o$ is a marked operation, ignore it. Otherwise, append to $\Xi$ a corresponding lock, unlock, put or get if $o$ is a lock, unlock, write, or a read operation, respectively. Next, add the assign and use events as follows.

For each process $p$, iterate through $p$’s computation considering one operation $o$ at a time.

1. If $o$ is a lock or unlock operation, then the corresponding lock or unlock event in $\Xi$ is considered to be the last inserted event by $p$ (initially, at the beginning of $\Xi$).
2. If $o$ is a marked operation (read or write), insert into $\Xi$ a corresponding use or assign to $o$ immediately after the last inserted event by $p$.
3. If $o$ is an unmarked write, insert into $\Xi$ a corresponding assign to $o$ immediately after the last inserted event by $p$.
4. If $o$ is an unmarked read, insert into $\Xi$ a corresponding use to $o$ immediately after the latest of its matching or semi-matching get or the last inserted event by $p$.

Now, we show that the constructed $\Xi$ satisfies $I$, program order, and $R$.

1. **Complying with $I$:**
   The lock and unlock operations were replaced by corresponding lock and unlock events, complying with $I$. The read and write operations comply with $I$ as shown in Lemma 5.2.11.

2. **Complying with program order:**
   The construction ensures that every lock, unlock, use, and assign event is inserted after the previous inserted event. Since the operations are considered incrementally in the program order sequence, $\Xi$ extends $(E, \rightarrow^\pi)$.

3. **Complying with $R$:**
   We show compliance with rules $\rho_4$ to $\rho_{10}$. For the rest of the rules, Lemma 5.2.11 applies.
Rule $\rho_4$ is obvious because for any $p$, $(O|p \cup O_{vis_p}, \overset{L_p}{\to})$ is a linearization which contains all of $O|s$.

Since $(O, \overset{L}{\to})$ extends $(O, \overset{jpo}{\to})$ rules $\rho_5$ and $\rho_6$ are satisfied by construction.

Rule $\rho_7$ is also satisfied by construction. If the read following the lock is domestic and the write that is causally related to the read precedes the lock, then the construction guarantees that both the write and the read were unmarked. So, the read was replaced by get-use pair such that the get follows the lock because $(O, \overset{L}{\to})$ satisfies $\overset{jpo}{\to}$ (synch-before condition). If the read following the lock is foreign or domestic such that its causally related write follows the lock, then the argument is trivial.

Rule $\rho_8$ is trivially satisfied by construction. Since $(O, \overset{L}{\to})$ maintains program order for operations on the same object, it follows from the construction that $\rho_9$ is satisfied. Finally, $\rho_{10}$ follows from the volatile-before-volatile condition of $\overset{jpo}{\to}$.

In what follows, we will only refer to $(O, \overset{jpo}{\to})$. When the discussion involves only base operations $(O, \overset{jpo}{\to})$ can be thought of as $(O, \overset{b\text{jpo}}{\to})$.

5.3 Discussion

It is obvious that Java$_2$ and Java$_1$ are special cases of Java$_{base}$.

**Theorem 5.3.1** Java$_2$ (consequently, Java$_1$) is strictly stronger than Java$_{base}$.

**Proof:** For all $p$, set $O_{vis_p}$ to $O|w$. Definition 5.2.4 becomes identical to Definition 5.2.2. Moreover, Computation 13 is Java$_{base}$ but not Java$_2$.

The following corollary follows immediately from the previous theorem.

**Corollary 5.3.2** $M^3_{jVM}$ realizes Java$_2$ (consequently, Java$_1$).

The Java definition is very complex, while the Java$_2$ (or Java$_1$) definition is elegant and simple. If $O_{vis_p}$, the visible operations to $p$, can be guaranteed to be $O|w$, then Java will be
equivalent to Java2. One way to achieve this is to implement a “cache consistency” protocol within JVM which keeps the working memories periodically consistent. Such a protocol must guarantee two things:

- Each assign is eventually put back to main memory.
- There must be a time window that restricts the fixate phase of a thread.

Chapters 6 and 7 will establish that Java_{\text{base}} is incapable of supporting any form of coordination between threads. There are two ways by which programmers can force JVM to behave more strongly than Java_{\text{base}}. The first is to declare every shared variable in a Java program as volatile. This is a very expensive option which might disallow several optimizations. The second is to practice declaring methods as synchronized all the time.

The problems arising from operation invisibility can be avoided in the context of terminating computations. For instance, had Computation 13 been finite, then for some finite \(i\), \(\{r(x)0\}^i w(x)1\) is a linearization which satisfies Java_1. Such a linearization implies that \(w(x)1\) took place after \(p\) has terminated. Although this is not an exact description of the semantics of the history of the mentioned computation, it does not alter any future conclusion because the computation has terminated.

**Theorem 5.3.3** For terminating computations, Java_{\text{base}} is equivalent to Java_2.

Theorem 5.3.1 proves one direction of the above theorem. The second direction is proved in the next lemma.

**Lemma 5.3.4** For terminating computations, Java_{\text{base}} is stronger than Java_2.

**Proof:** Let \(C\) be a finite computation satisfying Definition 5.2.4. Denote the total order \(\langle O[p \cup O_{\text{vis}}_p, \overrightarrow{L}_p] \rangle\) by \(\overrightarrow{L}(p)\), which is guaranteed by Definition 5.2.4. We build the linearizations \(\langle O[p \cup O[w, \overrightarrow{L}_p]\rangle\), denoted \(\overrightarrow{L}(p)\), from \(\overrightarrow{L}(p)\) and show they satisfy Definition 5.2.2.

Initially, set \(\overrightarrow{L}(p)\) to \(\overrightarrow{L}(p)\). If \(O_{\text{vis}}_p = O[w]\), then the case is trivial. Otherwise, \(O_{\text{vis}}_p \subset O[w]\) and there exists a nonempty set of writes that are invisible to \(p\), \(O_{\text{inv}}_p = O[w] \setminus O_{\text{vis}}_p\).

Let \((O[w, \overrightarrow{writes}]\) be \(w_1, w_2, \ldots, w_n\), and let \(i\) be the smallest index in \((O[w, \overrightarrow{writes}]\) such that \(w_i \in O_{\text{inv}}_p\). Locate a write \(w_j\) in \(\overrightarrow{L}(p)\) where \(j\) is the smallest index in \(\overrightarrow{L}(p)\) such that
$i < j$ (note that $w_j \in O_{vis_p}$). Insert $w_i$ into $\hat{L}(p)$ immediately before $w_j$. If both $w_i$ and $w_j$ are to the same variable, no further action is needed. Otherwise, certain reads in $\hat{L}(p)$ will be moved as follows.

Let $\hat{L}(p)$ be $S_1, w_1, S_2, w_2, \ldots, S_j, w_i, w_j, \ldots, S_n, w_n$, where the $S_i$ are finite sequences of reads. Let $w_k$ be the first write in $\hat{L}(p)$ such that $k > i$ and $w_k$ and $w_i$ are to the same variable, say $x$. Move $(S_{j+1}, S_{j+2}, \ldots, S_k) | x$ to the place sandwiched by $S_j$ and $w_i$. The procedure is repeated for the next smallest $i$.

The resulting $\hat{L}(p)$ are linearizations. Initially, $\hat{L}(p)$ are linearizations because they are set to $L(p)$. If validity has been violated by the construction, then $w_i$ and $w_j$ must be to two different variables. Otherwise, it could not have been violated because $w_i$ was inserted immediately before $w_j$ in $\hat{L}(p)$.

Thus, for validity to be violated, there must be a read, $r$, of $x$ returning a value that is different from the value written by $w_i$ and $r$ is between $w_i$ and $w_k$ in $\hat{L}(p)$. However, by construction there are no reads to $x$ in this interval.

We still need to show that $\hat{L}(p)$ is consistent with $(O, \rightarrow_{jpo})$. Since $L(p)$ is consistent with $(O, \rightarrow_{jpo})$, we need to argue that the above construction did not violate it. Note that $(O, \rightarrow_{write})$ maintains $(O, \rightarrow_{jpo})$ by Definition 5.2.4, and that our construction maintains $(O, \rightarrow_{write})$. So, we need only consider operations in $O | r$. That is, we need only show that the movement of $(S_{j+1}, S_{j+2}, \ldots, S_k) | x$ did not violate $(O, \rightarrow_{jpo})$.

Note that such a movement is moving only reads backwards. However, $\rightarrow_{jpo}$ could be violated only if $\rightarrow_{prog}$ is violated between operations on the same object. This cannot happen because $(S_{j+1}, S_{j+2}, \ldots, S_k) | x$ precedes $w_k$ and there is no other write to $x$ in the interval between $w_i$ and $w_k$. Furthermore, the original order of the reads in $(S_{j+1}, S_{j+2}, \ldots, S_k) | x$ is not affected by such a movement.

**Corollary 5.3.5** For terminating computations, Java base is equivalent to Java$_1$. 


5.4 Comparisons

5.4.1 Java versus Coherence

Gontmakher et al. [32, 33] argue that Java is Coherent. Their proof relies on the regular expression $R$, which is an elegant distillation of the rules for a single Java process, on a per-location basis (Appendix C Section C.1 Rules 2, 3, 4 and 5).\(^6\)

\[
R \left\{ \begin{array}{l}
\text{Order} = \text{get-block} \mid \text{put-block}^* \\
\text{get-block} = \text{get}(\text{use})^* \\
\text{put-block} = \text{assign}(\text{use} \mid \text{assign})^* \text{put}(\text{use})^*
\end{array} \right.
\]

They claim that the events corresponding to the operations of a single process to a fixed variable satisfy $R$. In fact, if an $\text{assign}(t,x,v)$ is not followed by a $\text{get}(t,x,u)$, then a subsequent put is optional. Thus, if a put-block is not followed by a get-block, the put-block need not contain a put. A modification of $R$ to $\tilde{R}$ captures this more general situation ($\lambda$ denotes the empty string.)

\[
\tilde{R} \left\{ \begin{array}{l}
\text{Order} = \text{get-block} \mid \text{put-block}^* \text{tail-block} \\
\text{get-block} = \text{get}(\text{use})^* \\
\text{put-block} = \text{assign}(\text{use} \mid \text{assign})^* \text{put}(\text{use})^* \\
\text{tail-block} = \text{assign}(\text{use} \mid \text{assign})^* \mid \lambda
\end{array} \right.
\]

Coherence still holds for any computation such that for each variable, and for each process, the events corresponding to the operations of that process on that variable satisfy $\tilde{R}$. This can be proved by modifying the proof of Gontmakher et al. slightly so that, in the linearization, each tail-block follows every get-block and put-block. Thus we can conclude that all finite Java computations are Coherent.

Unfortunately, non-terminating Java computations are not necessarily Coherent. Computation 13 is $\text{Java}_{\text{base}}$ but it is not $\text{Java}_1$. Since Computation 13 uses only one variable, it cannot be Coherent because when there is only one variable, $\text{Java}_1$ and Coherence are the same. One important consequence of this is that $\text{Java}_{\text{base}}$ cannot support solutions to ver-

---

\(^6\)To be consistent with this chapter, we renamed $\text{store}$ to put and $\text{load}$ to get in $R$. 

---
Comparisons of the non-terminating producer/consumer coordination problem as will be discussed in Chapter 7 even though Coherence suffices to solve this problem.

**Computation 14**
\[
\begin{align*}
  p &: r(x) \land w(y) \\
  q &: r(y) \land w(x)
\end{align*}
\]

Computation 14 [32] is Coherent but not Java\textsubscript{base}. Hence, the following theorem.

**Theorem 5.4.1** Java\textsubscript{base} and Coherence are incomparable.

### 5.4.2 Java versus Other Consistency Models

Gontmakher et al. [32, 33] showed by example that

- Java\textsubscript{1} and P-RAM are incomparable.
- Java\textsubscript{1} and PC-G are incomparable.

Since their examples are finite, these same conclusions apply to Java\textsubscript{base}.

**Theorem 5.4.2** Java\textsubscript{base} and CC are incomparable.

**Proof:** It is easily confirmed that Computation 14, which is not Java\textsubscript{base}, is CC. Moreover, the Java\textsubscript{base} Computation 13 is not CC.

**Theorem 5.4.3** Java\textsubscript{base} and WO\textsubscript{base} are incomparable.

**Proof:** It is easily verified that Computation 14 is WOC\textsubscript{base}, and (the non-terminating) Computation 13 is not WOC\textsubscript{base}. Therefore, Java\textsubscript{base} and WOC\textsubscript{base}, and consequently WO\textsubscript{base}, are incomparable.

To see that PSO\textsubscript{base} is strictly stronger than Java\textsubscript{base}, imagine a situation in which the working memory in JVM mimics the behavior of the store-buffer in PSO\textsubscript{base}. Specifically, (1) every assign is paired with a put and (2) every use that follows a put by the same process on the same variable is paired with a get that follows the put. So, a use that is paired with a get in Java\textsubscript{base} simulates a load-request-load-reply pair in PSO\textsubscript{base} such that the reply event completes at main memory; otherwise, it simulates a pair that returns the value from the store-buffer.
**Theorem 5.4.4** \(PSO_{\text{base}}\) (consequently, \(TSO_{\text{base}}\)) is strictly stronger than \(Java_{\text{base}}\).

**Proof:**

Since \((O, L^{po}) \subseteq (O, pso)\), it follows that \(PSO_{\text{base}}\) is strictly stronger than \(Java_2\). Moreover, the \(Java_{\text{base}}\) Computation 13 is impossible in a \(PSO_{\text{base}}\) or \(TSO_{\text{base}}\) system.

Since \(TSO_{\text{base}}\) is strictly stronger than \(PSO_{\text{base}}\) and since \(Java_2\) is strictly stronger than \(Java_{\text{base}}\), we conclude that \(TSO_{\text{base}}\) and \(PSO_{\text{base}}\) are strictly stronger than \(Java_{\text{base}}\). □

**5.5 Summary**

We have derived a non-operational definition for Java Consistency. The definition applies to both terminating and non-terminating systems. Without the use of Java’s volatile variables and synchronized constructs, \(Java_{\text{base}}\) is the ‘weakest’ model we define in this dissertation. Probably, \(Java_{\text{base}}\) is the weakest memory consistency model in the whole literature.

---

**Figure 5.4:** Relationships between Java models and other models

\(A \rightarrow B\) indicates that model \(A\) is strictly stronger than model \(B\).

---

Although, \(Java_{\text{base}}\) and Coherence are incomparable, we shall see later that \(Java_{\text{base}}\) is
somehow “weaker” than Coherence in the sense that certain forms of process coordination are possible in Coherence but not in Java\textsubscript{base}. Figure 5.4 summarizes the comparison results.
CHAPTER 6

Critical Section Coordination

Algorithms that coordinate processes via critical sections have long been known for sequentially consistent systems [64, 50, 57, 62, 20]. However, many weaker memory models cannot provide this coordination without the use of explicit synchronization operations. In this chapter, we use the formal definitions of the memory models described in previous chapters to determine their capabilities (or in-capabilities) to support a solution to the critical section problem using only their read/write variables (henceforth, called base variables or simply variables).

6.1 Automatic Verification

Proofs in the context of weak memory consistency models can be very slippery. The major results in this and the next chapters have been confirmed by the SPIN model checker [44]. SPIN is a generic verification tool for asynchronous multiprocess systems. It supports verification of process interaction in Sequentially Consistent or Linearizable systems. A process meta language for design specification called PROMELA is part of the SPIN system. PROMELA allows the specifications to be expressed in an intuitive, program-like notation. It also includes powerful constructs, such as assertions and progress cycles, that allow verifying properties such as safety, deadlock, and starvation. PROMELA also accepts linear temporal logic correctness claims. SPIN matches the design specification versus the correctness claims by performing a partial order reduction of the search space. For an example on how to use SPIN in the context of weak memory consistency models,
6.2 Critical Section Problem

6.2.1 Problem Definition

The Mutual Exclusion Problem is the most famous and well-studied problem in concurrency since the 1960’s[64]. In this dissertation, it is called the Critical Section Problem (CSP) as in [65] to isolate Mutual Exclusion as a property of the problem rather than being the problem itself.

In CSP [64], a set of processes coordinate to share a resource. We denote a CSP problem by CSP($n$) where $n$ is the number of processes in the system. Each process has the following structure:

```
repeat
  <remainder>
  <entry>
  <critical section>
  <exit>
until false
```

Given the multiprocess system ($P$, $J$), a solution to CSP($n$), $n \geq 2$, must satisfy the following properties$^1$:

- **Mutual Exclusion:** At any time there is at most one $p \in P$ in its `<critical section>`.

- **Progress:** If at least one process is in `<entry>`, then eventually one will be in `<critical section>`.

- **Fairness:** If $p \in P$ is in `<entry>`, then $p$ will eventually be in `<critical section>`.

Let $A$, $P_m$, and $D$ be an algorithm, problem, and memory consistency model, respectively. Then, $A$ solves $P_m$ for $D$ if for every system $S$ that delivers $D$, $A$ solves every instance of $P_m$ on $S$.

---

$^1$Other forms of defining solution properties are possible as is given by Attiya et al.[12]. The form chosen here suffices to serve the purpose of this dissertation.
6.2. Critical Section Problem

| Algorithm                  | Year | \(|P|\) | Variables | \(\text{flag Values}\) | Fairness Delay |
|----------------------------|------|--------|-----------|--------------------------|----------------|
| Dekker’s                   | 1965 | \(n = 2\) | \(n + 1\) | 2                        | \(\infty\)     |
| Dijkstra’s                 | 1965 | \(n \geq 2\) | \(n + 1\) | 3                        | \(\infty\)     |
| Knuth’s                    | 1966 | \(n \geq 2\) | \(n + 1\) | 3                        | \(2^{n-1} - 1\) |
| De Bruijn’s                | 1967 | \(n \geq 2\) | \(n + 1\) | 3                        | \(\frac{(n^2 - n)}{2}\) |
| Eisenberg and MacGuire’s   | 1972 | \(n \geq 2\) | \(n + 1\) | 3                        | \(n - 1\)      |
| Burns’                     | 1981 | \(n \geq 2\) | \(n + 1\) | 2                        | \(\infty\)     |
| Peterson’s                 | 1981 | \(n \geq 2\) | \(2n - 1\) | 2                        | \(\frac{(n^2 - n)}{2}\) |

Figure 6.1: Well known CSP algorithms for SC

6.2.2 SC Solutions

As we have pointed out earlier, CSP solutions for SC and stronger systems have been known since the 1960s. In fact, as shown by Lamport [50], even one single-reader, single-writer bit is sufficient for an unfair solution to CSP, as long as accesses to these seemingly weak objects are guaranteed to be SC. Lamport also shows that three such bits suffice for fair solutions.

Raynal [64] provides an extensive survey of CSP algorithms. A collection of algorithms that we refer to in this chapter, are quoted in Appendix D and listed in Figure 6.1. Although they are different, these solutions share a common property (except Peterson’s Algorithm). They all use the same number of variables: one multi-writer (\(\text{turn}\)) and \(n\) single-writers. Furthermore, each process writes and reads \(\text{turn}\), and each process \(i\) is associated with the single-writer \(\text{flag}[i]\). Every process \(j \neq i\) reads \(\text{flag}[i]\).

Figure 6.1 characterizes each algorithm by four attributes: number of processes \(|P| = n\), number of variables, number of values that a \(\text{flag}\) variable can be assigned, and delay. The delay is an upper bound on the total number of entries to the critical section by all processes before a certain process gets the opportunity to enter its critical section. When there is no upper bound on the fairness delay (\(\infty\)), the algorithm is prone to starvation, and is thus unfair.
6.3 Impossibilities

Now, the CSP is re-addressed in systems that are weaker than SC. Minimum requirements for weak memory consistency models to provide a solution to CSP without the use of strong synchronization primitives are investigated. Each memory model is addressed in a separate section, but the proofs template is common to all sections and is given first.

6.3.1 Template for Proofs

We will use the partial computations 15, 16, and 17 defined below. First, we assume for the sake of contradiction that there exists an algorithm $A$ that solves CSP($n$) for a certain memory consistency model $D$ for some $n \geq 2$. This solution must work when exactly two processes, say $p$ and $q$, are participating and the rest engaging in $<\text{remainder}>$. If $A$ runs with $p$ in $<\text{entry}>$ and with $q$ in $<\text{remainder}>$, then by the Progress property, $p$ must enter its $<\text{critical section}>$ producing a partial computation of the form of Computation 15, where $\lambda$ denotes the empty sequence and $o_i^p$ denotes the $i^{th}$ operation of $p$.

$$\text{Computation 15} \begin{cases} p : o_1^p, o_2^p, \ldots, o_k^p & (p \text{ is in its } <\text{ critical section}> ) \\ q : \lambda & \end{cases}$$

Similarly, if $A$ runs with $q$’s participation only, Computation 16 results which is also guaranteed to exist by Progress.

$$\text{Computation 16} \begin{cases} p : \lambda \\ q : o_1^q, o_2^q, \ldots, o_l^q & (q \text{ is in its } <\text{ critical section}> ) \end{cases}$$

Now, with the existence of computations 15 and 16, we build Computation 17 where both $p$ and $q$ are participating, but both are in their $<\text{critical section}>$. By assumption, both computations 15 and 16 satisfy the conditions of $D$. If we can show that Computation 17 also satisfies $D$, we get the desired contradiction, concluding that $A$ does not exist.

$$\text{Computation 17} \begin{cases} p : o_1^p, o_2^p, \ldots, o_k^p & (p \text{ is in its } <\text{ critical section}> ) \\ q : o_1^q, o_2^q, \ldots, o_l^q & (q \text{ is in its } <\text{ critical section}> ) \end{cases}$$
None of the arguments in the following theorems depends on the Fairness property, so the impossibilities include unfair solutions as well. Furthermore, none of these arguments depends on the size of variables. So, these results apply even to unbounded variables as well.

### 6.3.2 Coherence

**Theorem 6.3.1** There does not exist an algorithm that solves CSP(n) for Coherence, even if \( n = 2 \).

**Proof:** Assume that there is an algorithm \( A \) that solves CSP(n) for Coherence. Then computations 15 and 16 exist. We show that Computation 17 satisfies Coherence. For any \( x \in J \), let \( o^p_j \) be the first write to variable \( x \) by \( p \). Then,

\[
(O|x, \overset{L_\alpha}{\rightarrow}) = \langle (o^p_1, ..., o^p_{j-1})|x, (s^q_1, ..., s^q_t)|x, (o^p_j, ..., o^p_k)|x, j \rangle
\]

is a total order on \( O|x \) that clearly preserves \( \overset{prog}{\rightarrow} \). Each of the segments \( (o^p_1, ..., o^p_{j-1})|x \), \( (s^q_1, ..., s^q_t)|x \) and \( (o^p_j, ..., o^p_k)|x \) occur in a possible computation. The first two are concatenated before there has been any write to \( x \) by \( p \) and the first operation of the last segment is a write to \( x \), which obliterates any previous change to \( x \) by \( q \). Thus, the sequence is a linearization of \( O|x \), showing that Computation 17 is Coherent.

Therefore, our assumption must have been in error and \( A \) does not exist.

### 6.3.3 Causal Consistency

**Theorem 6.3.2** There does not exist an algorithm that solves CSP(n) for CC, even if \( n = 2 \).

**Proof:** Assume that there is an algorithm \( A \) that solves CSP(n) for CC. Then computations 15 and 16 exist. We show that Computation 17 satisfies CC.
In computations 15 and 16, $(O, wbr) \subseteq (O, prog)$ because only one process is participating in each computation. Thus, in these computations, $(O, causal) = (O, prog)$. Consider the linearizations for $p$ and $q$:

$$(O|p \cup O|w, L_p) = \langle (o_1^p, \ldots, o_k^p), (o_1^q, \ldots, o_l^q) | w \rangle$$

$$(O|q \cup O|w, L_q) = \langle (o_1^q, \ldots, o_l^q), (o_1^p, \ldots, o_k^p) | w \rangle$$

Clearly, each preserves prog as required. Also, each is a linearization because the first part (for instance, $(o_1^p, \ldots, o_k^p)$) corresponds to a possible computation, and the second part (for instance, $(o_1^q, \ldots, o_l^q) | w$) contains only writes. Thus, Computation 17 is CC.

Therefore, our assumption must have been in error and $A$ does not exist.

6.3.4 P-RAM

Theorem 6.3.3 There does not exist an algorithm that solves CSP(n) for P-RAM, even if $n = 2$.

Proof: Since P-RAM is strictly weaker than CC, the proof follows from Theorem 6.3.2.

We have shown in Chapter 2 that a system that satisfies both Coherence and P-RAM is not necessarily PC-G. Since Computation 17 satisfies both P-RAM and Coherence, we conclude that even when combined together P-RAM and Coherence do not suffice to solve CSP. However, as shown by Ahamad et al.[9] and confirmed in Section 6.4, PC-G is sufficient to solve CSP.

6.3.5 Weak Ordering

Theorem 6.3.4 There does not exist an algorithm that solves CSP(n) for WO base, even if $n = 2$.

Proof: It was shown in Section 2.3.1 that WO base is strictly weaker than Coherence. Therefore, the theorem follows from Theorem 6.3.1.
6.3.6 SPARC Consistency

Without the use of swap-atomics in TSO and store-barriers in PSO, it is impossible to solve CSP(n) as confirmed by the following two theorems.

**Theorem 6.3.5** There does not exist an algorithm that solves CSP(n) for \( TSO_{base} \), even if \( n = 2 \).

**Proof:** Assume that there is an algorithm \( A \) that solves CSP(n) for \( TSO_{base} \). Then computations 15 and 16 exist. We show that Computation 17 satisfies \( TSO_{base} \).

To see that Computation 17 satisfies \( TSO_{base} \), we imagine a situation where \( p \) and \( q \) enter their critical sections before the contents of their store-buffers are committed to main memory. So, the domestic reads by \( q \) complete at the buffer level, while foreign reads necessarily return initial values.

Specifically, \( (O, \overset{L}{\rightarrow}) \) is defined as follows. First, we build a sequence of operations \( S \). Initially, \( S = \lambda \). Set sequence \( Q \) to be \( q \)'s computation \( \langle o_1^q, o_2^q, \ldots, o_i^q \rangle \). Examine each \( o_i^q \) in \( Q \) in order from \( i = 1 \) to \( l \). If \( o_i^q \) is a foreign read, append \( o_i^q \) to \( S \) and remove it from \( Q \). When there are no foreign reads left in \( Q \), append to \( S \) \( p \)'s computation \( \langle o_1^p, o_2^p, \ldots, o_k^p \rangle \). Finally, append \( Q \) (with foreign reads removed) to \( S \).

Define \( (O, \overset{L}{\rightarrow}) \) to be \( S \). \( (O, \overset{L}{\rightarrow}) \) consists of three segments. The first consists entirely of foreign reads by \( q \), the second consists entirely of \( p \)'s computation, and the third consists entirely of \( q \)'s computation minus operations in the first segment.

To show that \( (O, \overset{L}{\rightarrow}) \) is a linearization, note that the reads in the first segment necessarily return initial values because in Computation 16 only \( q \) is participating. So, the first segment is valid. The second segment is valid because it is Computation 15 and the first segment does not contain any writes. The construction guarantees that each read left in the third segment is necessarily domestic. That is, it returns the value written by the most recent write to the same variable that precedes the read in the same segment. Therefore, \( (O, \overset{L}{\rightarrow}) \) is a linearization.

Furthermore, \( (O, \overset{tso}{\rightarrow}) \subseteq (O, \overset{L}{\rightarrow}) \). First of all, note that \( (O|p, \overset{L}{\rightarrow}) = (O|p, \overset{prog}{\rightarrow}) \). Consequently, \( (O|p, \overset{tso}{\rightarrow}) \subseteq (O|p, \overset{L}{\rightarrow}) \). Second, program order is maintained in the first seg-
ment by construction and also in the third segment. Finally, each read moved to the first
segment is foreign and returns the initial value. Therefore, such a read is not preceded in q’s
computation by any writes to the same variable. This implies that the moved reads do not
violate \((O, tso)\). Therefore, \((O, tso) \subseteq (O, L)\). Thus, Computation 17 satisfies TSO_{base}.

Therefore, our assumption must have been in error and A does not exist.

Since PSO_{base} is strictly weaker than TSO_{base}, the following theorem requires no proof.

**Theorem 6.3.6** There does not exist an algorithm that solves CSP(n) for PSO_{base}, even if
\(n = 2\).

### 6.3.7 Java Consistency

This section confirms that without the synchronized or volatile constructs, fixate and covert
invisibilities make Java too weak to support solutions to CSP(n). To show this, we prove a
stronger theorem showing that even Java_{1} is too weak to support this coordination.

**Theorem 6.3.7** There does not exist an algorithm that solves CSP(n) for Java_{1}, even if
\(n = 2\).

**Proof:** The proof follows from the proof of Theorem 6.3.5 because \((O, jpo) \subseteq (O, tso)\).

Since Java_{1} is strictly stronger than Java_{base} the following corollary is immediate.

**Corollary 6.3.8** There does not exist an algorithm that solves CSP(n) for Java_{base}, even if
\(n = 2\).

### 6.4 PC-G Solutions

None of the memory consistency models investigated in the previous section is capable of
supporting a solution to CSP without the use of hardware-based or explicit synchronization. On the other hand, Ahamad et al.[9] proved that Peterson’s algorithm [62], which was
originally developed for SC systems, solves CSP for PC-G. Since PC-G is capable of supporting such a solution without explicit synchronization, this section further investigates bounds and restrictions on these PC-G solutions.

**Theorem 6.4.1** [9] There exists an algorithm that solves CSP(2) for PC-G.

The correctness of a solution to CSP(n) follows from the solution of CSP(2). Given algorithm $A_2$ that solves CSP(2) for memory consistency model $D$, an algorithm $A_n$ that solves CSP(n) for $D$, where $n \geq 2$, can be always constructed from $A_2$ by building a tournament tree. Processes in $P$ are grouped into disjoint sets of size two each. For each set, $A_2$ is used to select a “winner”. The winners are regrouped into sets of size two, and $A_2$ can be used in this manner repeatedly until there is only one winner. Thus, the following corollary.

**Corollary 6.4.2** There exists an algorithm that solves CSP(n) for PC-G.

**SPIN Experiments**

*Peterson’s Algorithm has been verified as correct CSP solution for PC-G, confirming Theorem 6.4.1.*

### 6.4.1 Type of Variables

To further investigate minimum requirements for solving CSP in PC-G, we distinguish between *multi-writer* variables (simply, multi-writers) and *single-writer* variables (simply, single-writers). A multi-writer can be updated by any number of processes in the system, while a single-writer can be updated by exactly one designated process.

Recall that all the algorithms listed in Table 6.1 use at least one multi-writer. We show, next, that the use of multi-writers is crucial for such a solution to exist. First we need the following lemma.

**Lemma 6.4.3** In a system $(P, J)$ where $J$ consists entirely of single-writers, PC-G is equivalent to P-RAM.

**Proof:** Obviously, PC-G is stronger than P-RAM. We show that without the use of multi-writer variables, P-RAM is stronger than PC-G. Let $(O|p \cup O|w; L_p)$ and $(O|q \cup O|w; L_q)$
be linearizations for \( p \) and \( q \in P \) that are guaranteed by P-RAM. Since, for any variable \( x \in J \), there is only one process, say \( s \), that writes to \( x \), and both \( \langle O[p \cup O[w; L_p \rightarrow] \rangle \) and \( \langle O[q \cup O[w; L_q \rightarrow] \rangle \) have all these writes to \( x \) in the program order of \( s \), the order of the writes to \( x \) in \( \langle O[p \cup O[w; L_p \rightarrow] \rangle \) is the same as the order of the writes to \( x \) in \( \langle O[q \cup O[w; L_q \rightarrow] \rangle \). Therefore, the definition of PC-G (Definition 2.2.5) is satisfied.

**Theorem 6.4.4** There does not exist an algorithm that uses only single-writers and solves CSP\((n)\) for PC-G, even if \( n = 2 \).

**Proof:** Since the given algorithm only uses single-writers, PC-G and P-RAM are the same by Lemma 6.4.3. However, Theorem 6.3.3 establishes that such an algorithm does not exist for P-RAM and, thus, for PC-G.

Ahamad et al.[9] also prove that Lamport’s Bakery algorithm [48], which uses only single-writers, is incorrect for PC-G. The consequence of Theorem 6.4.4 is that any CSP solution for PC-G must use at least one multi-writer. In contrast to SC [70], the following corollary also follows from theorems 6.4.4 and 6.4.1.

**Corollary 6.4.5** In a system that delivers PC-G, multi-writers cannot be implemented from single-writers.

### 6.4.2 Number of Variables

After showing that at least one multi-writer is required by a CSP solution for PC-G, the natural question that arises is what is the minimum number of variables needed to solve CSP\((n)\) for PC-G? In this section, we derive a lower bound on the number and type of variables needed to solve CSP\((n)\) for PC-G. The bound is proven tight for the case when \( n = 2 \).

**Theorem 6.4.6** There does not exist an algorithm that uses fewer than \( n \) single-writers and exactly one multi-writer and solves CSP\((n)\) for PC-G, for any \( n \geq 2 \).

**Proof:** Assume that there is an algorithm \( A \) that uses fewer than \( n \) single-writers and one multi-writer and solves CSP\((n)\) for PC-G. Since there are fewer than \( n \) single-writers and \( n \)
processes, the pigeon-hole principle ensures that there is at least one process that does not write to any single-writer variable. Let $p$ be such a process and $q$ be any other process and consider computations 15 and 17. In Computation 15, $p$ does not write to any single-writer. We show that Computation 17 satisfies PC-G.

Let $o_i^q$ be $q$’s first write to the multi-writer. The following are the required PC-G linearizations for $p$ and $q$.

\[
(O|p \cup O|w, L_p) = \langle o_1^p, \ldots, o_k^p, (o_1^q, \ldots, o_l^q) | w \rangle
\]

\[
(O|q \cup O|w, L_q) = \langle o_1^q, \ldots, o_{k-1}^q, (o_1^p, \ldots, o_k^p) | w, o_1^q, \ldots, o_l^q \rangle.
\]

Both linearizations maintain program order. Moreover, $p$’s linearization is valid because it consists of Computation 15 followed by only writes by $q$. Also, $q$’s linearization is valid because the segment $o_1^q, \ldots, o_{k-1}^q$ does not contain any writes to the multi-writer. Since $p$ does not write to the single-writer, the segment $(o_1^p, \ldots, o_k^p) | w$ contains only writes to the multi-writer. Finally, the segment $o_1^q, \ldots, o_l^q$ starts with a write to the multi-writer overwriting any changes the segment $(o_1^p, \ldots, o_k^p) | w$ caused.

Finally, both linearizations agree on the order of writes for each variable (Condition 2 of Definition 2.2.5). Each linearization lists $p$’s writes to the multi-writer then $q$’s. Since only $q$ writes to a single-writer, the two linearizations also agree on the order of this variable.

We tighten the above theorem in the case where $n = 2$.

**Theorem 6.4.7** Two variables are insufficient to solve CSP(2) for PC-G.

**Proof:** Assume that there is an algorithm $A$ that uses exactly 2 variables (even both multi-writers) and solves CSP(2) for PC-G. Then, computations 15 and 16 exist. We show that Computation 17 satisfies PC-G.

Given that the algorithm uses two multi-writer variables, each process must write to each variable. Let these variables be $x$ and $y$. We divide $p$’s and $q$’s computations (of Computation 17) into subsequences as follows.

**Computation 18**

\[
\begin{align*}
    p &: S_0^p, S_1^p, \ldots, S_n^p \quad (p \text{ is in its } < \text{critical section}>) \\
    q &: S_0^q, S_1^q, \ldots, S_n^q \quad (q \text{ is in its } < \text{critical section}>)
\end{align*}
\]
The subsequences $S_p^i$ are defined as follows. The definition for $S_q^i$ is similar. Iterating through $p$’s computation, define:

1. $S_p^0$ to contain all operations from $o_1^p$ up to but not including the first write by $p$.

2. $S_p^i$, $i \neq 0$, to contain all operations from the first operation $o_j^p$ that was not included in $S_p^{i-1}$ up to but not including the first write $o_m$ such that $o_j^p$ and $o_m$ are applied to different variables.

The subsequence $S_p^0$ is either empty or consists entirely of reads returning initial values. Each subsequence $S_p^i$ $(i \neq 0)$ starts with a write and all the writes in $S_p^i$ are applied to the same variable. If the writes in $S_p^i$ are applied to $x$, $S_p^i$ is called $x$-gender; otherwise, it is called $y$-gender. Note that $S_p^i$ $(S_q^i)$ alternate in gender.

To show that Computation 18 satisfies PC-G, we consider two cases (the other two cases are symmetric).

**$S_p^1$ is an $x$-gender but $S_q^1$ is a $y$-gender:**

$(O|p \cup O|w, \frac{L_p}{w})$ and $(O|q \cup O|w, \frac{L_q}{w})$ are defined as follows.

\[
(O|p \cup O|w, \frac{L_p}{w}) = \langle S_p^0, (S_q^0)|w, S_1^p, (S_1^q)|w, S_2^p, \ldots, (S_i^q)|w, S_{i+1}^p, \ldots \rangle
\]

\[
(O|q \cup O|w, \frac{L_q}{w}) = \langle S_q^0, (S_p^0)|w, S_1^q, (S_1^p)|w, S_2^q, \ldots, (S_i^p)|w, S_{i+1}^q, \ldots \rangle
\]

Since each $S_p^i$ precedes $S_{i+1}^p$ and each $S_q^i$ precedes $S_{i+1}^q$, $(O|p \cup O|w, \frac{L_p}{w})$ and $(O|q \cup O|w, \frac{L_q}{w})$ maintain program order. They are also valid because for each $i \neq 0$, $S_p^i$ (respectively, $S_q^i$) is of the same gender as $S_{i+1}^q$ (respectively, $S_{i+1}^p$). For instance, since $S_1^q$ and $S_{i+1}^p$ are of the same gender, adding $(S_1^q)|w$ immediately before $S_{i+1}^p$ does not affect $p$’s computation because $S_{i+1}^p$ starts with a write that obliterates the changes caused by $(S_1^q)|w$.

It remains to show that Condition 2 of Definition 2.2.5 is satisfied. That is, both $(O|p \cup O|w, \frac{L_p}{w})$ and $(O|q \cup O|w, \frac{L_q}{w})$ maintain the same order on writes to $x$ (respectively, $y$). The order on the writes to $x$ in $p$’s linearization is given by:

\[
(S_1^p)|w, (S_2^p)|w, \ldots, (S_i^p)|w, (S_{i+1}^p)|w, \ldots
\]
which is the same order maintained by \( q \)'s linearization. The same applies to \( y \).

\( S_p^1 \) and \( S_q^1 \) are both \( x \)-gender

\[
(O|p \cup O|w, \overset{L_y}{\rightarrow}) \text{ and } (O|q \cup O|w, \overset{L_y}{\rightarrow}) \text{ are defined as follows.}
\]

\[
(O|p \cup O|w, \overset{L_y}{\rightarrow}) = \langle (S_{0}^{p} \mid w, S_{0}^{p} \mid w, S_{1}^{p} \mid w, S_{1}^{p} \mid w, \cdots) \rangle
\]

\[
(O|q \cup O|w, \overset{L_y}{\rightarrow}) = \langle (S_{0}^{q} \mid w, S_{1}^{q} \mid w, S_{2}^{q} \mid w, S_{3}^{q} \mid w, S_{4}^{q} \mid w, \cdots) \rangle
\]

Similar analysis to the previous case shows that these are PC-G linearizations. Thus, Computation 17 is PC-G, and our assumption must have been in error.

Since at least one multi-writer is necessary to solve CSP for PC-G, and since two multi-writers are insufficient to solve CSP(2) for PC-G, we conclude the following.

**Corollary 6.4.8** Any solution to CSP(2) for PC-G requires three variables one of which must be a multi-writer.

After recalling that Peterson’s Algorithm for CSP(2) uses exactly two single-writers and one multi-writer, we conclude the following.

**Corollary 6.4.9** Two single-writers and one multi-writer is a tight bound on the number and type of variables required to solve CSP(2) for PC-G.

### 6.4.3 On the General Case

By theorems 6.4.4 and 6.4.6, an algorithm that solves CSP(\( n \)) for PC-G must use at least \( n \) single-writers and one multi-writer. Most algorithms that solve CSP(\( n \)) for SC use exactly this number and type of variables. In particular, all the algorithms of Figure 6.1 (except Peterson’s which uses \( n \) single-writers and \( n - 1 \) multi-writers) use \( n \) single-writers and one multi-writer. Although this number of variables is a necessary requirement for a PC-G solution, we show next that most of these algorithms do not solve CSP(\( n \)) for PC-G. First, we provide some rules-of-thumb that allows us to nail down certain properties of correct solutions for PC-G. Then, these rules are used to show that Dekker’s, Dijkstra’s, Knuth’s, De Bruijn’s, and Eisenberg and MacGuire’s fail to solve CSP(\( n \)) for PC-G.
Lemma 6.4.10 Any algorithm $A$ that uses exactly $n$ single-writers and one multi-writer and solves CSP($n$) for PC-G must satisfy each of the following properties:

1. There is a bijection between single-writers and processes. Moreover, every process writes its single-writer at least once in $<\text{entry}>$.

2. Each process must write the multi-writer at least once in $<\text{entry}>$.

3. If a process writes the multi-writer exactly once in $<\text{entry}>$, then this write cannot be the last operation in $<\text{entry}>$.

4. Each process must read every other single-writer in $<\text{entry}>$.

Proof: We follow the same proof template given in Section 6.3.1.

1. Assume it is not the case; then there is at least one process, say $p$, that does not write to any single-writer. The case is very similar to Theorem 6.4.6. The linearizations there also apply.

2. Assume for the sake of contradiction that there is a process $p$ that does not write to the multi-writer before entering the $<\text{critical section}>$. The following are the PC-G linearizations for $p$ and $q$, showing that Computation 17 satisfies PC-G.

$$
(O|p \cup O|w, L^p_p) = \langle o^p_1, \ldots, o^p_k, (o^q_1, \ldots, o^q_l) | w \rangle
$$

$$
(O|q \cup O|w, L^q_q) = \langle o^q_1, \ldots, o^q_k, (o^p_1, \ldots, o^p_k) | w \rangle
$$

Obviously, both are valid and maintain program order. They also agree on the writes to the multi-writer because in each linearization, only $q$ writes to that variable.

3. Assume that a process $p$ writes the multi-writer exactly once and this write operation is $o^p_k$. Under this assumption, Computation 17 satisfies PC-G as shown by the following linearizations.

$$
(O|p \cup O|w, L^p_p) = \langle o^p_1, \ldots, o^p_{k-1}, (o^q_1, \ldots, o^q_l) | w, o^p_k \rangle
$$

$$
(O|q \cup O|w, L^q_q) = \langle o^q_1, \ldots, o^q_l, (o^p_1, \ldots, o^p_k) | w \rangle
$$
Both maintain program order and are valid. They also maintain the same order on the writes to the multi-writer, which is simply q’s writes then $o_k^p$. Note that this case is equivalent to the case where multi-writer is written in the $<\text{critical section}>$ rather than in $<\text{entry}>$.

4. Assume for the sake of contradiction that there is a process, $q$, that does not read some single-writer of another process $p$. The linearizations of Theorem 6.4.6 apply.

Corollary 6.4.11 The following CSP algorithms[64] do not solve CSP($n$) for PC-G, even if $n = 2$:

1. Dekker’s Algorithm
2. Dijkstra’s Algorithm
3. Knuth’s Algorithm
4. De Bruijn’s Algorithm
5. Eisenberg and MacGuire’s Algorithm

Proof: First, note that all these algorithms use $n$ single-writers and one multi-writer.

1. Dekker’s Algorithm: The multi-writer is only written in $<\text{exit}>$. This case follows from Lemma 6.4.10(2).

2. Dijkstra’s Algorithm: Assume without loss of generality that the multi-writer $\text{turn}$ is initially $p$. So in Computation 15, $p$ enters its $<\text{critical section}>$ without writing to the multi-writer. Therefore, this case follows from Lemma 6.4.10(2).

3. Knuth’s Algorithm: The multi-writer is only written as the last step in $<\text{entry}>$. This case follows from Lemma 6.4.10(3).

4. De Bruijn’s Algorithm: A process writes the multi-writer, $\text{turn}$, in the $<\text{exit}>$ section only. This case follows from Lemma 6.4.10(2).

5. Eisenberg and MacGuire’s Algorithm: The write to $\text{turn}$, the multi-writer, in $<\text{entry}>$ is the last operation a process does before entering the $<\text{critical section}>$. This case follows from Lemma 6.4.10(3).
Burns’ Algorithm [17], which also uses \( n \) single-writers and one multi-writer, is an unfair solution to CSP\((n)\) for PC-G. The algorithm is shown in Figure 6.2.

**Theorem 6.4.12** Burns’ Algorithm is an unfair CSP\((n)\) solution for PC-G.

**Proof:** Burns’ CSP\((n)\) Algorithm for PC-G satisfies the Mutual Exclusion and Progress properties but not the Fairness property.

- **Mutual Exclusion:** Assume for the sake of contradiction that there exists some PC-G computation of Burns’ Algorithm where two processes, say \( i \) and \( j \), execute in their \(<\text{critical section}>\) simultaneously. Then, \( i \) (respectively, \( j \)) must read \( f1\text{ag}[j] \) (respectively, \( f1\text{ag}[i] \)) to be \text{false} at line 11 before entering its \(<\text{critical section}>\) as shown by the following computation.

\[
\text{Computation 19:}\begin{cases} i: \ldots \text{r(flag[j]}\text{false < critical section>} \\ j: \ldots \text{r(flag[i]}\text{false < critical section>}
\end{cases}
\]

Note that any time a process, say \( i \), executes a \( w(f1\text{ag}[i])\text{true} \), the next operation it executes is a \( w(t\text{urn})i \). Let \( w(t\text{urn})i \) be the last write operation to \( t\text{urn} \) that \( i \) executes before entering its \(<\text{critical section}>\) (This write could be performed at line 2 or 8.) Similarly, let \( w(t\text{urn})j \) be the last write to \( t\text{urn} \) that \( j \) did before entering its \(<\text{critical section}>\) in the same computation.

Since Computation 19 satisfies PC-G, there must exist two linearizations, \((O|i \cup O|w, L^{-i})\) and \((O|j \cup O|w, L^{-j})\), such that both agree on the order of writes to \( t\text{urn} \). Without loss of generality, let \( w(t\text{urn})i \) precedes \( w(t\text{urn})j \) in both linearizations. Since \( w(t\text{urn})j \xrightarrow{L^{-j}} \text{r(flag[j]}\text{false} \) by the program order of process \( j \), \( w(t\text{urn})i \xrightarrow{L^{-i}} \text{r(flag[i]}\text{false} \). There must be some write \( w(f1\text{ag}[i])\text{true} \), such that this write is the last write by \( i \) that precedes \( w(t\text{urn})i \) in \( j \)'s view. Since \( w(t\text{urn})i \) is the last

---

**SPIN Experiments**

Corollary 6.4.11 has been confirmed by verifying that each of the listed algorithms violates the Mutual Exclusion property of CSP for PC-G.
Processes have unique identifiers from the set \( \{0, \ldots, n-1\} \), where \( n \) is the total number of processes. The algorithm is given by specifying the \(<\text{entry}>\) and \(<\text{exit}>\) sections of process \( i \), \( i \in \{0, \ldots, n-1\} \).

write by \( i \) before it enters its \(<\text{critical section}>\), \( w(\text{flag}[i] \text{true}) \) must be the last write to \( \text{flag}[i] \) before \( i \) enters its \(<\text{critical section}>\). This write also precedes \( r(\text{flag}[i]) \text{false} \) in \( j \)'s view by transitivity. Nevertheless, \( w(\text{flag}[i]) \text{true} \) is the most recent write to \( \text{flag}[i] \) that precedes \( r(\text{flag}[i]) \text{false} \) in \( (O[j \cup O[w, L_j]) \), contradicting the assumption that \( (O[j \cup O[w, L_j]) \) is a linearization. Therefore, Burns’ algorithm satisfies Mutual Exclusion for PC-G.

- **Progress:** First of all, note that if only one process is participating, then it will enter the \(<\text{critical section}>\). So let \( m \) processes, \( 1 < m \leq n \), be participating in a computation of Burns’ Algorithm, such that none of them makes progress to the \(<\text{critical section}>\). We show next that this is impossible. By PC-G, all processes must agree on the order of the writes to \( \text{turn} \), and eventually \( m - 1 \) of them will enter the body
of the while loop. At least one process will fail the test on line 4 skipping the while loop. This is because of the total order on the writes to turn that all processes agree on. Since there is at least one process, say \( j \), that does not engage in the while loop, we must have the following, where \( i \neq j \):

\[
w(\text{turn})i \xrightarrow{L_i} w(\text{turn})j \xrightarrow{L_j} r(\text{turn})j.
\]

Since \( w(\text{flag}[j])\text{true} \) precedes \( w(\text{turn})j \) in program order, we conclude the following:

\[
w(\text{flag}[j])\text{true}\xrightarrow{L_j} r(\text{flag}[j])\text{true}.
\]

In other words, lines 7 and 8 are unreachable for \( i \) unless \( j \) makes progress to <exit>. So, \( i \) is repeatedly executing lines 4 and 5 and \( w(\text{flag}[i])\text{false} \) of line 5 must eventually appear in \( (O|j \cup O|w, L_j) \), and consequently \( j \) enters its <critical section>.

- **Fairness:** To see that Burns’ algorithm is unfair for PC-G\(^2\), consider the Computation 20 which represents a starvation scenario for \( j \), where the segments enclosed by square brackets can be repeated indefinitely.

\[
\text{Computation 20} \begin{cases} 
  i : [w(\text{flag}[i])\text{true} \ w(\text{turn})i \ r(\text{turn})i \ r(\text{flag}[j])\text{false} \\
  \quad < \text{critical section} > \ w(\text{flag}[i])\text{false} ] \\
  j : w(\text{flag}[j])\text{true} \ w(\text{turn})j \ [r(\text{turn})i \ w(\text{flag}[j])\text{false} \\
  \quad r(\text{flag}[j])\text{true} ]
\end{cases}
\]

The following are PC-G linearizations for \( i \) and \( j \):

\[
(O|i \cup O|w, L_i) = (w(\text{flag}[j])\text{true}, w(\text{turn})j, w(\text{flag}[i])\text{true}, w(\text{turn})i, r(\text{turn})i, w(\text{flag}[j])\text{false}, r(\text{flag}[j])\text{false} < \text{critical section} > w(\text{flag}[i])\text{false}).
\]

\[
(O|j \cup O|w, L_j) = (w(\text{flag}[j])\text{true}, w(\text{turn})j, w(\text{flag}[i])\text{true}, w(\text{turn})i, r(\text{turn})i, w(\text{flag}[j])\text{false}, r(\text{flag}[i])\text{true}, w(\text{flag}[j])\text{false}).
\]

These linearizations can be extended by the repetition of the segments enclosed by

\[\footnote{It is known that Burns’ algorithm is unfair even for SC.}\]
square brackets in both $i$’s and $j$’s computations. This extension satisfies Definition 2.5.1 for non-terminating computations in a system that delivers PC-G.

\[
\begin{array}{|c|c|c|}
\hline
\text{SPIN Experiments} & \text{Theorem 6.4.12 has been confirmed by verifying that Burns’ Algorithm satisfies the Mutual Exclusion and Progress properties for PC-G. Moreover, Computation 20 is based on a SPIN simulation run.} \\
\hline
\end{array}
\]

6.5 Dining Philosophers Problem

A closely related problem to CSP is the Dining Philosophers Problem (DPP for short). DPP is a special case of the general resource allocation problem. It was originally introduced by Dijkstra in 1965 [57]. The problem is described in terms of $n$ philosophers\(^3\) sitting at a round dining table. On the table, there is a plate for each philosopher; however, there are only $n$ chopsticks interleaved within the plates. The life of a philosopher runs in two phases: thinking and eating. When a philosopher is thinking, he/she does not touch anything on the table. In order to eat, a philosopher needs two chopsticks—those that are immediately to his/her left and right sides. A philosopher can never use other than those two chopsticks.

Each philosopher-process is of the following form:

\[
\text{repeat} \\
< \text{think} > \\
< \text{enter} > \\
< \text{eat} > \\
< \text{exit} > \\
\text{until false}
\]

Similarly to CSP, we indicate by DPP($n$) the problem with $n$ processes. Given the multiprocess system $(P, J)$, a solution to DPP must satisfy the following properties:

- **Mutual Exclusion**: Two philosophers (in $P$) sharing the same chopstick cannot be in their $< \text{eat} >$ simultaneously.

\(^3\)Philosopher and process are used interchangeably.
• **Progress:** If at least one philosopher is in \(<\text{enter}>\), then eventually one will be in \(<\text{eat}>\).

• **Fairness:** If a philosopher is in \(<\text{enter}>\), then it will eventually be in \(<\text{eat}>\).

We can use exactly the same techniques we used to prove the CSP impossibilities to show the following theorems. Note that CSP(2) reduces to DPP(2).

**Theorem 6.5.1** There does not exist an algorithm that solves DPP(\(n\)) for: Coherence, P-RAM, CC, \(WO_{base}\), \(TSO_{base}\), \(PSO_{base}\), Java\(_1\), or Java\(_{base}\), even if \(n = 2\).

**Theorem 6.5.2** There does not exist an algorithm that uses only single-writers and solves DPP(\(n\)) for PC-G, even if \(n = 2\).

### 6.6 Summary

The need to build “higher”-performance multiprocess systems has resulted in several architectural approaches that relax SC. Such a performance gain sacrifices the programmability of these machines. We have shown in this chapter that programmers must rely on explicit synchronization instructions in order to coordinate processes via critical sections in almost all the weak memory consistency models that we have studied. The summary of these impossibilities is given in Figure 6.3.

PC-G is the only model that is capable of supporting a solution to CSP that uses only read/write variables. Unlike SC, PC-G requires at least one multi-writer and \(n\) single-writers for any CSP(\(n\)) solution. We showed this bound to be tight for \(n = 2\).

<table>
<thead>
<tr>
<th></th>
<th>Coherence</th>
<th>P-RAM</th>
<th>CC</th>
<th>(WO_{base})</th>
<th>(TSO_{base})</th>
<th>(PSO_{base})</th>
<th>Java(_1)</th>
<th>Java(_{base})</th>
<th>PC-G</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSP</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Figure 6.3:** Summary of impossibilities (×)
CHAPTER 7

Producer/Consumer Coordination

Another fundamental process coordination pattern is the producer/consumer form. This pattern is “weaker” than the CSP form in the sense that a solution to CSP implies a solution to producer/consumer coordination, but the other direction is not necessarily true. We continue investigating the capabilities of weak consistency models to support solutions to variants of this coordination pattern. In some models, certain variants of producer/consumer coordination are possible without any explicit synchronization even though a solution to CSP is impossible, as we have seen in Chapter 6.

7.1 Producer/Consumer Problem

7.1.1 Problem Definition

Producer/Consumer [20] objects are frequently used for process coordination. The producer is a process that produces items and places them in a shared structure. A consumer is a process that removes these items from the structure. We distinguish two structures the solution requirements of which vary: the set structure where the order of consumption is insignificant, and the queue structure where items are consumed in the same order as that in which they were produced.

Producers and consumers are assumed to have the following forms:
CHAPTER 7. PRODUCER/CONSUMER COORDINATION

producer: repeat
   <entry>
   <producing>
   <exit>
until false

consumer: repeat
   <entry>
   <consuming>
   <exit>
until false

Note that the form of the producer and consumer processes assumes non-terminating computations. We denote the producer/consumer queue problem as \( P_mC_n \)-queue where \( m \) and \( n \) are respectively the number of producer and consumer processes. Similarly the producer/consumer set problem is denoted \( P_mC_n \)-set.

Define the production order as follows. Item \( x \) precedes in production order item \( y \) if the production of \( x \) completes before the production of \( y \) begins. Similarly, define the consumption order as follows. Item \( x \) precedes in consumption order item \( y \) if the consumption of \( x \) completes before the consumption of \( y \) starts. Consider the following properties:

- **Safety:** Every produced item is consumed exactly once, and every consumed item is produced exactly once.

- **Progress:** If a producer (respectively consumer) is in \(<entry>\), then it will eventually be in \(<producing>\) ((respectively \(<consuming>\)) and subsequently in \(<exit>\).

- **Order:** The consumption order must respect the production order.

A solution to \( P_mC_n \)-queue must satisfy the Safety, Progress, and Order properties, while a solution to \( P_mC_n \)-set must satisfy the Safety and Progress properties only.

Note that the \( P_mC_n \)-queue is different from CSP\((m + n)\) in two aspects:

1. \( P_mC_n \)-queue requires mutual exclusive access on a per-item-basis. Producers and consumers could be in their \(<producing>\) and \(<consuming>\) sections at the same time if they are accessing different items. In CSP\((m + n)\), two processes cannot be in \(<critical\ section>\) at the same time under any circumstances.

2. A consumer cannot consume an item except if it has been produced. In CSP\((m + n)\), there is no difference in the role of processes, and nothing should stop a process from entering its \(<critical\ section>\) other than the presence of another process in its
7.1. PRODUCER/CONSUMER PROBLEM

7.1.2 SC Solutions

A solution to $P_mC_n$-queue can be constructed from a solution to CSP$(m + n)$ by protecting the queue structure in a critical section, so that it is accessed by only one process at a time. Hence, SC and PC-G (with multi-writers) systems can solve the $P_mC_n$-queue problem.

Note that the solutions to the producer/consumer problem (PCP) presented in the literature are in general CSP solutions [64, 20]. That is, these solutions tend to solve a “stronger” problem than PCP. In this chapter, we present algorithms that are dedicated to solve variants of PCP where CSP is unsolvable.

7.1.3 Process Self-Consistency

For the purpose of chapters 2 and 3, it sufficed to consider a process to be a sequence of operation invocations. For the purpose of providing algorithms that solve a given problem, however, we need to assume that this sequence of operations is given by a program and that these operations are invoked in an order that delivers the same result as the program order.

The minimum requirement, called process self-consistency, we impose on a process in consistency models that do not necessarily respect the program order is that the process must respect the data and control dependence between invocations. For instance, a process must respect the branching logic of its corresponding program. This is a reasonable assumption without which programming language constructs do not have any meaning. Moreover, systems that implement speculative execution include the necessary mechanisms to “undo” the consequences of a mistakenly taken branch.

**Definition 7.1.1** A process is self-consistent if the outcome of its isolated execution (without other processes intervention) is the same as if it executed sequentially.

For the proofs in this chapter, we assume that every process is self-consistent.
7.2 Impossibilities

The general queue problems $P_mC_n$-queue, $P_1C_n$-queue, or $P_mC_1$-queue are unsolvable in almost all the weak models mentioned in this dissertation without using explicit synchronization. The impossibility proof template is very similar to that of Section 6.3.1.

7.2.1 Template for Proofs

We will give the impossibility proofs for $P_1C_n$-queue. Those for $P_mC_1$-queue are very similar and are omitted. The proofs for $P_mC_n$-queue follow as corollaries.

Suppose there is a solution $A$ to $P_1C_n$-queue for some $n \geq 2$ for some system $(P, J)$. In particular, $A$ should work when two consumers $c$ and $d$ are participating with producer $p$. Consider (partial) Computation 21 where $p$ places item $\iota$ in the queue, then $c$ removes item $\iota$ while $d$ is idle (i.e., $\sigma^p_k$ completes before $\sigma^c_1$ starts):

\[
\text{Computation 21} \quad \begin{cases} 
  p : \sigma^p_1, \sigma^p_2, \ldots, \sigma^p_k \ (p \ has \ produced \ item \ \iota.) \\
  c : \sigma^c_1, \sigma^c_2, \ldots, \sigma^c_j \ (c \ has \ consumed \ item \ \iota.) \\
  d : \lambda.
\end{cases}
\]

By Progress, this computation must be possible in the system under consideration. Similarly Computation 22 is also possible in the system. Order guarantees that $c$ will consume item $\iota$ in Computation 21, and that $d$ will consume the same item $\iota$ in Computation 22.

\[
\text{Computation 22} \quad \begin{cases} 
  p : \sigma^p_1, \sigma^p_2, \ldots, \sigma^p_k \ (p \ has \ produced \ item \ \iota.) \\
  c : \lambda. \\
  d : \sigma^d_1, \sigma^d_2, \ldots, \sigma^d_j \ (d \ has \ consumed \ item \ \iota.)
\end{cases}
\]

The sequence for $p$ is identical in both Computation 21 and Computation 22; it represents a (partial) computation where $p$ completes production before $c$ or $d$ begin.

For each system under consideration, we will show that if Computation 21 and Computation 22 are possible then so is Computation 23.
7.2. Impossibilities

Computation 23

\[
\begin{align*}
 p & : o_1^p, o_2^p, \ldots, o_k^p \quad (p \text{ has produced item } \tau.) \\
 c & : o_1^c, o_2^c, \ldots, o_l^c \quad (c \text{ has consumed item } \tau.) \\
 d & : o_1^d, o_2^d, \ldots, o_j^d \quad (d \text{ has consumed item } \tau.)
\end{align*}
\]

However, in Computation 23 both \(c\) and \(d\) have consumed the same item, contradicting the Safety requirement. Thus, we can conclude that a solution to \(P_1C_n\)-queue is impossible in that system.

7.2.2 Coherence

Theorem 7.2.1 There does not exist an algorithm that solves \(P_1C_n\)-queue or \(P_mC_1\)-queue for Coherence when \(n \geq 2\) or \(m \geq 2\).

Proof: Assume that there is an algorithm \(A\) that solves \(P_1C_n\)-queue for Coherence. Then, computations 21 and 22 exist. We show that Computation 23 satisfies Coherence.

For each \(x \in J\), let \(o_m^c\) be the first write to variable \(x\) by \(c\) in Computation 23. Then,

\[
\langle (o_1^p, \ldots, o_k^p, o_1^c, \ldots, o_{m-1}^c, o_1^d, \ldots, o_j^d, o_m^c, \ldots, o_l^c) | x \rangle.
\]

is a total order on \(O|x\) that clearly preserves \(\langle O, \text{prog} \rangle\). Each of the segments \((o_1^p, \ldots, o_k^p, o_1^c, \ldots, o_{m-1}^c, o_1^d, \ldots, o_j^d) | x\) occurs in a possible computation; the first two are concatenated before there has been any write to \(x\) by \(c\) so that the computation of \(d\) on \(x\) is unchanged from Computation 22. The last segment begins with a write to \(x\) which obliterates any previous change to \(x\) by \(d\) so that the computation of \(c\) on \(x\) is unchanged from Computation 21. Therefore the sequence is a linearization of \(O|x\), showing that Computation 23 is Coherent.

Thus, we get the desired contradiction; \(A\) does not exist.

Corollary 7.2.2 There does not exist an algorithm that solves \(P_mC_n\)-queue or \(P_mC_n\)-queue for Coherence when \(m + n \geq 3\).
7.2.3 Causal Consistency

Theorem 7.2.3 There does not exist an algorithm that solves $P_1C_n\text{-queue}$ or $P_mC_1\text{-queue}$ for CC when $n \geq 2$ or $m \geq 2$.

Proof: Assume that there is an algorithm $A$ that solves $P_1C_n\text{-queue}$ for CC. Then, computations 21 and 22 exist. We show that Computation 23 is CC.

First of all, the $\overrightarrow{wbr}$ partial order in Computation 21 does not involve any operations by $d$. Furthermore, if $o_1\overrightarrow{wbr}o_2$ in the same computation, then $o_1 \not\in O[c]$. Thus, $(O, \overrightarrow{wbr}) \setminus (O, \overrightarrow{prog})$ only relates writes by $p$ to reads by $c$. Similarly for Computation 22, $(O, \overrightarrow{wbr}) \setminus (O, \overrightarrow{prog})$ only relates writes by $p$ to reads by $d$. Now, consider the following linearizations:

$$ (O[p \cup O[w, \overrightarrow{L_p}]) = \langle o_1^p, \ldots, o_k^p, (o_1^c, \ldots, o_j^c, o_1^d, \ldots, o_j^d) \rangle |w| $$

$$ (O[c \cup O[w, \overrightarrow{L_c}]) = \langle (o_1^p, \ldots, o_k^p) |w, o_1^c, \ldots, o_j^c, (o_1^d, \ldots, o_j^d) |w \rangle $$

$$ (O[d \cup O[w, \overrightarrow{L_d}]) = \langle (o_1^p, \ldots, o_k^p) |w, o_1^d, \ldots, o_j^d, (o_1^c, \ldots, o_j^c) |w \rangle $$

These linearizations preserve $(O, \overrightarrow{causal})$ because $(O[p \cup O[w, \overrightarrow{L_p}])$, $(O[c \cup O[w, \overrightarrow{L_c}])$, and $(O[d \cup O[w, \overrightarrow{L_d}])$ have $p$’s operations as the leading segment, and the reads by $c$ and $d$, if any, are linearized after that segment, thus, maintaining $(O, \overrightarrow{wbr})$. Also, $(O, \overrightarrow{prog})$ is clearly preserved. Finally, each is a linearization because each processor behaves exactly as it did in Computation 21 or Computation 22 where it acted alone and is followed by a segment containing only writes. Therefore, Computation 23 is CC.

Hence, $A$ does not exist.

Corollary 7.2.4 There does not exist an algorithm that solves $P_mC_n\text{-queue}$ or $P_mC_1\text{-queue}$ for CC when $m + n \geq 3$. 
7.2.4 P-RAM

**Theorem 7.2.5** There does not exist an algorithm that solves $P_1C_n$-queue or $P_mC_1$-queue for P-RAM when $n \geq 2$ or $m \geq 2$.

**Proof:** P-RAM is strictly weaker than CC. The proof follows from Theorem 7.2.3. ■

**Corollary 7.2.6** There does not exist an algorithm that solves $P_mC_n$-queue or $P_mC_n$-queue for P-RAM when $m + n \geq 3$.

7.2.5 Weak Ordering

**Theorem 7.2.7** There does not exist an algorithm that solves $P_1C_n$-queue or $P_mC_1$-queue for WO base when $n \geq 2$ or $m \geq 2$.

**Proof:** Since WO base is strictly weaker than Coherence, the proof follows from Theorem 7.2.1. ■

**Corollary 7.2.8** There does not exist an algorithm that solves $P_mC_n$-queue or $P_mC_n$-queue for WO base when $m + n \geq 3$.

7.2.6 SPARC Consistency

**Theorem 7.2.9** There does not exist an algorithm that solves $P_1C_n$-queue or $P_mC_1$-queue for TSO base when $n \geq 2$ or $m \geq 2$.

**Proof:** Assume that there is an algorithm $A$ that solves $P_1C_n$-queue for TSO base. Then, computations 21 and 22 exist. We show that Computation 23 satisfies TSO base.

To see that Computation 23 satisfies TSO base, we imagine a situation where $p$ completes its computation and its write-buffer is emptied first, then both $c$ and $d$ consume item $i$ before the contents of their write-buffers are committed to main memory. Specifically, $(O, \xrightarrow{L})$ is defined as follows. First, we build a sequence of operations $S$. Initially, $S = p$’s computation $\langle o_1^p, o_2^p, \ldots, o_k^p \rangle$. Set sequence $S_c$ to be $c$’s computation $\langle o_1^c, o_2^c, \ldots, o_l^c \rangle$. Also, set sequence $S_d$ to be $d$’s computation $\langle o_1^d, o_2^d, \ldots, o_j^d \rangle$. Examine each $o_i^c$ (respectively, $o_i^d$) in
$S_c$ (respectively, $S_d$) in order from $i = 1$ to $l$ (respectively, $j$). Let $o$ denote $o^c_i$ (respectively, $o^d_j$) and $S_o$ denote $S_c$ (respectively, $S_d$). If $o$ is a foreign read, append $o$ to $S$ and remove it from $S_o$. When there are no foreign reads left in $S_o$, append $S_o$ to $S$.

Define $(O, \xrightarrow{L})$ to be $S$. $(O, \xrightarrow{L})$ consists of three segments. The first consists entirely of $p$’s computation, the second consists entirely of foreign reads by $c$ and $d$, and the third consists entirely of $c$’s and $d$’s computations minus the operations in the second segment.

To show that $(O, \xrightarrow{L})$ is a linearization, note that the reads in the second segment necessarily return values written by $p$. So, the first and the second segments are valid. The construction guarantees that each read left in the third segment is necessarily domestic. That is, it returns the value written by a write in the same segment. Therefore, $(O, \xrightarrow{L})$ is a linearization.

Furthermore, $(O, \xrightarrow{LSO}) \subseteq (O, \xrightarrow{L})$. First of all, note that $(O|p, \xrightarrow{L}) = (O|p, \xrightarrow{prog})$. Consequently, $(O|p, \xrightarrow{LSO}) \subseteq (O|p, \xrightarrow{L})$. Second, program order is maintained in the second segment by construction and also in the third segment. Finally, each read moved to the first segment is foreign and, therefore, is not preceded in $c$’s (respectively, $d$’s) computation by any writes to the same variable. This implies that the moved reads do not violate $(O, \xrightarrow{LSO})$. Therefore, $(O, \xrightarrow{LSO}) \subseteq (O, \xrightarrow{L})$. Thus, Computation 23 satisfies $TSO_{base}$.

Therefore, our assumption must have been in error and $A$ does not exist.

**Corollary 7.2.10** There does not exist an algorithm that solves $P_m C_n$-queue or $P_m C_n$-queue for $TSO_{base}$ when $m + n \geq 3$.

Since $PSO_{base}$ is strictly weaker than $TSO_{base}$, the following follows immediately.

**Theorem 7.2.11** There does not exist an algorithm that solves $P_m C_n$-queue or $P_m C_n$-queue for $PSO_{base}$ when $m + n \geq 3$.

### 7.2.7 Java Consistency

In $Java_{base}$, it is impossible to achieve any form of coordination due to covert or fixate invisibilities. The only way by which a producer “produces” an item is by writing the item
into some shared variable. If the producer is covert, a consumer can never be made aware of the act of production. Similarly, the only way by which a consumer “consumes” an item is by reading a variable. If the consumer is fixate, it is possible for it to keep on reading a stale variable value forever.

**Observation 7.2.12** There does not exist an algorithm that solves $P_mC_n$-set for Java$_{base}$, for any $m,n \geq 1$.

Note that Observation 7.2.12 is a stronger claim than any of the previous theorems because it dismisses the possibility of solving $P_1C_1$-queue as well as $P_mC_n$-set. The rest of the memory consistency models, including Java$_1$, allow solutions for $P_1C_1$-queue and $P_mC_n$-set as we will be shown later in this chapter.

**Theorem 7.2.13** There does not exist an algorithm that solves $P_1C_n$-queue for Java$_1$ when $n \geq 2$ or $m \geq 2$.

**Proof:** Since $(O,\overset{jpo}{\rightarrow}) \subseteq (O,\overset{tso}{\rightarrow})$, the theorem follows from Theorem 7.2.9.

**Corollary 7.2.14** There does not exist an algorithm that solves $P_mC_n$-queue or $P_mC_n$-queue for Java$_1$ when $m+n \geq 3$.

### 7.2.8 PC-G

Since CSP is solvable for PC-G as shown in Chapter 6, any version of PCP is solvable for PC-G. However, PC-G with only single writers does not suffice to solve $P_1C_n$-queue, $P_mC_1$-queue, or $P_mC_n$-queue problems. This observation is immediate from Lemma 6.4.3 and Theorem 7.2.5.

**Theorem 7.2.15** There does not exist an algorithm that uses only single-writers and solves $P_mC_n$-queue for PC-G, when $m+n \geq 3$. 

7.3 Possibilities

We show in this section that $P_1C_1$-queue is solvable in many of the systems where the general case, $P_mC_n$-queue, is not. We also show that $P_mC_n$-set is also solvable in many cases.

7.3.1 Single-Writer $P_1C_1$-queue Algorithm

Figure 7.1 shows a single-writer $P_1C_1$-queue algorithm, $A_{PC1}$. The producer $p$ produces an item by writing it in $\mathbb{P}$, and the consumer $c$ consumes an item in $\mathbb{C}$. An item $i\;\mathbb{I}$ is composed of $k$ bits, and is encoded such that the first $k - 1$ bits contain the actual data. The $k$th bit is used for implicit synchronization. Items are initialized to $\perp$ concatenated with bit 0 (denoted $\perp || 0$). The function $b(\;\mathbb{I}t)$ returns the $k$th bit of $\;\mathbb{I}t$. Initially, $b(\mathbb{C}[i])$, $b(\mathbb{P}[j])$, $\mathbb{b}_{p}$, and $\mathbb{b}_{c}$ are all set to 0 forcing the condition at line (1) to evaluate to false, and allowing $p$ to progress into $\langle$producing$\rangle$. When $p$ writes its produced item to $\mathbb{P}[j]$, it toggles the synchronization bit, allowing $c$ to progress into $\langle$consuming$\rangle$. In the same manner, $c$ toggles the synchronization bit, allowing $p$ to progress into $\langle$producing$\rangle$. We denote the complement of bit $b$ as $\tilde{b}$.

Observe that for $p$ and $c$ to be self-consistent the data-dependences among accesses of in and out must be respected. So, it follows that both $p$ and $c$ access the arrays $\mathbb{P}$ and $\mathbb{C}$ circularly in order.

**Observation 7.3.1** In algorithm $A_{PC1}^p$, processes $p$ and $c$ access cell $j$ of any array only in iterations\(^1 j + nk$, $k \geq 0$.

The above observation implies that if $A_{PC1}^p$ solves $P_1C_1$-queue for a certain system with the arrays $\mathbb{P}$ and $\mathbb{C}$ each of size one, then it solves $P_1C_1$-queue for the same system with the arrays $\mathbb{P}$ and $\mathbb{C}$ each of size any positive integer.

We prove that $A_{PC1}^p$ is correct for $WO_{base}$. To simplify notation, we will pay no attention to the $k - 1$ data bits written or read in $A_{PC1}^p$. What really matters is the value of the $k$th

---

\(^1\)Iteration here refers to the non-terminating repeat loop of the process structure.
7.3. Possibilities

shared var:
P: array[0..n-1] of item (initialized to (⊥ || 0))
C: array[0..n-1] of item (initialized to (⊥ || 0))
define function b(it : item): returns the synchronization bit in it
define function v(it : item): returns the data bits in it

producer:

var

in : 0..n-1 (initially 0)
it_p : item
bit_p : bit (initially 0)

... produce it_p

consumer:

var

out : 0..n-1 (initially 0)
it_c : item
bit_c : bit (initially 0)

... consume it_c

< entry >
1) while (b(C[in]) ≠ bit_p) do nothing < consuming >
< producing >
bit_p ← bit_c
2) P[in] ← v(it_p) || bit_p < exit >
    in ← in + 1 mod n
    out ← out + 1 mod n
< exit >

Figure 7.1: A PC_1, a single-writer P_1C_1-queue algorithm

synchronizing bit. We also assume that P and C are of size one each. In the following, o^j_i denotes operation o pertaining to line (j) in the ith iteration. For instance, w^j_i(P)∗ is p’s write of line (2) in the iteration i.² Similarly, w^j_i(C)∗ is that of line (4). Let r^i_j(C)∗ denote the last read in line (1) which caused p to break the while loop; call such a read a successful read. Similarly, let r^i_j(P)∗ be c’s successful read in line (3).

Algorithm A^PC_1 is obviously correct for SC. The initial state of C and P allows p to progress to < producing > and prevents c from progressing to < consuming >. The consumer c can proceed to < consuming > only after p toggles the synchronization bit of P at line (2), thus, only after p completes production at the same line. Similarly, p cannot proceed to produce its second item until c progresses in < exit > toggling the synchroniza-

²The “∗” is used when the value read or written is of no importance to the discussion.
tion bit of $C$ at line (4). Let $O$ be all the operations resulting from $A^\text{PC}_1$ in a system that delivers SC. Then, $(O, \overset{L}{\rightarrow})$ consists of a production cycle corresponding to lines (1) and (2) followed by a consumption cycle corresponding to lines (3) and (4), and so on. So, $p$ cannot produce the $i\text{th}$ item until $c$ consumes the $i-1\text{st}$. Safety and Order obviously follow. Progress is also satisfied because it is either the case that $P \neq C$ or $P = C$, allowing one of $p$ or $c$ to progress.

**Theorem 7.3.2** $A^\text{PC}_i$ solves $P_1/C_1$-queue for $WO_{\text{base}}$.

**Proof:** Let $C_A$ be a finite prefix of an infinite computation of $A^\text{PC}_1$ in a $WO_{\text{base}}$ system. Let $O$ be all the operations resulting from $C_A$. By Definition 2.3.1, there is a linearization for each $q \in \{p, c\}$ $(O|q \cup O|w, \overset{L_q}{\rightarrow})$ such that $(O|q \cup O|w, \overset{\text{wpo}}{\rightarrow}) \subseteq (O|q \cup O|w, \overset{L_q}{\rightarrow})$.

Observe that $(O|w, \overset{\text{wpo}}{\rightarrow}) = (O|w, \overset{\text{pro}}{\rightarrow})$ because $p$ and $c$ each write to a single variable. Therefore, $(O|w, \overset{L_p}{\rightarrow}) = (O|w, \overset{\text{pro}}{\rightarrow})$ and $(O|w, \overset{L_c}{\rightarrow}) = (O|w, \overset{\text{pro}}{\rightarrow})$.

By $p$’s self consistency, the successful read at line (1) must precede the production at line (2). That is, $r^i_1(C) \overset{L_p}{\rightarrow} w^i_2(P) \overset{\star}{\rightarrow}$. Similarly, $r^i_3(P) \overset{L_c}{\rightarrow} w^i_4(C) \overset{\star}{\rightarrow}$. Since $(O|c \cup O|w, \overset{L_c}{\rightarrow})$ is a linearization and since only $p$ writes to $P$, preceding each $r^i_3(P) \overset{\star}{\rightarrow}$ there must be $k_p \geq 1$ writes by $p$ to $P$, $w^i_2(P)b$. Similarly, preceding each $r^i_1(C)b$ (except when $i = 0$) there must be $k_c \geq 0$ writes by $c$ to $C$, $w^i_4(C)b$. Since there is one successful read in each iteration of $p$ (respectively, $c$) at line (1) (respectively, (3)), a double induction implies that $k_c = k_p = 1$, except the very first iteration (iteration 0) of $p$, when the successful read returns an initial value. By the pigeon-hole principle, the write preceding the successful read at line (1) in iteration $i > 0$ must be caused by the write of line (4) in iteration $i$ as well. Moreover, the successful read at line (1) in iteration $i \geq 0$ must be caused by the write of line (4) in the same iteration. Therefore, $(O|w, \overset{L_p}{\rightarrow}) = \langle w^0_1(P)1, w^0_2(C)1, w^1_1(P)0, w^1_4(C)0, \ldots, w^i_1(P)b, w^i_2(C)b, w^{i+1}_1(P)b, w^{i+1}_4(C)b, \ldots \rangle$. This show that production is safe. Note that consumption effectively takes place when the successful read of line (3) takes place. Since the write that causes this read to succeed in any iteration $i$ of $c$ belongs to iteration $i$ of $p$, Safety and Order follow. Progress follows from the above argument in addition to the
observation that in any iteration \( i > 0, w_i^1(P) b \) and \( w_i^4(C) b' \) write opposite values \((b = \bar{b}')\). \(^3\)

**Corollary 7.3.3** \( A_1^{PC} \) solves \( P_1 C_1\)-queue for each of: Coherence, P-RAM, CC, TSO\(_{base}\), PSO\(_{base}\), and Java\(_1\).

**Proof:** Each one of these models is strictly stronger than WO\(_{base}\). \( \blacksquare \)

### SPIN Experiments

We have confirmed that \( A_1^{PC} \) solves \( P_1 C_1\)-queue for Coherence and P-RAM.

---

### 7.3.2 Simpler Single-Writer \( P_1 C_1\)-queue Algorithm

Figure 7.2 shows a simpler single-writer solution to \( P_1 C_1\)-queue. The algorithm uses the same ideas as that of \( A_1^{PC} \). Arrays \( P \) and \( C \) are used for synchronization purposes only. The actual data is produced in array \( Q \). If \( P[i] \neq C[i] \), then \( Q[i] \) contains a produced, unconsumed item. When \( P[i] = C[i] \), \( Q[i] \) is ready to hold a newly produced item.

Although \( A_2^{PC} \) is similar to \( A_1^{PC} \), \( A_2^{PC} \) does not solve the problem for many weak models such as WO\(_{base}\), PSO\(_{base}\), and P-RAM. However, \( A_2^{PC} \) solves \( P_1 C_1\)-queue for TSO\(_{base}\). In the following discussion, \( P, C, \) and \( Q \) are assumed to be of size one each. Also, Observation 7.3.1 applies to \( A_2^{PC} \).

**Theorem 7.3.4** \( A_2^{PC} \) solves \( P_1 C_1\)-queue for SC.

**Proof:** Let \( C_A \) be a finite prefix of an infinite computation of \( A_2^{PC} \) in a SC system. Let \( O \) be all the operations resulting from \( C_A \). By Definition 2.2.1, there is a linearization \((O, \overset{L}{\rightarrow})\) such that \((O, \overset{prog}{\rightarrow}) \subseteq (O, \overset{L}{\rightarrow})\). We show that any such linearization \((O, \overset{L}{\rightarrow})\) satisfies Safety, Progress, and Order.

The initial state of \( C \) and \( P \) allows \( p \) to progress to \(< producing >\) and prevents \( c \) from progressing to \(< consuming >\). The consumer \( c \) can proceed to \(< consuming >\) only.

\(^3\)We could have shown that the outcome of \( A_1^{PC} \) in WO\(_{base}\) is the same as that of SC. This can be achieved by constructing a SC linearization \((O, \overset{L}{\rightarrow})\) from the linearizations \((O[p \cup O[w, \overset{L}{\rightarrow}])\) and \((O[c \cup O[w, \overset{L}{\rightarrow}])\). The proof we chose here is shorter.
shared var:
Q: array[0..n-1] of item (initialized to 0)
P: array[0..n-1] of bit (initialized to 0)
C: array[0..n-1] of bit (initialized to 0)

producer:
var
in : 0..n-1 (initially 0)
it_p : item

... produce it_p

< entry >
1) while (Q[in] ≠ P[in]) do nothing

< producing >
2) Q[in] ← it_p

< exit >
3) P[in] ← P[in]
in ← in + 1 mod n

consumer:
var
out : 0..n-1 (initially 0)
it_c : item

4) < entry >
5) it_c ← Q[out]

< consuming >
6) C[out] ← C[out]
out ← out + 1 mod n

... consume it_c

Figure 7.2: A PC^2, a simpler single-writer P1C1-queue algorithm

after p toggles the value in P at line (3), thus, only after p completes production at line (2). Similarly, p cannot proceed to produce its second item until c progresses in < exit > toggling the value of C at line (6). So, p cannot produce the ith item until c consumes the i - 1st. Thus, (O, L) starts with a production iteration and consists of alternating production and consumption iterations. Safety and Order obviously follow. Progress is also satisfied because it is either the case that P ≠ C or P = C, allowing one of p or c to progress.

To show A PC^2 solves P1C1-queue for TSO base, the next theorem shows that its outcome for TSO base is the same as that for SC.

**Theorem 7.3.5** A PC^2 solves P1C1-queue for TSO base.

**Proof:** Let C_A be a finite prefix of an infinite computation of A PC^2 in a TSO base system. Let O be all the operations resulting from C_A. By Definition 4.2.3, there is a linearization
(O, \xrightarrow{L}) such that (O, \xrightarrow{ts}) \subseteq (O, \xrightarrow{L}). For each such (O, \xrightarrow{L}), we show how to construct (O, \xrightarrow{\text{L}}) such that (O, \xrightarrow{\text{L}}) is a SC linearization.

Since the value of P (respectively, C) is always known to p (respectively, c), we need only consider the successful read of C at line (1) (respectively, P at line (4)). Note that any read pertaining of C in line (1) and any read of P pertaining to line (4) is necessarily foreign because both P and C are single-writer variables. Also, a read of Q pertaining to line (5) is foreign.

In any fixed iteration i of A_{2}^{PC}, (O, \xrightarrow{ts}) = (O, \xrightarrow{\text{prog}}). Precisely, \text{r}_{i}^{1}(C) * \xrightarrow{L} \text{w}_{2}^{j}(Q) * \text{by the read-before-write constraint of (O, \xrightarrow{ts}). Also, \text{w}_{2}^{j}(Q) * \xrightarrow{L} \text{w}_{3}^{j}(P) * \text{by the two-writes constraint of (O, \xrightarrow{ts}). Similarly for c, \text{r}_{4}^{j}(P) * \xrightarrow{L} \text{r}_{5}^{j}(Q) * \text{by the two-reads constraint and \text{r}_{5}^{j}(Q) * \xrightarrow{L} \text{w}_{6}^{j}(C) * \text{by the read-before-write constraint.}

Now we argue that \text{w}_{3}^{j}(P) * \xrightarrow{L} \text{r}_{i}^{j+1}(C) * \text{The case where \text{w}_{6}^{j}(C) * \xrightarrow{L} \text{r}_{4}^{j+1}(P) * \text{is similar.}

Intuitively, \text{w}_{3}^{j}(P) * \text{must complete (written to main memory) before the read of C succeeds. This is because the read of C succeeds in iteration } i \geq 1 \text{ if and only if c performs the write at line (6) in iteration } i - 1. \text{So, c must have performed a successful read of P at line (4) in iteration } i - 1. \text{Thus, p must have completed the write to P at line (3) before its successful read of C at line (1). Therefore, } \text{w}_{3}^{j}(P) * \xrightarrow{L} \text{r}_{i}^{j+1}(C) * \text{for } i \geq 0.

The unsuccessful reads of C at line (1) may violate (O, \xrightarrow{\text{prog}}) in (O, \xrightarrow{L}) because the first is a write and the second is a read (intuitively, they might take place while the write to P at line (3) is still buffered). These unsuccessful reads in iteration i are moved in (O, \xrightarrow{L}) to form (O, \xrightarrow{\text{L}}) such that they follow \text{w}_{3}^{j}(P) and precede \text{w}_{6}^{j}(C). The consumer’s unsuccessful reads are moved in a similar way. Finally, (O, \xrightarrow{\text{L}}) is a linearization that maintains program order. By Theorem 7.3.4, A_{2}^{PC} solves P_1C_1-queue for TSO_{base}.

A_{2}^{PC} fails to solve P_1C_1-queue for many other memory consistency models. Computation 24 shows a scenario where the producer p produces two different items v_1 and v_2; however, c consumes only v_2 losing v_1. The operations are subscripted by the line numbers of Figure 7.2.
Computation 24 \[
\begin{align*}
p &: r_1(C)0, r_1(P)0, w_2(Q)v_1, w_3(P)1, r_1(C)1, r_1(P)1, w_2(Q)v_2 \\
c &: r_4(C)0, r_4(P)1, r_5(Q)v_2, w_6(C)1
\end{align*}
\]

Computation 24 satisfies CC. The following are linearizations for \( p \) and \( q \) that extend \((O, \xrightarrow{causal})\):

\[
(O|p \cup O|w, \xrightarrow{L_p}) = \langle r_1(C)0, r_1(P)0, w_2(Q)v_1, w_3(P)1, w_6(C)1, r_1(C)1, r_1(P)1, w_2(Q)v_2 \rangle.
\]

\[
(O|c \cup O|w, \xrightarrow{L_c}) = \langle r_4(C)0, w_2(Q)v_1, w_3(P)1, r_4(P)1, w_2(Q)v_2, r_5(Q)v_2, w_6(C)1 \rangle.
\]

\( A^PC_2 \) fails for P-RAM and \( WO_{\text{base}} \) because they are strictly weaker than CC. The algorithm also fails for \( PSO_{\text{base}} \). Consider Computation 25 where \( p \) produces \( v \) and \( c \) consumes \( v_u \), an item that has never been produced.

Computation 25 \[
\begin{align*}
p &: r_1(C)0, r_1(P)0, w_2(Q)v, w_3(P)1 \\
c &: r_4(C)0, r_4(P)1, r_5(Q)v_u
\end{align*}
\]

Because the buffers in \( PSO_{\text{base}} \) are partially FIFO, it is possible for \( w_3(P)1 \) to be committed to main memory before \( w_2(Q)v \). That is, \( p \) signals the completion of the production of \( v \) prematurely. Precisely, the following is a \( PSO_{\text{base}} \) linearization:

\[
(O, \xrightarrow{L}) = \langle r_1(C)0, r_1(P)0, w_3(P)1, r_4(C)0, r_4(P)1, r_5(Q)v_u, w_2(Q)v \rangle.
\]

Since \( A^PC_2 \) fails for \( PSO_{\text{base}} \), it also fails for Coherence since Coherence is strictly weaker than \( PSO_{\text{base}} \).

The essential difference between \( A^PC_1 \) and \( A^PC_2 \) is that the first combines actual production (respectively, consumption) with the signaling of production (respectively, consumption) completion in one operation. So, for consistency models that relax the write-before-write order such as \( PSO_{\text{base}} \), it is possible for a process to signal the completion of consumption or production before it is actually completed. Moreover, for systems where a mutual agreement on the order on writes might not exist such as P-RAM, production in iteration \( i \) might be regarded by the consumer as a production for iteration \( j \neq i \).
7.3.3 Multi-Writer \( P_1 C_1 \)-queue Algorithm

Whenever \( P_1 C_1 \)-queue can be solved using only single-writer variables, it can be solved using multi-writer variables. The multi-writer algorithm in Figure 7.3, \( A^\text{PC}_3 \), does not use the trick of synchronization bits used in \( A^\text{PC}_1 \) and is easier to prove correct for Coherence.

This algorithm uses a single array \( Q \). When \( Q[i] \neq \bot \), \( Q[i] \) bears a produced, unconsumed item, and \( Q[i] = \bot \) means that it is safe for the producer to write its production into \( Q[i] \). When \( Q[i] \neq \bot \), we say that \( Q[i] = T \). The following discussion assumes that \( Q \) is of size one and Observation 7.3.1 applies to \( A^\text{PC}_3 \).

The intuition behind \( A^\text{PC}_3 \) is that it uses a single multi-writer object \( Q \), and Coherence requires a total order on all the operations on \( Q \) which is consistent with program order. Therefore, the outcome of \( A^\text{PC}_3 \) for Coherence is the same as its outcome for SC.

**Theorem 7.3.6** \( A^\text{PC}_3 \) solves \( P_1 C_1 \)-queue for SC.

**Proof:** Let \( C_A \) be a finite prefix of an infinite computation of \( A^\text{PC}_3 \) in a SC system. Let \( O \) be all the operations resulting from \( C_A \). By Definition 2.2.1, there is a linearization \((O, L)\)
such that \((O, \stackrel{\text{prog}}{\longrightarrow}) \subseteq (O, \stackrel{L}{\longrightarrow})\). We show that \((O, \stackrel{L}{\longrightarrow})\) satisfies Safety, Progress, and Order.

In \((O, \stackrel{L}{\longrightarrow})\), only \(p\) can progress to \(<\text{producing}\>\) in the first iteration (by the initial state of \(Q\)). So, \(c\) can progress to \(<\text{consuming}\>\) only after \(p\) writes to \(Q\) at line (2). In subsequent iterations, \(p\) can progress to \(<\text{producing}\>\) only after \(c\) writes \(\bot\) to \(Q\) at line (5). Therefore, \((O, \stackrel{L}{\longrightarrow})\) consists of a production iteration followed by a consumption iteration, and so on. Safety and Order follow obviously. Also, Progress is satisfied because at any time \(Q = \bot\) or \(Q \neq \bot\), allowing one process to proceed.  

\[\square\]

**Theorem 7.3.7** \(A_3^\text{PC}\) solves \(P_1C_1\)-queue for Coherence.

**Proof:** \(A_3^\text{PC}\) uses only one variable; thus \(O|Q = O\), and \((O|Q, \stackrel{L_Q}{\longrightarrow})\) is defined to be \((O, \stackrel{L}{\longrightarrow})\) given by SC. The correctness follows from Theorem 7.3.6.  

Since \(A_3^\text{PC}\) is correct for Coherence, it is also correct for any system stronger than Coherence.

**Corollary 7.3.8** \(A_3^\text{PC}\) solves \(P_1C_1\)-queue for each of: TSO_{base}, PSO_{base}, or Java_{1}.

However, \(A_3^\text{PC}\) can deadlock for P-RAM as confirmed by Computation 26.

**Computation 26**

\[
\left\{ \begin{array}{l}
p : r_1(Q) \bot, w_2(Q) \top, [r_1(Q) \top], [r_1(Q) \bot], \ldots \\
c : r_3(Q) \top, r_4(Q) \top, w_5(Q) \bot, [r_3(Q) \bot], [r_3(Q) \bot], [r_3(Q) \bot], \ldots
\end{array} \right.
\]

In Computation 26, \(p\) produces an item \(\top\) and loops forever at line (1) of \(A_3^\text{PC}\) as indicated by the repetition of \([r_1(Q) \top]\). Similarly, \(c\) consumes \(\top\) and gets stuck in the loop of line (3) represented by the repetition of \([r_3(Q) \bot]\).

Computation 26 satisfies P-RAM as indicated by the following linearizations:

\[
(O|p \cup O|w, \stackrel{L_p}{\longrightarrow}) = \langle w_5(Q) \bot, r_1(Q) \bot, w_2(Q) \top, [r_1(Q) \top], [r_1(Q) \top], [r_1(Q) \top] \rangle
\]

\[
(O|c \cup O|w, \stackrel{L_c}{\longrightarrow}) = \langle w_2(Q) \top, r_3(Q) \top, r_4(Q) \top, w_5(Q) \bot, [r_3(Q) \bot], [r_3(Q) \bot], [r_3(Q) \bot] \rangle
\]

Note that these linearizations can be always extended by appending \([r_1(Q) \top]\) and \([r_3(Q) \bot]\) to \((O|p \cup O|w, \stackrel{L_p}{\longrightarrow})\) and \((O|c \cup O|w, \stackrel{L_c}{\longrightarrow})\) respectively, satisfying Definition 2.5.1 of non-termination. Finally, since \(A_3^\text{PC}\) can deadlock for P-RAM, it can also deadlock for WO_{base}.  

\[\square\]
With similar analysis, we can show that it deadlocks for CC. In fact, the linearizations given for P-RAM are not valid for CC. The reason is that \( r_1(q) \perp 1 \) is not causally related to \( w_5(q) \perp 2 \) because the former is returning the initial \( \perp 1 \) while the latter is writing a new \( \perp 2 \). We could achieve the desired conclusion by running \( p \) for an additional iteration.

**SPIN Experiments**

A \( \text{PC}^C \) has been verified to solve \( P_1C_1 \)-queue for Coherence. Moreover, Computation 26 is based on a SPIN simulation run.

### 7.3.4 \( P_mC_n \)-set Algorithm

Although the \( P_mC_n \)-queue cannot be solved in many weak systems, we show in this section that the \( P_mC_n \)-set can be solved in many of those systems.

**Theorem 7.3.9** There exists an algorithm that solves \( P_mC_n \)-set for \( \text{WO}_{\text{base}} \).

**Proof:** Since we can solve \( P_1C_1 \)-queue in such a system, \( P_1C_n \)-set can be solved by associating a separate queue with each consumer. The producer inserts its items into these queues using any discipline, say round-robin. Safety and Progress follow from the correctness of the \( P_1C_1 \)-queue solution. Similarly, \( P_mC_1 \)-set can be solved by associating queues with producers. To solve the general \( P_mC_n \)-set we combine these two solutions. If \( m \geq n \) let consumer \( c_i \) consume from producers \( p_{ik} \) for \( ik \leq n \), and similarly for \( n \geq m \).

**Corollary 7.3.10** There exists an algorithm that solves \( P_mC_n \)-set for each of the models of Theorem 7.3.3.

**Proof:** Each of these systems is strictly stronger than \( \text{WO}_{\text{base}} \).

### 7.4 Summary

We have revisited PCP in the context of weak memory consistency models. The solution requirements for both versions of PCP are weaker than those of CSP. This allows some memory models to support a solution, using only base read and write operations.
Figure 7.4: Summary of coordination possibilities (√) and impossibilities (×)

<table>
<thead>
<tr>
<th>Model</th>
<th>CSP</th>
<th>$P_1C_1$-queue</th>
<th>$P_1C_n$-queue</th>
<th>$P_mC_1$-queue</th>
<th>$P_mC_n$-queue</th>
<th>$P_mC_n$-set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coherence</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>P-RAM</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>CC</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>TSO$<em>{base}$ or PSO$</em>{base}$</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>Java$_1$</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>Java$_{base}$</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>WO$_{base}$</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>PC-G without multi-writers</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>PC-G [9]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SC</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Figure 7.4 summarizes the impossibilities and possibilities. The figure also restates the CSP results from the previous chapter for comparison purposes. Note that Java$_{base}$ is the only model where no form of coordination is possible.
CHAPTER 8

Conclusion

The more complex an entity is, the less useful it becomes. Weak memory consistency models are very complex. Even though they might enhance performance and scalability, they shift the burden of creating useful systems to the programmer. If the programmers, especially system programmers, are unaware of the exact behavior of the underlying memory system, incorrect or inefficient (or both) systems must be expected.

8.1 Summary

This dissertation develops and exploits a formalism for rigorously describing memory consistency models. The framework described in Chapter 2 lays down the foundations for high-level description of operations on memory, in terms of read and write program instructions or higher-level operations. This high-level description is non-operational because it hides the operational details of the underlying multiprocess system. The need of non-operational specification stems from the need to specify a contract between the underlying system and the set of behaviors it might exploit. Programmers are in need of such a contract in order to be able to properly program these systems. Unfortunately, actual systems are normally described in terms of the low level operations they support, rather than high-level program operations. This further complicates the job of associating a non-operational memory consistency specification with the actual operational behavior of a certain memory system. Chapter 3 extends the formalism of Chapter 2 and provides a mechanism to specify the operational behavior of the underlying multiprocess system. This operational specifi-
cation does not deviate significantly from the original description, but is rigorous enough to establish a relationship between the non-operational specification and the operational implementation.

The non-operational description allows us to compare these models with many other models that have been described in the literature, such as P-RAM, CC, PC-G, and WO. Figure 5.4 summarizes the relationships between all these models.

Chapters 4 and 5 apply these foundations to two state-of-the-art multiprocess systems. Chapter 4 provides non-operational definitions for the SPARC version 8 multiprocessor architecture. This is an example of a physical system that deviates substantially from traditional Sequential Consistency. Chapter 5 formalizes the Java memory consistency model. The Java language includes multi-threading as part of the language specification. Java provides a variety of memory consistency models ranging from Sequential Consistency to a model that is extremely weak. The SPARC and Java models incorporate expensive synchronization constructs that enforce additional constraints on the ordering and interleaving of memory accesses. The use of these synchronization constructs allows solving coordination problems between processes.

This formalism is exploited further in chapters 6 and 7 to answer the following natural question: What forms of fundamental process coordination problems can be solved without resorting to the use of expensive synchronization primitives? With the exception of PC-G, none of these models is capable of supporting a solution with only read/write variables to the critical section problem. The producer/consumer problem is a weaker form of coordination. So, it might be solvable where the critical section problem is not. Chapter 7 shows that certain versions of this problem are solvable in all these models (with the exception of Java) without the use of explicit synchronization. A summary of impossibilities and possibilities is given in Figure 7.4.
8.2 Conclusions

Weak memory consistency models could boost the performance and improve the scalability of a multiprocess system. However, they pay the price of programmability. This dissertation shows that weakening the constraints of the consistency model creates a very complex system. This complexity makes reasoning about correctness of programs very difficult. This in turn leads programmers to overuse expensive synchronization primitives to guarantee correctness. This overuse of synchronization might be very expensive, sometimes to the extent where the performance gain of weakening the memory consistency model is out-weighed by the overuse of synchronization. To see whether weak memory consistency models pose the same problem they are intending to solve, more extensive and comprehensive experimental work is required.

SPARC

The SPARC consistency models turned out to be elegant to describe. Given that Coherence is a basic “reasonable” requirement for a consistency model and that the SPARC TSO and PSO are strictly stronger than Coherence, it is concluded that the SPARC TSO and PSO are reasonable models. However with the exception of a simpler $P_1C_1$-queue algorithm for TSO, it is not clear from the instances studied in this dissertation how TSO and PSO have more coordination capabilities than Coherence.

Java

Surprisingly, Java Consistency was very complex to describe and for base operations, too weak to accept as a consistency model. While this very weak model has been, probably, designed with performance and scalability as a target, it turned out too weak to support any minimum form of coordination. The result is a programming language where programmers must use synchronization all the time. If some “cache consistency” mechanism is built into the Java Virtual Machine, the extreme weakness of Java consistency could be strength-
ended to something stronger than Coherence. It is recommended that Java implementations incorporate such a mechanism.

**PC-G**

PC-G is the only model that can support solutions to most forms of coordination without using explicit synchronization. It is not clear to us, though, how this model could be implemented.

**Process Coordination**

The process coordination patterns studies in this dissertation are fundamental; they can appear in a wide range of parallel and distributed applications. The classic critical section problem does require strong consistency conditions. On the contrary, certain versions of the producer/consumer, especially $P_m C_n$-set problem, are solvable in models as weak as Coherence. Even though the latter problem is frequently used in the literature to motivate the former, the producer/consumer problem is less demanding than the critical section problem. Certain versions of the problem do not require mutual exclusion.

The impossibility of the critical section coordination implies directly the impossibility of building read-modify-write objects\(^1\) from read and write variables. The argument is very simple; if building such an object is possible, then solving the critical section problem is also possible. In fact, a two way reduction between the critical section problem and building a read-modify-write object is straightforward.

All the process coordination impossibility proofs do not depend on the fairness of the solution. It is not clear to us yet how impossibilities could be proven based on fairness. For example, we have shown that $n$ single-writer and 1 multi-writer variables is a lower-bound for $n$-process critical section problem for PC-G. However, to tighten this lower bound, we feel that fairness must be exploited.

---

\(^1\)These include test-and-set, fetch-and-add, swap, swap-atomic, fetch-and-increment, and others.
8.3 Limitations

Hierarchy of Systems

We restricted discussion in this dissertation to two levels of consistency, which we called operational and non-operational. In theory, a non-operational description in one setting could be regarded as an operational description in another, and vice versa. For instance, building a PC-G system with read and write variables from an underlying message passing architecture requires the message passing architecture to be operational and the former to be non-operational. Nevertheless, building a Linearizable stack system from PC-G read and write variables requires the former to be the non-operational level and the latter to be operational. This hierarchy of systems is theoretically sound. However, the way we chose to define a machine in Chapter 3 disallows this approach. As a matter of fact, the approach chosen here is ad hoc; it served the purpose of the dissertation with intended simplicity at the expense of generality and soundness. Adjusting the framework to support this hierarchy of systems is the subject of future research.

Architecture Design

One major advantage of the non-operational description demonstrated in this dissertation is that it provides programmers with an intuitive, yet precise way to reason about the underlying memory consistency model. However, it is not clear to us how our operational description might be also useful to architects and hardware designers. The techniques demonstrated here help establish a relation between a given architecture and a memory consistency model. However, guiding the architecture design process to meet certain consistency criteria (although beyond the objective of this dissertation) might be beyond the capabilities of this framework.
Restricted Schedulers

To the best of our knowledge, schedulers are implicitly assumed to function in an arbitrary way in all the research in the area of memory consistency models (including this dissertation). If these schedulers are restricted by creating timing constraints, we expect some of the driven conclusions to change. It is not clear to us yet how to incorporate these constraints into the techniques illustrated here.

8.4 Future Research

Wait-Free Coordination

The process coordination problems studied in this dissertation either have “equal-authority” processes (critical sections) or “division of labor” (producers and consumers). In certain resource allocation problems, it might be essential to elect a designated leader process. The leader election problem is equivalent to building a wait-free test-and-set object. Building a test-and-set from read/write variables is impossible where critical sections cannot be built. Furthermore, deterministic wait-free test-and-set is impossible even for Linearizability. An expected wait-free test-and-set might be possible for PC-G. We are investigating a randomized algorithm developed for the purpose. The number of cases that arise from analyzing all possible executions of this algorithm in PC-G is extraordinary. Moreover, since processes are allowed in PC-G to “leap into the future”, proving the correctness of this algorithm, if possible at all, is a painful long process.

Illusion of Sequential Consistency

In the not too distant past, programmers had to master a different assembly language for each different architecture. While architectures are rarely the same, some of the complexity of building meaningful computer systems had to be hidden for most programmers. This was accomplished by the illusion of the random access machine.
History is repeating itself with parallel and distributed systems, but the illusion of Sequential Consistency has not emerged yet. Programmers and algorithm designers continue to develop algorithms for Sequential Consistency (if not for Linearizability). This is an indication that the power of these strong models cannot be dismissed. These models seem to be irreplaceable; they are “natural” for designing solutions to fundamental problems.

If time reveals that weak memory consistency models are an inevitable route to build higher-performance and larger-scale systems, building the illusion of Sequentially Consistent systems is the natural choice. Although there are a few attempts in this direction, as discussed in Chapter 1, we still lack the knowledge required to build efficient compilers to translate Sequentially Consistent code to a weaker consistency model.

**Experimental Analysis**

There is a lack of sufficient experimental analysis to justify weak memory consistency models. As discussed in Chapter 1, earlier results reported that these models outperform traditional systems. Some recent results [60] reported that the same performance gain provided by weak models can be achieved by retaining a strong memory model, while incorporating state-of-the-art processor elements. With many vendors adopting these weak models, this question should continue to gain importance.

Since the majority of these weak models require explicit synchronization in order to be programmed properly, it is very important to find ways to quantify and measure the loss in performance incurred by such synchronization. This leads to many other questions, such as how, when, and where do programmers tend to use synchronization instructions? Is there a form of synchronization better than another, regardless of the underlying memory model? Or, what is the best synchronization form, memory consistency model combination?
8.5 Final Comment

Although the area of memory consistency models has received substantial attention from researchers, many of the posed questions are still open. Controversy, regrets, and abandoning the area do not change facts; these models exist. We would better understand and use them properly.


[52] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multi-

[53] Doug Lea. *Concurrent Programming in Java Design Principles and Patterns*. Addison-
Wesley, 1997.

[54] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta,
John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor.

1997.


[58] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quar-

26, January 1993.

[60] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An evaluation

[61] Seugjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO
(relaxed memory order). In *Proc. 7th Int’l Symp. on Parallel Algorithms and Architectures*,
July 1995.


APPENDIX A

Partial and Total Orders

A.1 Basic Definitions

Let $S$ be a set, and $R \subseteq S \times S$. $R$ is anti-reflexive if $\forall x \in S$, $(x, x) \notin R$. $R$ is transitive if $\forall x, y, z \in S$, if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.

A (strict) partial order (simply, partial order) is an anti-reflexive, transitive relation. Denote a partial order by a pair $(S, R)$. The notation $s_1 Rs_2$ means $(s_1, s_2) \in R$. When the set $S$ is understood, $R$ denotes the partial order. If $S' \subseteq S$ then $(S', R)$ denotes the relation $(S, R \cap (S' \times S'))$.

A (strict) linear order (also called (strict) total order) is a partial order $(S, R)$ such that $\forall x, y \in S$, $x \neq y$, either $xRy$ or $yRx$. A (strict) linear or total order is simply referred to as linear or total order.

A.2 Redundant Transitivity

Given a partial order $(S, R)$ where $S$ is finite, define $(S, R_c)$ as follows. $(a, b) \in (S, R_c)$ if and only if $(a, b) \in (S, R)$ and $a$ covers $b$ ($\forall c \in S$ such that $aRc$ and $cRb$).

Proposition. For each pair $(a, b)$ in $(S, R)$, there is a nonempty chain $(a, c_1)$, $(c_1, c_2)$, $(c_2, c_3)$, $\cdots$, $(c_k, b) \in (S, R)$, such that for every pair $(x, y)$ in the chain, $x$ covers $y$.

Let $(S, T)$ be a total order.

Claim. $(S, R) \subseteq (S, T)$ if and only if $(S, R_c) \subseteq (S, T)$.
**Proof:** Since \((S, R_c) \subseteq (S, R)\) by definition, it follows that if \((S, R) \subseteq (S, T)\), then \((S, R_c) \subseteq (S, T)\).

For the other direction, let \((S, R_c) \subseteq (S, T)\). Let \((a, b) \in (S, R)\). Then, there exists a chain \((a, c_1), (c_1, c_2), (c_2, c_3), \cdots, (c_k, b) \in (S, R)\), such that every pair in the chain is in \((S, R_c)\). Since \((S, T)\) is transitive, \((a, b) \in (S, T)\). 

The consequence of the above claim is that even though some relations on operations such as \((O, \overset{tso}{\rightarrow}), (O, \overset{pso}{\rightarrow}),\) and \((O, \overset{jpo}{\rightarrow})\) are defined to be transitive (partial orders), transitivity does not add anything to the definitions of TSO (Definition 4.2.3), PSO (Definition 4.2.8), or Java (Definition 5.2.1). This is because, these models are defined in terms of a total order \((O, \overset{L}{\rightarrow})\) that extends the appropriate partial order.
In this appendix, we quote the ordering rules from the SPARC manual [67]. The ordering rules are given in the form of axiomatic specifications. These specifications include a FLUSH instruction which “synchronizes the instruction fetches of the processor issuing the FLUSH to the loads, stores, and atomic load-stores of that processor and forces the instruction fetches of all other processors to observe any store done to the FLUSH target prior to the FLUSH”. The FLUSH instruction as well as instruction fetches and loads are omitted from what follows since they are not of any interest to this dissertation. Omitted sentences or phrases are denoted by [...].

**B.1 Basic Definitions**

“Data loads and stores are denoted by $L$ and $S$, respectively. Atomic load-stores are denoted by $[L;S]$, where [ ] represents atomicity. The instruction STBAR is denoted by $\mathcal{S} [...]$. Superscripts on $L$, $S$, $\mathcal{S}$, [...] refer to processor numbers, while subscripts on $L$, $S$ , [...] refer to memory locations. $\mathcal{S}$ does not carry a subscript because conceptually it applies to all memory locations. A $#n$ after $S$ refers to the value written by $S$. [...].

[...] The value returned by an $L$ [...] or stored by an $S$ is denoted by $\text{Val}[]$. $\text{Val}$ is not defined for $[L;S]$ as a whole or for $\mathcal{S}$. [...].

$SOp$ is used as a shorthand for $S$ [...]. $Op$ is used as a shorthand for $L$, $S$, [...]. Finally, $(Op;)_\infty$ denotes the infinite sequence of $Op$.”

“The formalism uses two types of orders defined over the set of operations $\{L, S, [...], \mathcal{S}\}$:
A single partial order ≤ called the **memory order**. [...].

A per-processor order \(i\) that denotes the sequence in which processor \(i\) executes instructions. This is called the **program order**. [...].”

### B.2 Total Store Ordering

“The complete semantics of TSO are captured by six axioms: Order, Atomicity, Termination, Value, LoadOp, and StoreStore.”

- **Order:**
  \[(SOp^i_a \leq SOp^j_b) \lor (SOp^j_b \leq SOp^i_a)\]

- **Atomicity:**
  \[[L^i_a; S^i_a] \Rightarrow (L^i_a \leq S^i_a) \land (\forall SOp^j_b : SOp^j_b \leq L^i_a \lor S^i_a \leq SOp^j_b)\]

- **Termination:**
  \[S^i_a \land (L^i_a; \infty) \Rightarrow \exists \text{ an } L^i_a \text{ in } (L^i_a; \infty) \text{ such that } S^i_a \leq L^i_a\]

- **Value:**
  \[\text{Val}[L^i_a] = \text{Val}[S^i_a|S^i_a = \text{Max}_{\leq \{\{S^k_a|S^k_a \leq L^i_a\} \cup \{S^i_a|S^i_a:L^i_a\}\]}

- **LoadOp:**
  \[L^i_a; Op^i_b \Rightarrow L^i_a \leq Op^i_b\]

- **StoreStore:**
  \[SOp^i_a, SOp^i_b \Rightarrow SOp^i_a \leq SOp^i_b\]

### B.3 Partial Store Ordering

“The complete semantics of PSO are captured by seven axioms: Order, Atomicity, Termination, Value, LoadOp, StoreStore, and StoreStoreEq. The first five axioms are identical to those for TSO.”
B.3. Partial Store Ordering

- **StoreStore:**
  \[ SOP_a^i; SOP_b^i \Rightarrow SOP_a^i \preceq SOP_b^i \]

- **StoreStoreEq:**
  \[ SOP_a^i; SOP_a^i \Rightarrow SOP_a^i \preceq SOP_a^i \]
APPENDIX C

Java Ordering Constraints

In this appendix, we quote the ordering rules from the Java manuals [35, 55]. Let T be a thread, V be a variable, and L be a lock variable.

C.1 Rules for One Thread

1. “A use or assign by T of V is permitted only when dictated by execution by T of the Java program according to the standard Java execution model. For example, an occurrence of V as an operand of the + operator requires that a single use operation occur on V; an occurrence of V as the left-hand operand of the assignment operator = requires that a single assign operation occur. All use and assign actions by a given thread must occur in the order specified by the program being executed by the thread. If the following rules forbid T to perform a required use as its next action, it may be necessary for T to perform a load first in order to make progress.”

2. “A store operation by T on V must intervene between an assign by T of V and a subsequent load by T of V. (Less formally: a thread is not permitted to lose the most recent assign.)”

3. “An assign operation by T on V must intervene between a load or store by T of V and a subsequent store by T of V. (Less formally: a thread is not permitted to write data from its working memory back to main memory for no reason.)”

4. “After a thread is created, it must perform an assign or load operation on a variable
before performing a use or store operation on that variable. (Less formally: a new
thread starts with an empty working memory.)"

5. “After a variable is created, every thread must perform an assign or load operation
on that variable before performing a use or store operation on that variable. (Less
formally: a new variable is created only in main memory and is not initially in any
thread’s working memory.)”

### C.2 Rules for Thread-Main Memory Interaction

1. “For every load operation performed by any thread T on its working copy of a vari-
   able V, there must be a corresponding preceding read operation by the main memory
   on the master copy of V, and the load operation must put into the working copy the
data transmitted by the corresponding read operation.”

2. “For every store operation performed by any thread T on its working copy of a vari-
   able V, there must be a corresponding following write operation by the main memory
   on the master copy of V, and the write operation must put into the master copy the
data transmitted by the corresponding store operation.”

3. “Let action A be a load or store by thread T on variable V, and let action P be the
   corresponding read or write by the main memory on variable V. Similarly, let action
   B be some other load or store by thread T on that same variable V, and let action Q be
   the corresponding read or write by the main memory on variable V. If A precedes B,
   then P must precede Q. (Less formally: operations on the master copy of any given
   variable on behalf of a thread are performed by the main memory in exactly the order
   that the thread requested.)”
C.3 Rules for Locks

1. “With respect to a lock, the lock and unlock operations performed by all the threads are performed in some total sequential order. This total order must be consistent with the total order on the operations of each thread.”

2. “A lock operation by T on L may occur only if, for every thread S other than T, the number of preceding unlock operations by S on L equals the number of preceding lock operations by S on L. (Less formally: only one thread at a time is permitted to lay claim to a lock; moreover, a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of unlock operations have been performed.)”

3. “An unlock operation by thread T on lock L may occur only if the number of preceding unlock operations by T on L is strictly less than the number of preceding lock operations by T on L. (Less formally: a thread is not permitted to unlock a lock it does not own.)”

4. “Between an assign operation by T on V and a subsequent unlock operation by T on L, a store operation by T on V must intervene; moreover, the write operation corresponding to that store must precede the unlock operation, as seen by main memory. (Less formally: if a thread is to perform an unlock operation on any lock, it must first copy all assigned values in its working memory back out to main memory.)”

5. “Between a lock operation by T on L and a subsequent use or store operation by T on a variable V, an assign or load operation on V must intervene; moreover, if it is a load operation, then the read operation corresponding to that load must follow the lock operation, as seen by main memory. (Less formally: a lock operation behaves as if it flushes all variables from the thread’s working memory, after which it must either assign them itself or load copies anew from main memory.)”
C.4 Rules for Volatiles

1. “A use operation by T on V is permitted only if the previous operation by T on V was load, and a load operation by T on V is permitted only if the next operation by T on V is use. The use operation is said to be ”associated” with the read operation that corresponds to the load.”

2. “A store operation by T on V is permitted only if the previous operation by T on V was assign, and an assign operation by T on V is permitted only if the next operation by T on V is store. The assign operation is said to be ”associated” with the write operation that corresponds to the store.”

3. “Let action A be a use or assign by thread T on variable V, let action F be the load or store associated with A, and let action P be the read or write of V that corresponds to F. Similarly, let action B be a use or assign by thread T on variable W, let action G be the load or store associated with B, and let action Q be the read or write of V that corresponds to G. If A precedes B, then P must precede Q. (Less formally: operations on the master copies of volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.) ”

C.5 Other Rules

These rules can be inferred from several places in the Java manuals. We quote these from [32].

1. “The operations performed by any one thread are totally ordered. A use of V or a store to V in one of the program orders always uses the most recent value that was given to V by an assign or a load operation in that order.”

2. “The operations performed by the main memory for any one variable are totally ordered. A read in the order of one of the variables always yields the value that was written by the last write in that order. If there is no preceding write in the order, the
value yielded by \textit{read} is some initial value.”

3. “It is not permitted for an instruction to follow itself.”
APPENDIX D

CSP Algorithms

For each of the following CSP algorithms, processes have unique identifiers from the set \( \{0, \cdots, n-1\} \), where \( n \) is the total number of processes. The algorithms are given by specifying the \(<entry>\) and \(<exit>\) sections of process \( i, i \in \{0, \cdots, n-1\} \).

D.1 Peterson’s Algorithm

Two Processes

shared objects
flag[0 .. 1] in \{true, false\}, single-writer
turn in \{0, 1\}, multi-writer

<entry>
flag[j] \gets true
turn \gets j
while (flag[j] and turn = j) do nothing

<critical section>

<exit>
flag[j] \gets false

n Processes

shared objects
flag[0 .. n-1] in \{-1 .. n-2\}, single-writer
turn[0 .. n-2] in \{0 .. n-1\}, multi-writer

<entry>
for \( k = 0 \) to \( n-2 \) do
  flag[k] \gets k
  turn[k] \gets i
  while (\( \forall j \neq i \), flag[j] \( \geq k \) and turn[k] = i) do nothing

<critical section>

<exit>
flag[i] \gets -1
D.2 Dekker’s Two-process Algorithm

2 Processes

shared objects
flag[0 .. 1] in {true, false}, single-writer
turn in {0, 1}, multi-writer

<entry>
flag[0] ← true
while (flag[j]) do
  if (turn = j) then
    flag[j] ← false
    while (turn = j) do nothing
    flag[j] ← true
  end-if
end-while

<critical section>

<exit>
turn ← j
flag[0] ← false

D.3 Dijkstra’s Algorithm

n Processes

shared objects
flag[0 .. n-1] in {idle, requesting, in-cs}, single-writer
turn in {0, ..., n-1}, multi-writer

<entry>
repeat
  flag[0] ← requesting
  while (turn ≠ i) do
    if (flag[turn] = idle) then
      turn ← i
    end-if
  end-while
  flag[0] ← in-cs
until (∀j ≠ i, flag[j] ≠ in-cs)

<critical section>

<exit>
flag[0] ← idle
D.4 Knuth’s Algorithm

n Processes

shared objects
flag[0 .. n-1] in {idle, requesting, in-cs}, single-writer
turn in {0, ..., n-1}, multi-writer

<entry>
repeat
flag[i] ← requesting
j ← turn
while (j ≠ i) do
if (flag[j] ≠ idle) then
  j ← turn
else j ← (j - 1) mod n
end-while
flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)
turn ← i

<critical section>

<exit>
turn ← (i - 1) mod n
flag[i] ← idle

D.5 De Bruijn’s Algorithm

n Processes

shared objects
flag[0 .. n-1] in {idle, requesting, in-cs}, single-writer
turn in {0, ..., n-1}, multi-writer

<entry>
repeat
flag[i] ← requesting
j ← turn
while (j ≠ i) do
if (flag[j] ≠ idle) then
  j ← turn
else j ← (j - 1) mod n
end-while
flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)
<critical section>

<exit>
if (flag[turn] = idle and turn = i) then
  turn ← (turn - 1) mod n
end-if
flag[i] ← idle
D.6 Eisenberg and MacGuire’s Algorithm

\( n \) Processes

shared objects
flag[0 .. \( n-1 \)] in \{idle, requesting, in-cs\}, single-writer
turn in \{0, \ldots, n-1\}, multi-writer

<entry>
repeat
  \( \text{flag}[i] \leftarrow \text{requesting} \)
  \( j \leftarrow \text{turn} \)
  while \((j \neq i)\) do
    if \(\text{flag}[j] \neq \text{idle}\) then
      \(j \leftarrow \text{turn}\)
    else \(j \leftarrow (j+1) \mod n\)
  end-while
  \(\text{flag}[i] \leftarrow \text{in-cs}\)
until ((\(\forall j \neq i, \text{flag}[j] \neq \text{in-cs}\)) and (\(\text{turn} = i \) or \(\text{flag[\text{turn}] = idle}\)))
\(\text{turn} \leftarrow i\)

<critical section>

<exit>
\(j \leftarrow (\text{turn} + 1) \mod n\)
while \((j \neq \text{turn} \) and \(\text{flag}[j] \neq \text{idle}\)) do
  \(j \leftarrow (j+1) \mod n\)
end-while
\(\text{turn} \leftarrow j\)
\(\text{flag}[i] \leftarrow \text{idle}\)
This appendix gives an example that illustrates how SPIN can be used to verify algorithms for weak memory consistency models.

The formalism described in chapters 2 and 3 allows us to establish the equivalence between a machine and a memory consistency model. So, these machines can be described as SPIN processes; furthermore, verification of a certain algorithm in the context of a particular memory consistency model constitutes implementing the algorithm instructions by a set of machine events. Consider the machine $M_C$ introduced in Chapter 3. This machine exactly implements Coherence, as we proved in that chapter. So verifying an algorithm, say Peterson’s Algorithm, comprises the interaction of the processes of this algorithm with a process that simulates $M_C$.

**PROMELA Example**

The PROMELA program that simulates $M_C$ and Peterson’s Algorithm are given next.

```c
/* Mc process */

/* Message Type */
mtype = {R, W}

/* Channels */
#define MESSAGE {mtype, byte, byte, byte}
#define Handshake(x) chan x = [0] of MESSAGE
#define EChannel(x) chan x = [BUFSIZE] of MESSAGE

/* Global Channels*/
chan SwitchToMem[MEM] = [BUFSIZE] of MESSAGE;
```
typedef ChannelsPerProcess
{
    chan Output[MEM] = [BUFSIZE] of MESSAGE;
    chan Input[MEM] = [BUFSIZE] of MESSAGE;
}

ChannelsPerProcess CohChannels[PROC];

/* Read/Write Implementations */
inline LoadRequest(loc)
{
    CohChannels[id].Output[loc] ! R(id, loc, 0)
}

inline LoadReply(loc, val)
{
    CohChannels[id].Input[loc] ? R(eval(id), loc, val)
}

inline StoreRequest(loc, val)
{
    CohChannels[id].Output[loc] ! W(id, loc, val)
}

proctype CoherentSwitch(byte myid, myloc)
{
    byte val;

    end: do
        :: CohChannels[myid].Output[myloc] ? R(eval(myid), eval(myloc), −) −>
            SwitchToMem[myloc] ! R(myid, myloc, 0);
        :: CohChannels[myid].Output[myloc] ? W(eval(myid), eval(myloc), val) −>
            SwitchToMem[myloc] ! W(myid, myloc, val);
    od
}

proctype CoherentMemLoc(byte myloc)
{
    byte mem;
    byte id;

    end: do
        :: SwitchToMem[myloc] ? R(id, eval(myloc), −) −>
            CohChannels[id].Input[myloc] ! R(id, myloc, mem)
        :: SwitchToMem[myloc] ? W(id, eval(myloc), mem)
    od
}
proctype Mc()
{
    /* Start All Memory Subprocesses */
    atomic
    {
        byte i = 0, j = 0;
        do
            :: j < MEM -> run CoherentMemLoc(j); j++;
            :: else -> break;
        od;
        j = 0;
        do
            :: i < PROC;
            do
                :: j < MEM -> run CoherentSwitch(i,j); j++;
                :: else -> break;
            od;
            i++;
            j = 0;
            :: else -> break;
        od
    }
}

/* Peterson’s Algorithm */
#define true 1
#define false 0

proctype Peterson(byte id)
{
    bit j = 1 - id;
    bit i = id;
    byte loc, val;

    /* Entry Section */

do
    :: skip;
    StoreRequest(id,i,true); /* flag[i] = true; */
    StoreRequest(id,turn,j); /* turn= j; */
    do
        :: skip;
        LoadRequest(id,turn);
        LoadReply(id,loc,val);
        if
            :: (val == i) -> break /* (turn == i) */
            :: else -> skip
        fi;
    LoadRequest(id,j);
}
LoadReply(id,loc,val);
if
  :: (val == false) -> break /* (flag[j] == false) */
  :: else -> skip
fi;
fi;

/* Critical Section */

progress: skip;

/* Exit Section */
  StoreRequest(id,i,false);
od
}

/* Running All Processes */
init
{
  atomic
  {
    run McO;
    run Peterson(0);
    run Peterson(1);
  }
}