

**FORMALIZING SHARED MEMORY CONSISTENCY  
MODELS FOR PROGRAM ANALYSIS**

by

Yue Yang

A dissertation submitted to the faculty of  
The University of Utah  
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2005

Copyright © Yue Yang 2005

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

## SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Yue Yang

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

---

Chair: Ganesh Gopalakrishnan

---

Co-Chair: Gary Lindstrom

---

Konrad Slind

---

Matthew Flatt

---

Radu Grosu

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

**FINAL READING APPROVAL**

To the Graduate Council of the University of Utah:

I have read the dissertation of                     Yue Yang                     in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

\_\_\_\_\_  
Date

\_\_\_\_\_  
Ganesh Gopalakrishnan  
Chair: Supervisory Committee

\_\_\_\_\_  
Date

\_\_\_\_\_  
Gary Lindstrom  
Co-Chair: Supervisory Committee

Approved for the Major Department

\_\_\_\_\_  
Christopher R. Johnson  
Chair/Director

Approved for the Graduate Council

\_\_\_\_\_  
David S. Chapman  
Dean of The Graduate School

## ABSTRACT

Shared memory consistency models are critical for system correctness but difficult to analyze. The increasing popularity of multithreaded programming also creates a new challenge in how to help programmers reason about thread executions against the underlying memory consistency rules.

This dissertation addresses the problem of formally specifying memory models to support program analysis. Two analytical frameworks are established to support both operational and nonoperational specification styles. These frameworks are applied to formalize a wide range of memory consistency designs, including classical, industrial processor level, and language level memory models.

We first present an operational style specification framework called UMM, which supports a generic memory abstraction and provides built-in model checking capability. Then we present Nemos, a nonoperational specification framework that uses higher order predicate logic to capture memory ordering constraints in a declarative and compositional fashion. After imposing the consistency rules on a symbolic program execution, the validity of the execution can be automatically checked through constraint solving or SAT solving. This method makes a memory model executable, a powerful feature lacking in previous nonoperational methods. Furthermore, a constraint-based approach is applied to specify control/data dependence and to capture “programmer expectations.” By combining the constraints of program properties and language semantics, memory-model-sensitive program analysis can be automated. Three concrete applications of this approach are proposed, including execution validation, race detection and atomicity verification.

To my family.

# CONTENTS

<b>ABSTRACT</b> .....	<b>iv</b>
<b>LIST OF FIGURES</b> .....	<b>x</b>
<b>LIST OF TABLES</b> .....	<b>xi</b>
<b>ACKNOWLEDGMENTS</b> .....	<b>xii</b>
<b>CHAPTERS</b>	
<b>1. INTRODUCTION</b> .....	<b>1</b>
1.1 Motivation .....	1
1.1.1 The need for a memory model .....	1
1.1.2 The importance of a memory model .....	3
1.1.2.1 Impact on software correctness .....	3
1.1.2.2 Impact on compiler optimizations .....	5
1.2 Background .....	6
1.2.1 Semantic description methods .....	6
1.2.1.1 Operational semantics .....	7
1.2.1.2 Axiomatic semantics .....	7
1.2.1.3 Denotational semantics .....	7
1.2.2 Memory model designs .....	8
1.2.2.1 Example .....	10
1.2.3 Processor level memory model issues .....	10
1.2.4 Language level memory model issues .....	11
1.3 Related work .....	12
1.3.1 Memory model specification .....	12
1.3.2 Memory model verification .....	13
1.3.3 Program analysis .....	14
1.4 Contributions .....	15
1.5 Organization .....	16
<b>2. AN OPERATIONAL SPECIFICATION APPROACH</b> .....	<b>18</b>
2.1 Chapter overview .....	18
2.2 UMM overview .....	18
2.2.1 The generic memory abstraction .....	19
2.2.2 Specification method .....	20
2.2.3 Verification method .....	21
2.3 Formalizing memory models using UMM .....	21
2.3.1 Terminology .....	21
2.3.1.1 Instruction .....	21

2.3.2	Initial conditions . . . . .	22
2.3.3	Transition table . . . . .	22
2.3.4	Bypassing table . . . . .	23
2.3.5	Visibility ordering requirements . . . . .	23
2.4	An alternative Java memory model specification . . . . .	24
2.5	The Murphi implementation . . . . .	25
2.5.1	Encoding the transition table . . . . .	25
2.5.2	Encoding the test program . . . . .	26
2.6	Applying UMM for verification . . . . .	26
2.7	Summary . . . . .	30
<b>3.</b>	<b>A NONOPERATIONAL SPECIFICATION APPROACH . . . . .</b>	<b>32</b>
3.1	Chapter overview . . . . .	32
3.2	Nemos overview . . . . .	32
3.3	Formalizing classical memory models . . . . .	33
3.3.1	Terminology . . . . .	33
3.3.1.1	Execution . . . . .	33
3.3.1.2	Operation . . . . .	33
3.3.1.3	Initial write . . . . .	34
3.3.2	A library of memory consistency properties . . . . .	34
3.3.2.1	General ordering rules . . . . .	34
3.3.2.2	Read value rule . . . . .	35
3.3.2.3	Serialization . . . . .	35
3.3.2.4	Ordering relations . . . . .	35
3.3.2.5	Auxiliary predicates . . . . .	36
3.3.3	Five classical memory models . . . . .	36
3.3.3.1	Sequential consistency . . . . .	36
3.3.3.2	Coherence . . . . .	36
3.3.3.3	PRAM . . . . .	37
3.3.3.4	Causal consistency . . . . .	37
3.3.3.5	Processor consistency . . . . .	37
3.4	Formalizing the Intel Itanium memory model . . . . .	38
3.4.1	Terminology . . . . .	38
3.4.1.1	Instructions . . . . .	38
3.4.1.2	Operation tuple . . . . .	38
3.4.1.3	Address attributes . . . . .	39
3.4.2	The Itanium memory ordering rules . . . . .	39
3.4.2.1	General ordering requirement (Appendix B.1) . . . . .	40
3.4.2.2	Write operation order (Appendix B.2) . . . . .	41
3.4.2.3	Program order (Appendix B.3) . . . . .	41
3.4.2.4	Memory-data dependence (Appendix B.4) . . . . .	41
3.4.2.5	Data-flow dependence (Appendix B.5) . . . . .	41
3.4.2.6	Coherence (Appendix B.6) . . . . .	42
3.4.2.7	Read value (Appendix B.7) . . . . .	42
3.4.2.8	Total order of WB releases (Appendix B.8) . . . . .	42
3.4.2.9	Sequentiality of UC operations (Appendix B.9) . . . . .	43
3.4.2.10	No UC bypassing (Appendix B.10) . . . . .	43

3.5	Comparison between operational and nonoperational approaches . . . . .	43
3.6	Summary . . . . .	44
<b>4.</b>	<b>SOLVING MEMORY ORDERING CONSTRAINTS . . . . .</b>	<b>45</b>
4.1	Chapter overview . . . . .	45
4.2	Introduction . . . . .	45
4.3	Overview of the solving method . . . . .	46
4.3.1	The algorithm . . . . .	47
4.3.2	Applying constraint logic programming (CLP) . . . . .	47
4.3.3	Applying boolean satisfiability techniques . . . . .	48
4.4	The CLP approach . . . . .	48
4.5	Applying Nemos for verification . . . . .	50
4.5.1	Analyzing classical memory models . . . . .	50
4.5.2	Analyzing the Itanium memory model . . . . .	50
4.6	The SAT approach . . . . .	52
4.6.1	Initial approach . . . . .	52
4.6.2	Improved approach . . . . .	53
4.7	Performance results . . . . .	53
4.7.1	Analysis of classical memory models . . . . .	54
4.7.2	Analysis of the Itanium memory model . . . . .	54
4.7.3	The improved SAT approach . . . . .	54
4.8	Summary . . . . .	55
<b>5.</b>	<b>MEMORY-MODEL-SENSITIVE PROGRAM ANALYSIS . . . . .</b>	<b>56</b>
5.1	Chapter overview . . . . .	56
5.2	Introduction . . . . .	56
5.2.1	Race detection . . . . .	56
5.2.2	Execution validation . . . . .	57
5.2.3	Atomicity verification . . . . .	58
5.3	Overview of the methodology . . . . .	59
5.4	The source language . . . . .	59
5.4.1	Terminology . . . . .	60
5.4.1.1	Variables . . . . .	60
5.4.1.2	Instruction . . . . .	61
5.4.1.3	Execution . . . . .	61
5.4.1.4	Operation tuple . . . . .	61
5.4.2	Control flow . . . . .	62
5.4.3	Loops . . . . .	63
5.5	Language semantics . . . . .	63
5.5.1	Program order rule . . . . .	64
5.5.2	Read value rules . . . . .	64
5.5.3	Computation rule . . . . .	65
5.5.4	Mutual exclusion rule . . . . .	65
5.5.5	General ordering rules . . . . .	66
5.5.6	Auxiliary definitions . . . . .	66
5.6	Applications . . . . .	67

5.6.1	Execution validation	67
5.6.2	Race detection	67
5.6.3	Atomicity verification	69
5.7	Implementation	70
5.7.1	Constraint generation	70
5.7.2	Concurrency analysis	71
5.7.3	Performance results	72
5.8	Summary	72
<b>6.</b>	<b>CONCLUSIONS</b>	<b>75</b>
6.1	Thesis summary	75
6.1.1	Operational specification techniques	75
6.1.2	Nonoperational specification techniques	75
6.1.3	Reasoning about multithreaded programs	76
6.2	Future directions	76
6.2.1	Framework enhancements	76
6.2.2	Memory-model-sensitive compilers	77
6.2.3	Other application domains	77
<b>APPENDICES</b>		
<b>A.</b>	<b>SPECIFYING <math>JMM_{MP}</math> USING UMM</b>	<b>78</b>
<b>B.</b>	<b>ITANIUM MEMORY ORDERING RULES (IN PREDICATE LOGIC)</b>	<b>85</b>
<b>C.</b>	<b>ITANIUM MEMORY ORDERING RULES (IN HOL)</b>	<b>89</b>
	<b>REFERENCES</b>	<b>93</b>

## LIST OF FIGURES

1.1	The out-of-order effect allowed by the Itanium architecture. . . . .	2
1.2	The double-checked locking idiom. . . . .	4
1.3	An operational view of sequential consistency. . . . .	8
2.1	The generic memory abstraction of UMM. . . . .	19
2.2	Murphi rule that defines the read event. . . . .	25
2.3	Murphi rule that defines the write event. . . . .	26
2.4	Murphi function that defines legalWrite. . . . .	27
2.5	Encoding the test program in Table 2.3 using Murphi. . . . .	29
2.6	Two ways to check a test program in Murphi. . . . .	30
4.1	The process of making an axiomatic memory model executable. . . . .	47
4.2	Adjacency matrices for the execution in Table 4.2 under coherence. . . .	51
4.3	Adjacency matrices for the execution in Table 4.2 under PRAM. . . . .	51
5.1	Are these programs race-free? . . . . .	57
5.2	The processing flow for memory-model-sensitive program analysis. . . .	60
A.1	Extended conceptual architecture for the Java memory model. . . . .	79

## LIST OF TABLES

1.1	Peterson’s algorithm for mutual exclusion. . . . .	3
1.2	Coherence prohibits fetch elimination. . . . .	5
1.3	An execution prohibited by SC and coherence, but allowed by PRAM. . . . .	10
2.1	Transition table for sequential consistency, coherence, and PRAM. . . . .	22
2.2	Bypassing table for sequential consistency, coherence, and PRAM. . . . .	23
2.3	An execution that breaks Peterson’s algorithm. . . . .	28
3.1	The specification hierarchy of the Itanium memory ordering rules. . . . .	40
4.1	An execution allowed by coherence and PRAM but prohibited by PC. . . . .	46
4.2	The execution of the litmus test in Table 4.1. . . . .	50
4.3	An execution resulted from Program <i>b</i> in Figure 1.1. Stores are decomposed into local stores and remote stores. Loads are associated with return values. . . . .	52
4.4	A legal ordering matrix for the execution shown in Table 4.3 when <code>requireProgramOrder</code> is disabled. A possible interleaving <i>8 4 5 6 7 1 2 3</i> is also automatically derived from this matrix. . . . .	52
4.5	Performance summary for the test program in Table 4.1. . . . .	54
4.6	Performance summary for exercising the Itanium memory model. . . . .	55
4.7	Time for generating SAT instances for formula parts $b_1$ and $b_2$ . . . . .	55
5.1	The transactions are not atomic even though the program is race-free. . . . .	58
5.2	The execution derived from program <i>b</i> in Figure 5.1 with $r_1 = 1$ and $r_2 = 0$ . . . . .	61
5.3	The adjacency matrix for the execution shown in Table 5.2 under sequential consistency. . . . .	72
5.4	Performance statistics. . . . .	73
A.1	Transition table for the alternative Java memory model. . . . .	80
A.2	Bypassing table for the alternative Java memory model. . . . .	81

## ACKNOWLEDGMENTS

I would like to thank my primary research advisor Ganesh Gopalakrishnan and co-advisor Gary Lindstrom who taught me how to conduct research. This work would not have been possible without their timely guidance and continuing support. I am grateful to Konrad Slind, Matthew Flatt, and Radu Grosu, who kindly served on my committee. I especially benefited greatly from many fruitful discussions with Konrad Slind. I also appreciate Beat Bruderlin and Kris Sikorski, who supervised my master's program.

Special thanks are given to all contributors to the Java memory model discussion group for their stimulating and thought-provoking discussions. In particular, I thank Bill Pugh, Sarita Adve, Victor Luchangco, Xiaowei Shen, Jan-Willem Maessen, and Jeremy Manson for their insightful suggestions on my work.

I owe a special acknowledge to my wonderful co-workers at LSI Logic: Curtis Dahl, Robert Gaia, Mark Bsharah, Kelly Peterson, Dale Wilbourn, Roger Bryner, Jodi McPherson, and Michael Rally. In particular, I thank my former manager Steve Lynch who provided me the opportunity to pursue my Ph.D. degree while working at LSI Logic.

I am thankful to my fellow graduate students at University of Utah, in particular, Lixin Zhang, Zhen Fang, Xu Ji, Ji Jia, Ravi Hosabettu, Prosenjit Chatterjee, Annette Bunker, Ali Sezgin, Ritwik Bhattacharya, Robert Palmer, Hemanthkumar Sivaraj, Sean McDirmid, and Binu Mathew. In addition, I would like to say "thank you" to everyone in the CS front office, especially Colleen Hoopes, Sara Mathis, Karen Feinauer, and Ann Torrence, for their help through out the years.

I am truly grateful to my parents for their immeasurable love and care. They have always encouraged me to explore my potential and pursue my dreams. Making steady progress in my Ph.D. research while working full-time in industry is a major challenge; this would have been infeasible without the love, encouragement, and

dedication from my wife Wei. I owe my deepest gratitude to her!

The work presented in this dissertation is an extension and modification of the research reported in the following papers:

1. *Analyzing the CRF Java Memory Model* [80];
2. *Specifying Java Thread Semantics Using a Uniform Memory Model* [81];
3. *UMM: An Operational Memory Model Specification Framework with Integrated Model Checking Capability* [83];
4. *Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming and SAT* [84];
5. *Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models* [85];
6. *QB or not QB: An Efficient Execution Verification Tool for Memory Orderings* [38];
7. *Rigorous Concurrency Analysis of Multithreaded Programs* [82].

# CHAPTER 1

## INTRODUCTION

### 1.1 Motivation

With the ever growing demand for performance, parallel computing has become an increasingly attractive technology, exploited both in hardware and software. One of the most important classes of parallel systems is based on the *shared memory* abstraction. Employing a single address space, shared memory is easier to use in most applications in comparison to other communication mechanisms such as message passing. In recent years, many shared memory multiprocessors with aggressive memory architectures have been developed.

#### 1.1.1 The need for a memory model

Unlike a uniprocessor system, which simply requires that each read observes the latest write at the same address as given by the sequential program, a shared memory system may allow certain memory operations to be executed in different orders. A *memory consistency model* (or *memory model*) is needed to specify the legal ordering among memory operations that may be perceived at the user level.

To illustrate this out-of-order effect allowed by a shared memory system, consider two small programs (generally known as *litmus tests*) shown in Figure 1.1, each of which is run concurrently on two Intel Itanium processors. Processor P1 stores the data value 1 into addresses  $a$  and  $b$ . Processor P2 loads the results back from  $b$  and  $a$  into registers  $r1$  and  $r2$ , respectively. The following question arises: if all locations initially contain 0, can the final register values be  $r1 = 1$  and  $r2 = 0$ ? For program  $a$ , where “ordinary” loads and stores are used, the answer is “yes.” Hence, at the user level, the two loads in P2 may appear to be reordered. In contrary, the answer is “no” for program  $b$  since the pair of store-release and load-acquire would impose a stronger ordering restriction. Informally, a store-release instruction will, at

Initially, $a = b = 0$	
P1	P2
st $a, 1$ ;	ld $r1, b$ ;
st $b, 1$ ;	ld $r2, a$ ;

Initially, $a = b = 0$	
P1	P2
st $a, 1$ ;	ld.acq $r1, b$ ;
st.rel $b, 1$ ;	ld $r2, a$ ;

(a)  $r1 = 1$  and  $r2 = 0$  is allowed. (b)  $r1 = 1$  and  $r2 = 0$  is prohibited.

**Figure 1.1.** The out-of-order effect allowed by the Itanium architecture.

its completion, ensure that all previous instructions are completed; a load-acquire instruction correspondingly ensures that all following instructions will complete only after it completes. These explanations are far from precise — what do “previous” and “completion” mean? A formal memory model specification is crucial for precisely capturing these notions and other similar requirements.

Traditionally, memory consistency is only the concern of a group of specialists, e.g., system designers who develop multiprocessor architectures or cache coherence protocols. This, however, is dramatically changed with the widespread use of *multithreading*, a modern programming paradigm for implementing parallel applications based on shared memory. The adoption of multithreading is accelerated by two factors: (i) the availability of various thread libraries (e.g., POSIX and Win32), and (ii) the advent of built-in threading support in programming languages such as Java.

*Threads* are sequences of instructions that can be executed in parallel. Compared with processes, threads are lightweight since they share many common resources. Using threads is an effective way to increase the responsiveness of an application. For example, Java developers routinely rely on threads for structuring their programs, sometimes even without explicit awareness. Like a multiprocessor program, a multithreaded program also needs a memory model (also referred to as *thread semantics*) to define its legal outcome allowed by the underlying threading platform.

Architectural level memory models are usually described in terms of *processors* and *memory locations*. Language level memory models, on the other hand, are often discussed using *threads* and *shared variables*. Since this dissertation addresses memory models at both levels, it uses the two sets of terminology interchangeably

depending on the context of the discussion.

### 1.1.2 The importance of a memory model

Although multithreading is highly useful for developing well-structured and high performance software, it is also notoriously hard to get right. For instance, programmers are torn on the horns of a dilemma regarding the use of synchronization: too much may impact performance and risk deadlock; too little may lead to race conditions and application inconsistency. The growing complexity in multiprocessor machines and the increasing popularity of multithreaded programming have combined to create a big challenge in how to help programmers understand a memory model and reason about thread executions against the underlying consistency rules.

#### 1.1.2.1 Impact on software correctness

Memory consistency requirements can impact system correctness in a fundamental way. *Programming idioms* (commonly used software patterns or algorithms) developed for one memory model may not work for another. Consider, for example, Peterson's algorithm [66] for mutual exclusion shown in Table 1.1. Each processor first sets its own flag indicating its intention to enter the critical section, and then asserts that it is the other processor's turn if appropriate. The eventual value of the variable `turn` determines which processor enters the critical section first. The correctness of this well-known algorithm, however, crucially depends on the allowed scheduling. If both processors can still observe the default flag values while checking

**Table 1.1.** Peterson's algorithm for mutual exclusion.

Initially, <code>flag1 = flag2 = turn = 0</code>	
P1	P2
<pre>flag1 = 1; turn = 2; while(turn == 2 &amp;&amp; flag2 == 1)     ; &lt; critical section &gt; flag1 = 0;</pre>	<pre>flag2 = 1; turn = 1; while(turn == 1 &amp;&amp; flag1 == 1)     ; &lt; critical section &gt; flag2 = 0;</pre>

the loop conditions, they are able to enter the critical section at the same time. Many memory systems with weak ordering restrictions do permit such a behavior due to optimization needs. Consequently, programs relying on Peterson's algorithm would be erroneous on those systems. Therefore, when selecting an existing algorithm, one must make sure that the algorithm is compliant with the target memory model.

The impact of a memory model is not just limited to low-level system software. Subtle implications of a language level memory model often need to be carefully considered for developing robust application software. To illustrate, consider a popular idiom called *double-checked locking* [16], which is designed for creating a *singleton object* (an object that is constructed once, but may be used many times). As shown in Figure 1.2, this programming pattern only uses locking as needed, i.e., when the object is actually created, to avoid unnecessary synchronization overhead. In the synchronization block, the reference is double-checked (which leads to the name of the idiom) to make sure that the object has not been created.

As first discovered by people from the JMM discussion group [3], the double-checked locking algorithm is unsafe under the original Java memory model (JMM) given in Chapter 17 of the Java Language Specification [42]. The reason is that the original JMM does not impose sufficient ordering restrictions for a constructor. On certain weak memory platforms such as the Alpha architecture, under race conditions,

```
class foo {
    private static Helper helper = null;
    public static Helper get() {
        if (helper == null) {
            synchronized(this) {
                if(helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }
}
```

**Figure 1.2.** The double-checked locking idiom.

the write to an object reference can be visible to other threads even before the object construction has completed. As a result, the assignment to `helper` may be visible to other threads before the constructor completes. Hence, uninitialized object fields may be inadvertently accessed.

### 1.1.2.2 Impact on compiler optimizations

Memory ordering rules also dictate what optimizations are allowed. This can be illustrated by the program in Table 1.2 (adapted from [67]). Since both `r1` and `r3` read from `p.x`, a common compiler optimization technique called *fetch elimination* would allow `r3 = p.x` to be replaced by `r3 = r1`. As pointed out in [67], this optimization is prohibited by the original JMM. This is because the original JMM requires *coherence*, i.e., operations involving the same variable must exhibit a total order. Consequently, `r2 = q.x` and `r3 = p.x` cannot be reordered due to the potential aliasing between `p` and `q` — this in effect prohibits fetch elimination.

From these examples, it is clear that memory consistency rules can impact design decisions taken by system designers, application developers, as well as compiler writers. Therefore, a memory model specification needs to be clearly understood by all these groups.

Unfortunately, due to the complexity of advanced memory architectures, practicing designers and programmers face a serious problem in reliably comprehending the memory model specification. For instance, it is well documented that even experts have often misunderstood classical memory models. As revealed in [11], contradictory claims have been made regarding the requirements of a widely cited memory model called *processor consistency*. The difficulty involved in defining and understanding a

**Table 1.2.** Coherence prohibits fetch elimination.

Initially, p.x = 0	
Thread 1	Thread 2
r1 = p.x;	p.x = 1;
r2 = q.x;	q = p;
r3 = p.x;	

complex memory system is also attested to by the revision effort for the Java memory model. The existing JMM was designed to be easy to use, but it has turned out to be poorly understood and seriously flawed. The ongoing revision process has already taken many years but there still remain issues that need to be finalized.

Given the complicated nature of multithreaded programming and the critical role played by a memory model, several important questions arise:

- Is there a way to help programmers verify whether their programs behave properly in a multithreaded environment according to their expectations?
- Is there a way to help compiler writers perform aggressive optimizations while conforming to all memory model constraints?
- Is there a way to help system implementers (e.g., the Java virtual machine developers) support a language level memory model (e.g., the Java memory model) on an architectural level memory system (e.g., the Itanium architecture), in a correct and efficient manner?
- Is there a way to help developers reliably comprehend the differences as well as similarities among various memory model specifications?
- Is there a way to help memory model designers verify their design goals?

These issues are fundamental to system reliability, yet are not adequately addressed by previous methods. Virtually all industrial weak memory models do not have formal specifications. For those that do have formal specifications on paper, they cannot be executed. For those that have machine-readable formal specifications, they typically use implementation-specific data structures and cannot be decomposed into orthogonal components. Furthermore, there does not exist sufficient support for verifying program properties in multithreaded software against the precise thread semantics. Clearly, improvements are in order to address these critical issues. The objective of this dissertation is to lay a theoretical foundation for reasoning about memory models and multithreaded software.

## 1.2 Background

### 1.2.1 Semantic description methods

One of the central issues discussed in this dissertation is how to formally specify a memory consistency model. This involves the task of describing the *semantics*

(or *meaning*) of a shared memory concurrent program in a mathematically rigorous way. Primarily, there are three different and complementary approaches to semantic description: *operational*, *axiomatic*, and *denotational*.

### 1.2.1.1 Operational semantics

Operational semantics describes how a program is interpreted as sequences of computational steps using a machine (either real or virtual). For a concurrent program, there can exist different execution sequences and return values due to nondeterminism. One common way used to define operational semantics is to provide a state transition system, in which the update of the machine's global state is defined step-by-step with the execution of each program statement. Since operational semantics often mirrors a system's behavior, it can be very useful for system implementers and compiler writers. However, operational semantics tends to emphasize the *how* aspects through its use of machine-dependent data structures, not the *what* aspects that a formal specification is supposed to stress.

### 1.2.1.2 Axiomatic semantics

Axiomatic semantics employs logical formulae to describe states as well as the evolution of states. These formulae are sometimes called *predicates* or *assertions*. When defining the semantics of language syntax structures, an axiomatic specification usually asserts a precondition and postcondition for every program statement. In the context of memory model definitions, an axiomatic approach is generally referred to as a *nonoperational* approach. It is a rule-based specification method which directly defines the desired consistency properties as axioms. The nonoperational approach is often more intuitive due to its declarative nature. However, it does not directly help one build a mental process regarding how the desired properties can be achieved.

### 1.2.1.3 Denotational semantics

The idea behind denotational semantics is to use a valuation function to map the meaning (denotation) of a programming construct to its corresponding mathematical object. Denotational semantics is more abstract than operational semantics as it defines state changes using mathematical functions; it is more concrete than axiomatic

semantics as it specifies explicit meaning instead of properties. The denotational approach is solidly built on recursive function theory and provides an excellent means for precisely defining language semantics, but its usefulness is limited due to its complexity.

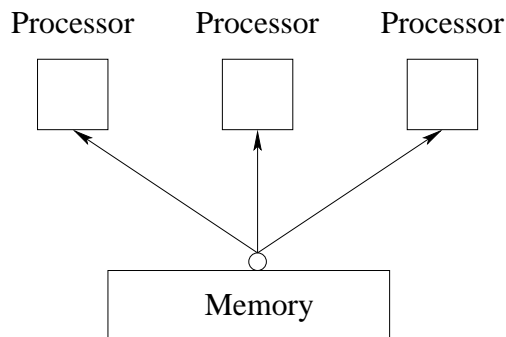
This dissertation concentrates on specification techniques with operational and axiomatic specification styles, with a particular emphasis on the support for formal verification.

### 1.2.2 Memory model designs

Numerous memory models, motivated by both academic and industrial settings, have been developed over the past three decades. The design of a shared memory system typically involves a tradeoff between programmability and efficiency.

One of the strongest models is *sequential consistency (SC)* [50], which is proposed as a natural extension of the sequential model. Informally, sequential consistency can be described operationally with the help of an abstract machine, such as the one shown in Figure 1.3. In this abstract machine, each processor executes one sequential program and these processors are connected to a memory module through a switch. At each step, the switch nondeterministically selects one processor, and the chosen processor completes an operation according to the program order, i.e., the operation order determined by software.

Alternatively, the requirements of sequential consistency can be summarized as three rules following a nonoperational approach: (i) operations of all processors can



**Figure 1.3.** An operational view of sequential consistency.

exhibit a total order, (ii) operations of each individual processor appear in this total order following the program order, and (iii) a read observes the latest write on the same variable according to this total order.

Since sequential consistency is very restrictive, many weaker memory models (see [8] for a survey) have been proposed to enable commonly used optimization techniques. For example, *coherence* (also known as *cache consistency*) [34] only enforces SC on a per-address basis. That is, coherence requires all operations involving the same address to exhibit a total order that also respects program order. *Pipelined Random Access Memory* (PRAM) [53] allows each observing processor to perceive its own view of memory operation order. Informally, PRAM requires that there exists an execution sequence for every processor, containing all operations from the observing processor and all write operations from other processors, such that it exhibits a total order that also respects program order. *Causal consistency* [12] follows a similar policy as in PRAM while also enforcing the causal order resulting from data flow. *Processor consistency* (PC) [11] (according to Goodman’s definition) combines coherence and PRAM in a mutually consistent manner.

Modern shared memory systems are often based on *hybrid* models, meaning programmers can apply special synchronization operations in addition to normal data operations such as reads and writes. In *lazy release consistency* [48], synchronization is performed by *release* and *acquire* operations. When release is performed, previous memory activities from the issuing processor need to be written to the shared memory. A processor reconciles with the shared memory to obtain the updated data when acquire is issued. Lazy release consistency requires coherence. This requirement is further relaxed by *location consistency* [32]. Operations in location consistency are only partially ordered if they are from the same processor or if they are synchronized through locks.

In general, memory models can be categorized according to their visibility ordering requirements. *Visibility order* (similar to the notions of “before order” in [51] and “visibility order” in [7]) is the final observable order of memory operations perceived by one or more processors. Early memory models, such as sequential consistency, were designed for single bus systems, where a common visibility order is enforced for

all observing processors. Some other models, such as PRAM, allow each individual processor to observe its own visibility order.

### 1.2.2.1 Example

The outcome of the program in Table 1.3 is not allowed by sequential consistency and coherence since there does not exist a common visibility order that can be agreed upon by both processors. However, the behavior is permitted by PRAM because each thread can perceive its own visibility order. That is, processor 1 and processor 2 may observe interleaving 1 3 2 and 3 1 4, respectively.

### 1.2.3 Processor level memory model issues

Modern shared memory architectures are highly complex. They employ sophisticated performance enhancing techniques such as superscalar pipelining, branch predication, and speculative execution. Contemporary multiprocessors support a rich set of memory operations to provide the flexibility needed by software. The Intel Itanium processor family, for example, provides two varieties of loads and stores in addition to fence and semaphore instructions, each associated with different ordering restrictions.

The original Itanium memory ordering specification was informally given in various places in the Itanium architecture manual [2]. Intel later provided an application note [7] to guide system developers. This document outlines more than 30 pages of memory ordering rules based on informal mathematics and a large amount of special notations. Since combining various rules may lead to nonobvious consequences, a collection of sample executions are given to illustrate different aspects of the model.

**Table 1.3.** An execution prohibited by SC and coherence, but allowed by PRAM.

Initially, a = 0	
P1	P2
(1) a = 1;	(3) a = 2;
(2) r1 = a;	(4) r2 = a;
Result: r1 = 2 and r2 = 1	

These example programs are helpful, but they cannot be automatically analyzed.

#### 1.2.4 Language level memory model issues

Java is the first widely deployed programming language that provides built-in threading support at the language level. The Java memory model (JMM) is a critical component of the Java threading system since it imposes significant implications on a broad range of activities, such as programming pattern development, compiler optimization, and Java virtual machine (JVM) implementation. Unfortunately, developing a rigorous and intuitive Java memory model has turned out to be much more difficult than first thought.

Compared with a processor level memory model, the design of the Java memory model faces many new challenges. First, it must maintain safety guarantees that cannot be compromised even under race conditions. Second, the JMM needs to be ported to many different processor architectures, some of which have very weak memory ordering restrictions. For the JMM designers, it is a nontrivial task to identify the optimal requirements that can be supported by all major architectures in a safe and efficient manner; for the JVM implementers, the compliance with the JMM needs to be ensured. Third, the Java memory model needs to address the semantics of many programming language constructs, such as final fields, volatile fields, constructors, and finalizers. Finally, run-time effects such as those caused by aliasing must be carefully considered because they can introduce more subtle implications.

The original Java memory model is given in Chapter 17 of the Java Language Specification [42], in which Java thread semantics is defined by eight different actions that are constrained by a set of informal rules. Due to the lack of rigor in specification, the design has several flaws. As pointed out by Pugh [67], the original JMM is both *too weak* and *too strong*. For example, it is too weak regarding the ordering constraints for a constructor (as illustrated by the double-checked locking idiom in Figure 1.2); and it is too strong because many useful compiler optimizations are prohibited (as shown in Table 1.2). Some of the major issues are listed as follows.

- The model requires coherence. Because of this restriction, important compiler optimizations such as fetch elimination are prohibited.

- The model requires a thread to flush all variables to main memory before releasing a lock, imposing a strong restriction on visibility order. Consequently, some seemingly redundant synchronization operations, such as thread local synchronization blocks, cannot be optimized away.
- The ordering guarantee for a constructor is not strong enough. On weak memory architectures such as Alpha, uninitialized fields of an object can be observable under race conditions even after the object reference is initialized and made visible to other threads. This problem opens a security hole to malicious attacks via race conditions (see [67] for details).
- Semantics of *final* variable operations is omitted.
- *Volatile* variable operations do not have synchronization effects on normal variable operations. As a result, volatile variables cannot be applied as synchronization flags to indicate the completion of nonvolatile variable operations.

Several efforts [37, 43] have been conducted to formalize the original Java memory model. Improved Java thread semantics have also been proposed to replace the original one. One proposal is from Manson and Pugh [56, 57], another proposal is from Maessen, Arvind, and Shen [54] based on the Commit/Reconcile/Fence (CRF) framework [74]. These two specifications are referred to as  $\text{JMM}_{\text{MP}}$  and  $\text{JMM}_{\text{CRF}}$ , respectively, in this dissertation. Most recently, Manson, Pugh and Adve announced a new Java memory model draft [3], which made both  $\text{JMM}_{\text{MP}}$  and  $\text{JMM}_{\text{CRF}}$  obsolete. The Java memory model is still under an official revision process [4] and there is an ongoing discussion through the JMM discussion group [3].

## 1.3 Related work

### 1.3.1 Memory model specification

Formalizing memory consistency models has been a topic of extensive research in the past decade. Collier [21] developed a formal theory of memory ordering rules. Using methods similar to Collier’s, Gharachorloo [34] described a generic framework for specifying the implementation conditions of different memory models. Adve [9] proposed a methodology to categorize memory models as various data-race-free classes. Mosberger [60] surveyed and classified several common models.

Kohli et al. [49] proposed a formalism for capturing memory consistency rules and identified parameters that could be varied to produce new models. Raynal et al. [68] provided formal definitions for the underlying consistency models for distributed shared memory (DSM) systems. Bernabu-Aubn et al. [15] and Higham et al. [45] proposed different specification frameworks to facilitate memory model comparison. Magalhes et al. [55] formalized a set of memory models, including several hybrid ones. Steinke et al. [75] developed a framework that captures the relationship among several existing models based on the idea of orthogonal consistency properties.

These efforts have tremendously enhanced the clarity of many memory models, but they all suffer from a common problem: it is hard to experiment with these specifications since they are *passive* objects and do not support automated verification. In our experience, without the ability to execute the underlying specification, serious bugs can lurk in programs, which are hard for human minds to discover. The ability to carry out automatic exhaustive analysis, albeit on small finite executions, is extremely helpful for finding these bugs.

Melo et al. [58] described a visual tool for displaying legal execution histories for SC and PRAM. Permitted executions are selected from a tree resulting from an enumeration of all possible interleavings. Memory model properties are checked through a depth first search. Their work did not address memory model specification techniques, hence only simple memory model constraints can be checked. They also rely on a straightforward search strategy, which cannot take advantage of the latest development in backtracking and state pruning techniques.

Lamport and colleagues have specified the Alpha and Itanium memory models in TLA+ [6] [47]. These specifications build visibility orders inductively and support the execution of litmus tests. Their approach also precisely specifies the ordering requirement, but the manner in which such inductive definitions are constructed will vary from memory model to memory model, making comparative analysis harder.

### 1.3.2 Memory model verification

Model checking has been used for analyzing operational style memory models. Park and Dill [25, 64] proposed to integrate a model checker with the Sparc *Relaxed*

*Memory Order* [79] specification to make the memory model executable. In our work [80] on the analysis of  $\text{JMM}_{\text{CRF}}$ , we extended this methodology to the domain of JMM and demonstrated its feasibility and effectiveness for analyzing language level thread semantics. After adapting  $\text{JMM}_{\text{CRF}}$  to an equivalent executable specification implemented in Murphi, we systematically exercised the underlying model with a suite of test programs to reveal pivotal properties and verify common programming idioms, such as the double-checked locking algorithm. Roychoudhury and Mitra [71] also applied techniques similar to [80] to study the existing Java memory model. They formalized the existing JMM with an operational representation using a local cache and a set of read/write queues and implemented the specification with an invariant checker. Although [64, 80, 71] have improved their respective target models by making them executable, they are limited to the specific designs from the original proposals. As a result, the intuitions behind the memory consistency requirements based on those notations are not immediately apparent. Furthermore, the complexity of the special data structures demands more memory consumption during model checking, which would worsen the state space explosion problem. In [19], a formal operational framework was developed to verify protocol implementation against weak memory models using model checking. In that framework, however, data structures of the transition system vary depending on whether a single visibility order or multiple visibility orders need to be defined. These variations make it difficult to create a parameterized analysis tool.

### 1.3.3 Program analysis

Formal methods have been applied for verifying multithreaded programs. Extensive research has been done in model checking Java programs, e.g. [44, 77, 22, 23, 63]. These tools, however, assume sequentially consistent program executions and do not address memory model issues. Therefore, they perform only “memory-model-insensitive” analysis and cannot analyze fine-grained thread interleavings.

Constraint solving was historically applied in AI planning problems. In recent years, it has started to show a lot of potential for program analysis as well. For example, constraints are used in [13] to analyze programs written in a factory control

language called Relay Ladder Logic. A constraint system is developed in [31] for inferring static types for Java bytecode. The work in [70] performs points-to analysis for Java by employing annotated inclusion constraints. Flanagan [27] proposed to use CLP for bounded software model checking. This dissertation applies constraint solving to a new problem domain, i.e., memory-model-sensitive program analysis.

There is a large body of work on race detection, which can be classified as *static* or *dynamic* analysis. The latter can be further categorized as *on-the-fly* or *post-mortem*, depending on how the execution information is collected. Netzer and Miller [61] proposed a detection algorithm using the post-mortem method. Adve and Hill proposed the *data-race-free* model [9] and developed a formal definition of data races under weak memory models [10]. Lamport’s happened-before relation has been applied in dynamic analysis tools, e.g., [26, 62, 65]. Several on-the-fly methods, e.g., [59, 69, 73], exploited information based on the underlying cache coherence protocol. The drawback of these dynamic techniques is that they can easily miss a data race, depending on how threads are scheduled.

Some race detectors, e.g., [72, 78, 20], were designed specifically for the lock-based synchronization model. Tools such as ESC/Java [29] and Warlock [76] rely on user-supplied annotations to statically detect data races. Type-based approaches, e.g., [28, 14, 17], have also been proposed for object-oriented programs. Although these tools are effective in practice, they do not address the issue that this dissertation focuses on, which is how to rigorously reason about multithreaded programs running in a complex shared memory environment.

Flanagan and Qadeer [30] developed a type system to enforce atomicity based on Lipton’s theory of right and left movers [52]. Since a race analysis is required to be performed in advance, the effectiveness of their approach depends on the accuracy of the race detector.

## 1.4 Contributions

The key contributions of this dissertation are as follows:

1. An operational style specification framework called UMM (Uniform Memory Model). The UMM framework supports a generic memory abstraction and

provides the built-in model checking capability. By employing a generic memory abstraction, it can capture a large collection of memory models as guarded commands in a uniform notation. By integrating a model checker, it enables formal reasoning about thread interleavings. With this framework, memory models can be specified in a parameterized style. Several well-known memory models as well as a replacement proposal for the Java memory model are formalized and analyzed.

2. A nonoperational style specification framework called Nemos (Nonoperational yet Executable Memory Ordering Specifications), which makes a memory model both *declarative* and *executable*. The Nemos framework uses higher order predicate logic to capture a complete set of ordering constraints on symbolic executions. It further applies constraint solving and SAT solving to make such specifications executable, a powerful feature lacking in previous nonoperational approaches. A collection of classical memory models are defined and compared using the Nemos framework. A concise specification of the Intel Itanium memory ordering rules is also provided to demonstrate the scalability of this approach. In addition, a comparison between our operational and nonoperational specification styles is presented.
3. A constraint-based approach to conducting memory-model-sensitive program analysis. This method relies on precise definitions of (i) programmer expectations, (ii) the interthread memory consistency rules, and (iii) the intrathread program semantics. By putting all these components together, this approach offers a unique advantage for program verification — it provides a precise and exhaustive coverage for all thread interleavings under any given memory model. Three concrete applications of this approach are proposed, including execution validation, race detection, and atomicity analysis.

## 1.5 Organization

The following chapters are organized as follows. Chapter 2 presents the UMM framework. After giving an overview, it formalizes sequential consistency, coherence, and PRAM to demonstrate the general strategies employed by UMM. It then

presents an alternative replacement proposal for the Java memory model. Finally, it demonstrates how to conduct memory model verification using the UMM framework.

Chapter 3 describes the Nemos framework. It first outlines the specification method applied by Nemos. Then a collection of classical memory models are formalized. It is followed by a formal definition of the Intel Itanium memory ordering rules. Lastly, the comparison between Nemos and UMM is given.

Chapter 4 presents various solving methods to make a nonoperational specification executable. In particular, the prototype implementations using constraint logic programming and SAT solving are discussed.

Chapter 5 discusses how to support rigorous program analysis for multithreaded software. Sequential consistency is presented for a Java-like source language. Critical program properties are formulated to enable three common analyses.

Chapter 6 summarizes this dissertation and discusses future directions. Specification details are presented in the appendices.

# CHAPTER 2

## AN OPERATIONAL SPECIFICATION APPROACH

### 2.1 Chapter overview

This chapter presents the UMM (Uniform Memory Model)<sup>1</sup> framework. Sequential consistency, coherence, and PRAM are defined and analyzed using UMM to illustrate key techniques exploited by the framework. As a case study, an executable specification is developed in UMM for a Java memory model replacement proposal. Its detailed specification is presented in Appendix A.

### 2.2 UMM overview

The objectives of the UMM framework are twofold: (i) to enhance verification capability for a memory model design, and (ii) to support a generic memory abstraction so that complexities associated with architecture-specific data structures can be eliminated.

Inspired by Park and Dill’s work [25, 64] on the memory model of the Sparc architecture, the UMM system adopts a guarded-command specification style and uses the description language of a model checker to encode the definition. With this model checking capability, the UMM framework can exhaustively exercise a test program to cover all thread interleavings.

UMM also employs a simple and generic transition system to define memory operations. The main insight is that by using only two kinds of buffers, namely *local instruction buffer* (LIB) and *global instruction buffer* (GIB), one can separately capture requirements on program order and visibility order, two pivotal properties

---

<sup>1</sup>The term “uniform models” (in contrast to “hybrid models”) has occasionally been used for memory models without special synchronization instructions. To avoid confusion, UMM can also be referred to as *Unified Memory Model*.

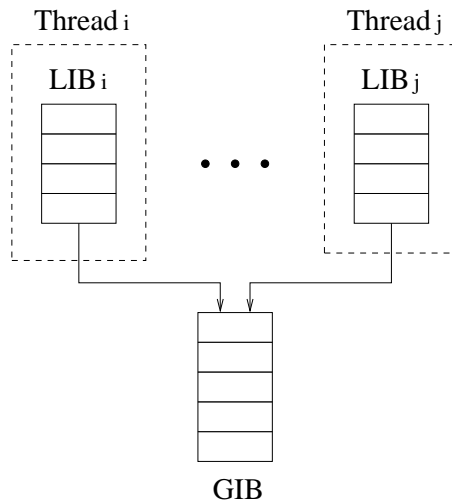
for understanding thread behaviors. Relaxations of the program order are configured through a bypassing table and rules in first-order logic are used to express these bypassing policies. Completed instructions in **GIB** are used to construct the legal visibility order subject to certain visibility ordering rules. With this approach, variations between memory models can be isolated into a few well-defined places such as the bypassing table and the visibility ordering rules, enabling easy configuration and comparison.

### 2.2.1 The generic memory abstraction

The UMM framework consists of an abstract transition system with an associated transition table. The conceptual architecture of the transition system, as illustrated in Figure 2.1, is based on a generic memory abstraction. Each thread  $k$  has a local instruction buffer  $\text{LIB}_k$ , which stores all pending memory operations in program order. Threads interact through a global instruction buffer **GIB**, which stores all previously completed memory operations that are necessary for fulfilling a future read request.

Compared with other real or conceptual memory architectures, the memory abstraction of UMM has two major differences:

1. In contrast to most processor level memory architectures that apply a cache layer between processors and the main memory, UMM only uses two layers —



**Figure 2.1.** The generic memory abstraction of UMM.

one for thread local information and the other for global trace information. This simple memory abstraction eliminates unnecessary complexities and clarifies the essential semantics of the shared memory system.

2. Instead of a fixed-size main memory, UMM applies a global instruction buffer whose size may be increased if necessary. This feature offers the flexibility for specifying certain relaxed memory models that require recording the history of multiple writes on a given variable.

### 2.2.2 Specification method

In UMM, semantics of memory operations are precisely defined as *guarded commands*, where memory operations are categorized as *events* that may be completed by carrying out some *actions* when certain *conditions* are satisfied. At a given step, any eligible event may be nondeterministically chosen and atomically completed by the abstract machine.

For clarity, the framework applies a bypassing table called **BYPASS** to configure the ordering policy for issuing instructions. These bypassing rules used in the instruction selection process serve as an *enabling mechanism*: (i) they impose an interleaving close to the memory model requirement, and (ii) they presciently enable certain operations when needed. Completed instructions in **GIB** are used to form the legal visibility order. The visibility ordering rules are imposed as a final *filtering mechanism* to guarantee proper *serialization*, i.e., a read returns the value from the latest write on the same variable.

A memory model  $\mathcal{M}$  defines what the *legal executions* of a given program are. Intuitively, a legal execution is a sequence of permissible actions from various threads that also forms a proper serialization. An actual implementation of  $\mathcal{M}$ ,  $\mathcal{I}_{\mathcal{M}}$ , may choose different architectures and optimization techniques as long as the legal executions allowed by  $\mathcal{I}_{\mathcal{M}}$  are also permitted by  $\mathcal{M}$ .

The operational definition in UMM uses rules expressed in first-order logic to capture details. Thus, in a sense, it has a dual status: the big picture is captured operationally, while the details are captured in a declarative manner. This style is also found in some related efforts, e.g., [33]. Here, the contributions are twofold: UMM employs this style for a wide spectrum of memory models using its generic

memory abstraction; and UMM integrates a model checking tool with the specification framework to support rigorous analysis.

### 2.2.3 Verification method

To make a memory model specification executable, the specification is encoded in Murphi [24], a description language with a syntax similar to C as well as a model checking system that supports exhaustive state space enumeration. Since Murphi naturally supports specifications based on guarded commands, this encoding process is relatively straightforward. The first-order logic style of the formal specification is retained in the Murphi model (Murphi supports first-order logic quantifiers, which are unravelled through state enumeration). This helps make the translation process reliable. With the model checking capability from Murphi, the UMM system can detect deadlocks and invariant violations.

## 2.3 Formalizing memory models using UMM

UMM can be configured to specify a large collection of memory models. The general strategy is to customize the bypassing table to control thread interleavings and impose proper visibility ordering constraints on the operations in GIB. This configuration process is typically trivial for memory models involving a common visibility order. If a model requires per-thread visibility orders, a write operation is decomposed into multiple subwrite operations targeting each thread so that the single GIB can be used to retrieve a unique visibility order for every observing thread. This technique is inspired by similar methods in [7, 21].

In this section, sequential consistency and coherence are presented to show the process of defining models with a single visibility order. PRAM serves as an example for models requiring per-thread visibility orders. Most other memory models can be defined in a similar way.

### 2.3.1 Terminology

#### 2.3.1.1 Instruction

For the memory models discussed in this section, an instruction  $i$  is represented by a tuple  $\langle t, pc, op, var, data, target, time \rangle$ , where

$t(i) = t$ :	issuing thread;
$pc(i) = pc$ :	program counter;
$op(i) = op$ :	operation type, can be <i>Read</i> or <i>Write</i> ;
$var(i) = var$ :	variable;
$data(i) = data$ :	data value;
$target(i) = target$ :	target thread observing a write;
$time(i) = time$ :	global time stamp, <i>incremented each time</i> when an instruction is added to GIB.

### 2.3.2 Initial conditions

LIB initially contains all instructions from each thread in program order. For sequential consistency and coherence, writes do not need to be decomposed. For PRAM, a write  $i$  is converted to a set of subwrite instructions for each thread  $k$  ( $target(i) = k$ ), including the issuing thread. The subwrite instructions that originate from the same write share the same program counter  $pc$ . GIB initially contains the default write instructions for every variable  $v$  (with the default value of  $v$ , a special thread ID  $t_{init}$ , and a  $time$  field of 0). After the abstract machine is initialized, it operates according to the transition table.

### 2.3.3 Transition table

Sequential consistency, coherence, and PRAM can all be specified by the same transition table given in Table 2.1. A read instruction completes when the return value is bound. A write instruction completes when it is added to GIB. A multi-threaded program terminates when all instructions from all threads complete.

**Table 2.1.** Transition table for sequential consistency, coherence, and PRAM.

Event	Condition	Action
read	$\exists i \in \text{LIB}_{t(i)} :$ $ready(i) \wedge op(i) = \text{Read} \wedge$ $(\exists w \in \text{GIB} : legalWrite(i, w))$	$i.data := data(w);$ $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i);$
write	$\exists i \in \text{LIB}_{t(i)} :$ $ready(i) \wedge op(i) = \text{Write}$	$\text{GIB} := append(\text{GIB}, i);$ $\text{LIB}_{t(i)} := delete(\text{LIB}_{t(i)}, i);$

### 2.3.4 Bypassing table

Table 2.2 outlines the bypassing table **BYPASS** for sequential consistency, coherence, and PRAM, where an entry  $\text{BYPASS}[op1][op2]$  determines whether an instruction with type  $op2$  can bypass a previous instruction with type  $op1$ . Values used in table **BYPASS** include *Yes*, *No*, *DiffVar*, and *DiffTgt*. Informally, *Yes* permits the bypassing, *No* prohibits it, *DiffVar* conditionally enables the bypassing only if the variables are different and not aliased, and *DiffTgt* conditionally enables the bypassing when a subwrite targeting a different thread is involved.

According to Table 2.2, no bypassing is allowed for sequential consistency. For coherence, instructions operated on different variables can be issued out of order. For PRAM, two subwrites targeting different threads can be reordered, and a subwrite can be reordered with a read if the subwrite targets another thread. These bypassing rules are precisely defined in condition *ready*.

$$\begin{aligned}
 \text{ready}(i) \equiv & \\
 & \neg \exists j \in \text{LIB}_{t(i)} : pc(j) < pc(i) \wedge \\
 & (\text{BYPASS}[op(j)][op(i)] = \text{No} \vee \\
 & \text{BYPASS}[op(j)][op(i)] = \text{DiffVar} \wedge var(j) = var(i) \vee \\
 & \text{BYPASS}[op(j)][op(i)] = \text{DiffTgt} \wedge op(j) = \text{Write} \wedge op(i) = \text{Read} \wedge target(j) = t(i) \vee \\
 & \text{BYPASS}[op(j)][op(i)] = \text{DiffTgt} \wedge op(j) = \text{Read} \wedge op(i) = \text{Write} \wedge t(j) = target(i) \vee \\
 & \text{BYPASS}[op(j)][op(i)] = \text{DiffTgt} \wedge op(j) = \text{Write} \wedge op(i) = \text{Write} \wedge \\
 & target(j) = target(i))
 \end{aligned}$$

### 2.3.5 Visibility ordering requirements

Condition *legalWrite* is a guard for read events that guarantees the serialization requirement. It states that a write  $w$  is not eligible for a read  $r$  if there exists an overwriting write  $w'$  between  $r$  and  $w$  in the ordering path. The *legalWrite* definition

**Table 2.2.** Bypassing table for sequential consistency, coherence, and PRAM.

	SC		coherence		PRAM	
2nd $\Rightarrow$	Read	Write	Read	Write	Read	Write
1st $\Downarrow$						
Read	No	No	DiffVar	DiffVar	No	DiffTgt
Write	No	No	DiffVar	DiffVar	DiffTgt	DiffTgt

of PRAM is only slightly different from that of sequential consistency and coherence because a reading thread  $t$  can only observe subwrites targeting  $t$  or the default writes.

For sequential consistency and coherence:

$$\begin{aligned} \text{legalWrite}(r, w) \equiv & \\ & op(w) = \text{Write} \wedge var(w) = var(r) \wedge \\ & (\neg \exists w' \in \text{GIB} : op(w') = \text{Write} \wedge var(w') = var(r) \wedge \\ & time(r) > time(w') \wedge time(w') > time(w)) \end{aligned}$$

For PRAM:

$$\begin{aligned} \text{legalWrite}(r, w) \equiv & \\ & op(w) = \text{Write} \wedge var(w) = var(r) \wedge (target(w) = t(r) \vee t(w) = t_{init}) \wedge \\ & (\neg \exists w' \in \text{GIB} : op(w') = \text{Write} \wedge var(w') = var(r) \wedge target(w') = t(r) \wedge \\ & time(r) > time(w') \wedge time(w') > time(w)) \end{aligned}$$

The above specifications of SC, coherence, and PRAM clearly illustrate a key feature of the UMM framework — memory models are specified in a *parameterized* style, meaning designers can simply redefine a few bypassing rules (defined in condition *ready*) and visibility ordering rules (defined in condition *legalWrite*) to obtain an executable specification for another memory model.

## 2.4 An alternative Java memory model specification

In [81], the UMM framework has been applied to formalize and analyze  $\text{JMM}_{\text{MP}}$ . The detailed UMM specification of  $\text{JMM}_{\text{MP}}$  is provided in Appendix A to (i) illustrate how to resolve some of the language level memory model issues, such as the treatment of local variables, (ii) show how synchronization operations may be defined using UMM, and (iii) demonstrate that UMM can be scaled up for analyzing complex memory model designs.

The memory abstraction is extended to track the status of local variables and locks. Java global variables are further categorized as a *normal*, *volatile*, or *final* variables. An auxiliary *freeze* operation is introduced for defining ordering constraints at constructor boundaries. Nine events are defined in the transition table.

## 2.5 The Murphi implementation

To support verification, a memory model is encoded using Murphi. The Murphi program consists of two parts: the first part implements the formal specification of a memory model; the second part comprises a collection of idiom-driven test programs. When a test program is executed under the guidance of the UMM transition system, the Murphi model checker exhaustively exercises all possible executions allowed by the memory model.

### 2.5.1 Encoding the transition table

The transition table is specified as Murphi rules. For example, the read and write events are defined in Figure 2.2 and Figure 2.3, respectively. Bypassing and visibility ordering conditions are implemented as Murphi procedures. Figure 2.4 illustrates the implementation of `legalWrite`.

```

Ruleset t:ThreadId_t Do      /* for any thread */
Ruleset i:Label_t Do        /* for any read in LIB */
Ruleset k: GibEnt_t Do      /* for any previous write in GIB */
Alias
  thread: Thread[t];
  r: thread.LIB[i];
  w: GIB.arr[k];
Do
  Rule "Read"
    inBound(i, thread.len) & notComplete(r) & ready(r) &
    (r.op = READ) & k < GIB.len & legalNormalWrite(r, w)
    ==>
  Begin
    r.data := w.data;
    r.done := True;
  End;
End; /* Alias */
End; /* Ruleset k */
End; /* Ruleset i */
End; /* Ruleset t */

```

**Figure 2.2.** Murphi rule that defines the read event.

```

Ruleset t:ThreadId_t Do      /* for any thread */
Ruleset i:Label_t Do        /* for any write in LIB */
Alias
  w: Thread[t].LIB[i];
Do
  Rule "write"
    inBound(i, Thread[t].len) & notComplete(w) & ready(w) &
    w.op = Write
  ==>
  Begin
    appendGIB(w);
    w.done := True;
  End;
End; /* Alias */
End; /* Ruleset i */
End; /* Ruleset t */

```

**Figure 2.3.** Murphi rule that defines the write event.

### 2.5.2 Encoding the test program

A test program, defined by specific Murphi initial state and invariants, consists of small concurrent programs comprised of two or more threads. Each test program is designed to reveal a certain memory model property or to simulate a common programming idiom.

The Murphi implementation is highly configurable, allowing one to easily set up test programs, UMM abstract machine parameters, and memory model properties. The scale of the system can be adjusted for different tests. For example, key parameters can be declared as follows:

```

Const
  THREAD_NUM:      2;      /* max number of threads */
  VAR_NUM:         3;      /* max number of global variables */
  VALUE_NUM:       2;      /* data range */
  LABEL_NUM:       6;      /* max number of instructions per thread */

```

## 2.6 Applying UMM for verification

The model checking capability enables one to exhaustively analyze a given execution of a test program. This feature can be very useful in several applications: (i) it can reveal critical properties of a memory model, (ii) it can help a user identify

```

Function legalWrite(r:Ins_t; w:GibIns_t): Boolean; Begin
  Switch MyMemModel
  case MM_SC:
    Return
    (
      w.op = WRITE & sameVar(w.v, r.v) &
      !(Exists k: GibEnt_t Do
        k < GIB.len & GIB.arr[k].op = WRITE &
        sameVar(GIB.arr[k].v, r.v) &
        GIB.len > GIB.arr[k].time & GIB.arr[k].time > w.time)
      End)
    );

  case MM_PRAM:
    Return
    (
      w.op = WRITE & sameVar(w.v, r.v) &
      (w.target = r.thread | w.thread = Tinit) &
      (sameThread(w.thread, r.thread) -> (w.pc < r.pc)) &
      !(Exists k: GibEnt_t Do
        k < GIB.len & GIB.arr[k].op = WRITE &
        sameVar(GIB.arr[k].v, r.v) &
        GIB.arr[k].target = r.thread &
        GIB.len > GIB.arr[k].time & GIB.arr[k].time > w.time)
      End)
    );
  End;
End;

```

**Figure 2.4.** Murphi function that defines legalWrite.

potential flaws for common programming patterns, and (iii) it facilitates formal comparative analysis for different memory models.

In [81], a suite of test programs is designed to reveal interesting properties of  $JMM_{MP}$ , including the ordering property, synchronization property, and constructor property. With our previously developed executable model [80] for  $JMM_{CRF}$ , we have also performed a comparative analysis between  $JMM_{MP}$  and  $JMM_{CRF}$  by running the same test programs. This helps one understand the subtle differences between the two models. Even though both  $JMM_{MP}$  and  $JMM_{CRF}$  are now obsolete, these verification efforts offer interesting case studies that demonstrate the effectiveness of

formal methods for evaluating various proposals and foreseeing pitfalls.

In [83],  $\text{JMM}_{\text{MP}}$  is compared with several well-known memory models, and the UMM system formally proves that  $\text{JMM}_{\text{MP}}$  does not require coherence, causal consistency, and write atomicity for normal variable operations. For example, recall the test program in Table 1.3, in which the result of interest should be prohibited by coherence. If this test is executed following the UMM definition of  $\text{JMM}_{\text{MP}}$ , the system immediately finds a legal interleaving that leads to such a result. This quickly proves that coherence is not enforced by  $\text{JMM}_{\text{MP}}$ .

The executable specifications coded in Murphi can also help one analyze common programming patterns against different memory models. For example, consider Peterson’s algorithm shown in Table 1.1. Table 2.3 abstracts the key memory operations from Peterson’s algorithm and illustrates a specific thread behavior that breaks the algorithm: if both threads can still observe the default flag values while checking the loop conditions, they are able to enter the critical section at the same time.

Figure 2.5 shows the Murphi encoding for the test program in Table 2.3. The test program is set up in Murphi `startstate`. A procedure `add` is used to add instructions to the `LIB` for the corresponding thread. To examine test results, two techniques can be applied. These techniques are shown in Figure 2.6 (local variables are used in this example, see Appendix A for details). The first one uses Murphi invariants to specify that a particular scenario can never occur. If it does occur, a violation trace can be generated to help understand the cause. The second technique uses a special “thread completion” rule, which is triggered when all threads are completed, to output all

**Table 2.3.** An execution that breaks Peterson’s algorithm.

Initially,  $\text{flag1} = \text{flag2} = \text{turn} = 0$

Thread 1	Thread 2
$\text{flag1} = 1;$	$\text{flag2} = 1;$
$\text{turn} = 2;$	$\text{turn} = 1;$
$\text{r1} = \text{turn};$	$\text{r3} = \text{turn};$
$\text{r2} = \text{flag2};$	$\text{r4} = \text{flag1};$

Result:  $\text{r1} = 2, \text{r3} = 1, \text{and } \text{r2} = \text{r4} = 0$

```

Startstate Begin
  If (MyTest = TEST_Peterson) Then
    addDefault(Flag1, ZERO);
    addDefault(Flag2, ZERO);
    addDefault(Turn, ZERO);

    add(T1, INS_W, Flag1, ONE);
    add(T1, INS_W, Turn, TWO);
    add(T1, INS_R, Turn, r1);
    add(T1, INS_R, Flag2, r2);

    add(T2, INS_W, Flag2, ONE);
    add(T2, INS_W, Turn, ONE);
    add(T2, INS_R, Turn, r3);
    add(T2, INS_R, Flag1, r4);
  End;
End;

```

**Figure 2.5.** Encoding the test program in Table 2.3 using Murphi.

possible results. A configuration flag `SHOW_ALL_RESULTS` is used to select the output mode.

If the program in Table 2.3 is executed under PRAM or coherence, the tool immediately detects certain thread interleaving that would lead to the result of interest, indicating that Peterson’s algorithm is broken under these memory models. Under sequential consistency, however, the execution in Table 2.3 is not allowed. If the test program in Table 2.3 is executed using normal variables under  $JMM_{MP}$ , it is immediately detected that such a scenario is allowed, proving that the algorithm is unsafe under  $JMM_{MP}$ . On the other hand, if volatile variables are used, such an execution would not be allowed. Carefully analyzing concurrent algorithms based on formal methods is an effective strategy for developing robust multithreaded programs.

Running on a PC with a 900 MHz Pentium III processor and 256 MB of RAM, most of the test programs terminate in less than one second.

```

/*****
 * Check an invariant.
 *****/
Invariant "Invariant Test"
  (!SHOW_ALL_RESULTS & AllDone())
  ->
  (MyTest = TEST_Peterson ->
   !(Thread[T1].LV[r1].data = TWO & Thread[T1].LV[r2].data = ZERO &
    Thread[T2].LV[r3].data = ONE & Thread[T2].LV[r4].data = ZERO));

/*****
 * Output all results.
 *****/
Rule "Output final result"
  SHOW_ALL_RESULTS & AllDone()
==> Begin
  If (MyTest = TEST_Peterson) Then
    Put " r1:"; Put Thread[T1].LV[r1].data;
    Put " r2:"; Put Thread[T1].LV[r2].data;
    Put " r3:"; Put Thread[T2].LV[r3].data;
    Put " r4:"; Put Thread[T2].LV[r4].data;
    Put "\n";
  End;
End;

```

**Figure 2.6.** Two ways to check a test program in Murphi.

## 2.7 Summary

UMM provides a uniform framework for specifying memory models and offers model checking support. As a specification and verification framework, UMM possesses many favorable characteristics:

1. The specification style based on guarded commands enables the integration of model checking techniques, providing strong support for verification. Formal methods can help one to better understand the subtleties of the model by detecting corner cases that would be very hard to find through traditional simulation techniques. In addition, the mathematical rules in the transition table makes the specification more rigorous.
2. The flexible design of UMM enables easy configuration. The abstraction mech-

anism in UMM offers a feasible common design interface for executable memory models. Different bypassing rules and visibility ordering rules can be carefully developed so that a user can select from a “menu” of primitive memory properties to assemble a desired formal specification.

3. The simple conceptual architecture of UMM eliminates unnecessary complexities introduced by implementation specific data structures. Hence, it helps clarify the essential semantics of the memory system.

Designed as a specification framework, however, UMM is not intended to be actually implemented in a real system since it chooses simplicity over efficiency in its architecture. For example, UMM does not support some of the *nonblocking* memory operations that could be expressed using intermediate data structures and fine-grained memory activities. Also, based on model checking techniques, UMM is exposed to the state explosion problem. Effective abstraction and slicing techniques are needed to verify larger programs.

# CHAPTER 3

## A NONOPERATIONAL SPECIFICATION APPROACH

### 3.1 Chapter overview

This chapter presents a nonoperational memory model specification framework called Nemos (Nonoperational yet Executable Memory Ordering Specifications). Five classical models, including sequential consistency, coherence, PRAM, causal consistency, and processor consistency, are defined in Nemos following a modular specification style. This chapter also describes how to formalize the memory ordering rules for the Intel Itanium processor family. The formal Itanium memory model is given in Appendix B using predicate logic and in Appendix C using the HOL notation. This chapter focuses on the specification aspects of Nemos. Chapter 4 will describe how to execute the specifications for formal verification.

### 3.2 Nemos overview

Because operational semantics relies on a machine to define a shared memory system and does not provide direct intuitions regarding the desired properties, an operational approach leads to two problems: first, it often makes the underlying consistency requirements unclear; second, it does not offer sufficient flexibility to capture complex ordering rules. A nonoperational specification directly defines the “intrinsic” memory ordering rules that makes a memory model easier to understand, but traditional nonoperational approaches do not support computer-aided analysis. The Nemos framework is designed to solve these issues by making a memory model specification both *declarative* and *executable*. Particular emphasis is placed in Nemos on proving system correctness against high level requirements.

A shared memory system imposes various ordering restrictions among the operations in an execution. The key objective for a memory model specification, therefore,

is to precisely capture these memory ordering requirements. In Nemos, *predicate logic* is used to specify the ordering constraints imposed on an ordering relation *order*. This approach mirrors the style adopted in modern declarative specifications from industry (e.g., [7]). To make the specifications compositional and executable, the notation of Nemos (inspired by unpublished papers from Gordon [39, 40]) differs from previous formalisms in two major ways:

1. The constraints in Nemos are defined in higher order logic, i.e. *order* can be used as a parameter in a constraint definition, so that new refinements to the ordering requirement can be repeatedly added. This allows one to construct a complex model using simpler components.
2. The specifications are fully explicit about all ordering properties, including previously implicit requirements such as totality, transitivity, and circuit-freedom. Without explicating such “hidden” requirements, a specification is not complete for execution.

The following sections illustrate these specification techniques by formalizing a collection of memory models.

### 3.3 Formalizing classical memory models

#### 3.3.1 Terminology

##### 3.3.1.1 Execution

An *execution*, also known as an *execution trace*, consists of a set of operations generated by program instructions, including the initial writes for every variable. An execution is *legal* if there exists an order among the operations that satisfies all memory model constraints.

##### 3.3.1.2 Operation

An *operation*  $i$  is represented by a tuple. For brevity, a common data structure is used to represent all operations, even though some of the fields are only used by certain operations. The tuple representation is as follows:

$\langle Proc, Pc, Op, Var, Data, Source, Id \rangle$ , where

<b>p</b> $i = Proc$ :	issuing processor ( $Proc \in P \cup \{p_{init}\}$ );
<b>pc</b> $i = Pc$ :	program counter;
<b>op</b> $i = Op$ :	instruction type ( $Op \in \{Read, Write\}$ );
<b>var</b> $i = Var$ :	shared variable ( $Var \in V$ );
<b>data</b> $i = Data$ :	data value;
<b>source</b> $i = Source$ :	source for a read, by the ID of the fulfilling write;
<b>id</b> $i = Id$ :	global ID of the operation.

### 3.3.1.3 Initial write

For each variable  $v$ , there is an *initial write* issued by a special processor  $p_{init}$  with the default value of  $v$ .

## 3.3.2 A library of memory consistency properties

Since Nemos characterizes a memory model via a set of ordering rules, it naturally supports a modular specification style, in which common axioms can be easily shared and reused. Hence, it is possible to develop a “library” of memory consistency properties. This section defines several common consistency requirements.

### 3.3.2.1 General ordering rules

General ordering rules are usually implicitly required in previous models. These key requirements must be mathematically defined to make a specification machine-readable. Constraint `requireWeakTotalOrder` takes *order* as a parameter and asserts that every two different operations must be ordered in some way. Constraint `requireTransitiveOrder` specifies that an order can be transitively induced between operations  $i$  and  $k$  through a third operation  $j$ . Constraint `requireAsymmetricOrder`, together with `requireTransitiveOrder`, captures the notion of circuit-freedom.

**requireWeakTotalOrder**  $ops$   $order \equiv \forall i, j \in ops.$   
 $\mathbf{id} \ i \neq \mathbf{id} \ j \Rightarrow (\mathbf{order} \ i \ j \ \vee \ \mathbf{order} \ j \ i)$

**requireTransitiveOrder**  $ops$   $order \equiv \forall i, j, k \in ops.$   
 $(\mathbf{order} \ i \ j \ \wedge \ \mathbf{order} \ j \ k) \Rightarrow \mathbf{order} \ i \ k$

**requireAsymmetricOrder**  $ops$   $order \equiv \forall i, j \in ops.$   
 $\mathbf{order} \ i \ j \Rightarrow \neg(\mathbf{order} \ j \ i)$

### 3.3.2.2 Read value rule

Memory consistency is essentially defined by observable read values. This is generically captured by `requireReadValue`. Intuitively, it requires that the value observed by a read must be provided by the latest write on the same variable. That is, for every read  $k$ , (i) there should exist a candidate write  $i$ , and (ii) there does not exist an overwriting write  $j$ . The constraints imposed on *order* precisely defines how the “latest” write should be determined.

$$\begin{aligned} \text{requireReadValue } ops \text{ order} &\equiv \forall k \in ops. \text{ op } k = \text{Read} \Rightarrow \\ &(\exists i \in ops. \text{ op } i = \text{Write} \wedge \text{ var } i = \text{var } k \wedge \\ &\text{ data } k = \text{data } i \wedge \text{ source } k = \text{id } i \wedge \neg(\text{order } k \ i) \wedge \\ &(\forall j \in ops. \neg(\text{op } j = \text{Write} \wedge \text{ var } j = \text{var } k \wedge \text{order } i \ j \wedge \text{order } j \ k))) \end{aligned}$$

### 3.3.2.3 Serialization

The notion of *serialization*, a common requirement in memory model definitions, is formalized in predicate `requireSerialization`. It requires a circuit-free weak total order among a set of memory operations such that the *read value rule* is also respected.

$$\begin{aligned} \text{requireSerialization } ops \text{ order} &\equiv \\ &\text{requireWeakTotalOrder } ops \text{ order} \wedge \\ &\text{requireTransitiveOrder } ops \text{ order} \wedge \\ &\text{requireAsymmetricOrder } ops \text{ order} \wedge \\ &\text{requireReadValue } ops \text{ order} \end{aligned}$$

### 3.3.2.4 Ordering relations

Ordering relations among memory operations can be induced under certain conditions. Predicate `requireProgramOrder` defines the condition of program order. Predicate `requireWriteIntoOrder` establishes an order between a write and a read according to data flow, which is needed to define causal consistency.

$$\begin{aligned} \text{requireProgramOrder } ops \text{ order} &\equiv \forall i, j \in ops. \\ &((\mathbf{p} \ i = \mathbf{p} \ j \wedge \mathbf{pc} \ i < \mathbf{pc} \ j) \vee (\mathbf{p} \ i = p_{init} \wedge \mathbf{p} \ j \neq p_{init})) \Rightarrow \text{order } i \ j \end{aligned}$$

$$\begin{aligned} \text{requireWriteIntoOrder } ops \text{ order} &\equiv \forall i, j \in ops. \\ &(\text{op } i = \text{Write} \wedge \text{op } j = \text{Read} \wedge \text{data } j = \text{data } i \wedge \text{source } j = \text{id } i) \Rightarrow \text{order } i \ j \end{aligned}$$

### 3.3.2.5 Auxiliary predicates

Sometimes serialization only needs to be enforced on a subset of an execution. Several predicates are provided for filtering operations based on various conditions. In addition, ordering constraints can be separately applied to different executions. Predicate `mapConstraints` is defined to ensure that these separate sets of constraints are consistent with each other. This technique is further demonstrated in the definition of processor consistency. In `mapConstraints`, `order1` and `order2` are the respective ordering relations among `ops1` and `ops2`, with `ops2` being a subset of `ops1`.

**restrictVar**  $ops\ v \equiv \{i \in ops \mid \mathbf{var}\ i = v\}$

**restrictVarWr**  $ops\ v \equiv \{i \in ops \mid \mathbf{var}\ i = v \wedge \mathbf{op}\ i = Write\}$

**restrictProc**  $ops\ proc \equiv \{i \in ops \mid \mathbf{p}\ i = proc \vee (\mathbf{p}\ i \neq proc \wedge \mathbf{op}\ i \neq Read)\}$

**mapConstraints**  $ops1\ order1\ ops2\ order2 \equiv \forall i, j \in ops1.$   
 $(i \in ops2 \wedge j \in ops2) \Rightarrow (\mathbf{order1}\ i\ j = \mathbf{order2}\ i\ j)$

## 3.3.3 Five classical memory models

Five classical memory models are formalized here based on the primitive ordering properties.

### 3.3.3.1 Sequential consistency.

Sequential consistency requires a common total order among all operations, in which program order is also respected.

**legal**  $ops \equiv \exists order.$   
 $\mathbf{requireProgramOrder}\ ops\ order \wedge$   
 $\mathbf{requireSerialization}\ ops\ order$

### 3.3.3.2 Coherence

For each variable, coherence requires a serialization among all memory operations involving that variable.

**legal**  $ops \equiv \forall v \in V. (\exists order.$   
 $\mathbf{requireProgramOrder}\ (\mathbf{restrictVar}\ ops\ v)\ order \wedge$   
 $\mathbf{requireSerialization}\ (\mathbf{restrictVar}\ ops\ v)\ order)$

### 3.3.3.3 PRAM

PRAM requires that for each observing processor  $p$ , there must exist an individual serialization among all memory operations of  $p$  and all writes from other processors.

**legal**  $ops \equiv \forall proc \in P. (\exists order.$   
**requireProgramOrder** (**restrictProc**  $ops\ proc$ )  $order \wedge$   
**requireSerialization** (**restrictProc**  $ops\ proc$ )  $order)$

### 3.3.3.4 Causal consistency

In causal consistency, two operations are ordered if (i) they follow program order, (ii) one operation observes the value provided by the other operation, or (iii) they are transitively ordered via a third operation. After these orders are established, serialization is formed on a per-processor basis similar to PRAM.

**legal**  $ops \equiv \forall proc \in P. (\exists order.$   
**requireProgramOrder**  $ops\ order \wedge$   
**requireWriteIntoOrder**  $ops\ order \wedge$   
**requireTransitiveOrder**  $ops\ order \wedge$   
**requireReadValue** (**restrictProc**  $ops\ proc$ )  $order \wedge$   
**requireWeakTotalOrder** (**restrictProc**  $ops\ proc$ )  $order \wedge$   
**requireAsymmetricOrder** (**restrictProc**  $ops\ proc$ )  $order)$

### 3.3.3.5 Processor consistency

We formalize Goodman's processor consistency based on the interpretation from [11]. As in PRAM, each processor must observe an individual serialization (as captured by  $order2$ ). Similar to coherence, all writes to the same variable must exhibit the same order across all these serializations (as captured by the total order imposed on  $order1$ ). In addition, these requirements must be satisfied at the same time in a mutually consistent manner. This critical requirement, which may be easily overlooked, is clearly spelled out by `mapConstraints`.

**legal**  $ops \equiv \exists order1.$   
 $(\forall v \in V. \mathbf{requireWeakTotalOrder} (\mathbf{restrictVarWr} \mathit{ops} \mathit{v}) \mathit{order1}) \wedge$   
 $(\forall proc \in P. \exists order2.$   
**requireProgramOrder** (**restrictProc**  $ops\ proc$ )  $order2 \wedge$   
**requireSerialization** (**restrictProc**  $ops\ proc$ )  $order2 \wedge$   
**mapConstraints**  $ops\ order1$  (**restrictProc**  $ops\ proc$ )  $order2)$

## 3.4 Formalizing the Intel Itanium memory model

In this section, the Itanium memory ordering rules outlined in the Intel application note [7] is adapted using Nemos. Virtually the entire Intel application note, except for the rules dealing with semaphore operations, has been captured. We assume proper address alignment and common address size for all memory accesses, which would be the common case encountered by programmers. The detailed definition of the Itanium memory model is presented in Appendix B. This section explains each of the rules.

### 3.4.1 Terminology

Some of the previous definitions need to be extended for the Itanium memory model.

#### 3.4.1.1 Instructions

Only instructions with memory access or memory ordering semantics are considered here. Five instruction types are defined for the Itanium architecture: load-acquire (`ld.acq`), store-release (`st.rel`), unordered load (`ld`), unordered store (`st`), and memory fence (`mf`). An instruction  $i$  may have *read semantics* (`isRd i = true`) or *write semantics* (`isWr i = true`). `Ld.acq` and `ld` have read semantics. `St.rel` and `st` have write semantics. `Mf` has neither read nor write semantics. Instructions are decomposed into *operations* to allow a fine-grained specification of the ordering properties.

#### 3.4.1.2 Operation tuple

To describe memory operations allowed by the Itanium architecture, the operation tuple is extended as  $\langle P, Pc, Op, Var, Data, WrId, WrType, WrProc, Reg, UseReg, Id \rangle$ , where

<b>p</b> $i = P$ :	issuing processor;
<b>pc</b> $i = Pc$ :	program counter;
<b>op</b> $i = Op$ :	instruction type;
<b>var</b> $i = Var$ :	shared memory location;
<b>data</b> $i = Data$ :	data value;
<b>wrID</b> $i = WrId$ :	identifier of a write operation;
<b>wrType</b> $i = WrType$ :	type of a write operation;
<b>wrProc</b> $i = WrProc$ :	target processor observing a write operation;
<b>reg</b> $i = Reg$ :	register;
<b>useReg</b> $i = UesReg$ :	flag of a write indicating if it uses a register;
<b>id</b> $i = Id$ :	global identifier of the operation.

A read instruction or a fence instruction is transformed to a single operation. A write instruction is decomposed into multiple operations, comprising a local write operation (**wrType**  $i = Local$ ) and a set of remote write operations (**wrType**  $i = Remote$ ) for each target processor (**wrProc**  $i$ ), which also includes the issuing processor. Every write operation  $i$  that originates from a single write instruction shares the same program counter (**pc**  $i$ ) and write ID (**WrID**  $i$ ).

Following the strategy of [7], we use a different treatment for default write values. Instead of including default write operations in the execution, the Itanium memory model applies a predicate **default** to get the default value of a variable in the read value rule.

### 3.4.1.3 Address attributes

Every memory location is associated with an address attribute, which can be write-back (WB), uncacheable (UC), or write-coalescing (WC). Memory ordering semantics may vary for different attributes. Predicate **attribute** is used to find the attribute of a location.

## 3.4.2 The Itanium memory ordering rules

As shown below, predicate **legal** is a top-level constraint that defines the legality of a trace  $ops$  by checking the existence of an  $order$  among  $ops$  that satisfies all requirements. Each requirement is formally defined in Appendix B.

**legal**  $ops \equiv \exists order.$   
**requireLinearOrder**  $ops order \wedge$   
**requireWriteOperationOrder**  $ops order \wedge$   
**requirePO**  $ops order \wedge$

**requireMemoryDataDependence** *ops order*  $\wedge$   
**requireDataFlowDependence** *ops order*  $\wedge$   
**requireCoherence** *ops order*  $\wedge$   
**requireReadValue** *ops order*  $\wedge$   
**requireAtomicWBRelease** *ops order*  $\wedge$   
**requireSequentialUC** *ops order*  $\wedge$   
**requireNoUCBypass** *ops order*

Table 3.1 illustrates the hierarchy of the Itanium memory model definition. Most constraints strictly follow the rules from [7]. We also explicitly add a predicate **requireLinearOrder** to capture the general ordering requirement since [7] has only English to convey this important ordering property.

### 3.4.2.1 General ordering requirement (Appendix B.1)

The general ordering requirement is specified in **requireLinearOrder**. It requires **order** to be a weak total order which is also transitive and asymmetric. It should be noted that even though the term “linear order” is inherited from [7], our definition is slightly different from the standard meaning of linear order, which would require the order to be *reflexive*, *transitive*, and *antisymmetric* (i.e., **order**  $i j \wedge$  **order**  $j i \Rightarrow i = j$ ).

**Table 3.1.** The specification hierarchy of the Itanium memory ordering rules.

<i>requireLinearOrder</i> - requireWeakTotal - requireTransitive - requireAsymmetric	<i>requireMemoryDataDependence</i> - MD:RAW - MD:WAR - MD:WAW	<i>requireReadValue</i> - validWr - validLocalWr - validRemoteWr - validDefaultWr - validRd
<i>requireWriteOperationOrder</i> - local/remote case - remote/remote case	<i>requireDataFlowDependence</i> - DF:RAR - DF:RAW - DF:WAR	<i>requireNoUCBypass</i>
<i>requirePO</i> - acquire case - release case - fence case	<i>requireCoherence</i> - local/local case - remote/remote case	<i>requireSequentialUC</i> - RAR case - RAW case - WAR case - WAW case
	<i>requireAtomicWBRelease</i>	

### 3.4.2.2 Write operation order (Appendix B.2)

The write operations that originate from a single write instruction is constrained by `requireWriteOperationOrder`. This rule guarantees that no write can become visible remotely before it becomes visible locally. Suppose  $i$  and  $j$  are two write operations from the same write instruction, `orderedByWriteOperation` outlines two situations when  $i$  and  $j$  should be ordered by *write operation order*: (1)  $i$  is local,  $j$  is remote and  $j$  targets the local processor; (2) both  $i$  and  $j$  are remote,  $i$  targets the local processor, and  $j$  targets a remote processor.

### 3.4.2.3 Program order (Appendix B.3)

Constraint `requireProgramOrder` in Appendix B.3 restricts reordering among instructions of the same processor with respect to the program order (notice that this is different from the conventional definition of program order given in Section 3.3.2). It follows the convention of [7] — rules that depend on memory locations are defined separately as *memory-data dependence*, thus excluded from this constraint. Program order rules fall into three categories according to different ordering semantics of `load-acquire`, `store-release`, and `memory fence`:

1. No operation can become visible before a preceding `load-acquire`.
2. No `store-release` can become visible before a preceding operation.
3. Operations become visible in-order with respect to `memory fence`.

### 3.4.2.4 Memory-data dependence (Appendix B.4)

This rule restricts reordering among instructions from the same processor when they access *common locations*. When write is involved, only local write operation is restricted by this rule.

### 3.4.2.5 Data-flow dependence (Appendix B.5)

This rule is intended to specify how local *data dependence* and *control dependence* should be treated. However, this is an area that is not fully specified in [7]. For instance, [7] does not define when memory operations should be ordered by data-flow and what data should be carried by a write operation when it uses a local register. Instead of pointing to an informal document as done in [7], we provide a formal

specification covering most cases of data dependence, namely, our rule establishes data dependence between two memory operations by checking the conflict uses of local registers.

This specification does not cover branch instructions or indirect-mode instructions that also induce data dependence. We provide enough data dependence specification to let designers experiment with straight-line code that uses registers — this is an important requirement to support execution. In Chapter 5, we will discuss how to specify control dependence using constraints.

Although [7] outlines four categories for data-flow dependence (RAR, RAW, WAR, and WAW), the WAW case (a *write* here is actually a *read* in terms of register usage) does not establish any value-based data dependence relation. Therefore, data dependence as specified in `orderedByLocalDepence` is only set up by the first three cases.

#### 3.4.2.6 Coherence (Appendix B.6)

This rule constrains the order of writes to a *common location*. If two writes to the same location with the attribute of WB or UC become visible to a processor in some order, they must become visible to all processors in that order.

#### 3.4.2.7 Read value (Appendix B.7)

This rule defines what value can be observed by a read operation. There are three scenarios: a read can get the data from a local write (`validLocalWr`), a remote write (`validRemoteWr`), or the default value (`validDefaultWr`).

Similar to shared memory read value rules, predicate `validRd` guarantees consistent assignments of registers — the value of a register is obtained from the most recent previous assignment of the same register.

#### 3.4.2.8 Total order of WB releases (Appendix B.8)

This specifies that store-releases to **write-back** (WB) memory must obey *remote write atomicity*, i.e., they become remotely visible atomically. To capture the semantics of remote write atomicity is one of the main reasons for dividing a write into multiple operations in the Itanium model.

### 3.4.2.9 Sequentiality of UC operations (Appendix B.9)

This specifies that operations to `uncacheable(UC)` memory locations must have the property of *sequentiality*, i.e., they must become visible in program order.

### 3.4.2.10 No UC bypassing (Appendix B.10)

This specifies that `uncacheable(UC)` memory does not allow local bypassing from UC writes.

## 3.5 Comparison between operational and nonoperational approaches

An axiomatic approach such as Nemos divides the global ordering relation in terms of *facets*, each of which constrains a specific aspect of the memory system. The UMM framework applies a two-layer abstract machine to generate operational memory model definitions. Variations of memory consistency properties are parameterized as different bypassing rules and visibility ordering rules. The total order property, if required, can be implicitly built up during the execution based on interleavings allowed by the bypassing rules.

Despite its operational style, the UMM framework is closely related to axiomatic specification methods. If a memory model allows *all* instructions to be sent to GIB in any arbitrary order and then impose additional ordering constraints when read values are obtained, a UMM specification degenerates to a nonoperational one.

Each of the two styles has its own advantages. The operational style can often simplify the specification using its interleaving mechanism. But it is not easy to perform correctness proof against high level properties. The nonoperational approach, on the other hand, is declarative and more flexible. One can disable/enable the constraints and study the impact on the global ordering requirement. However, each constraint itself may involve aspects that pertain to both program order and visibility order, which cannot be easily distinguished.

Understanding the different specification mechanisms can help one to transform a memory model definition from one style to the other. To capture an axiomatic definition using UMM, one needs to consider all the ordering rules and extract those that can be enforced using the front-end instruction selection process of the UMM

framework — this would simplify the filtering process when read values are obtained. To convert a UMM specification to an axiomatic definition, one must encode all the ordering requirements implied by the UMM front-end process and impose them as axioms on the final execution trace.

### 3.6 Summary

The expressive power offered by higher order predicate logic allows the Nemos framework to capture complex ordering rules using the same notation, making comparative analysis an easier task. Even though this chapter is by no means an effort to comprehensively cover all existing proposals, other models can be adapted using Nemos with moderate efforts.

The compositional specification style makes it possible to develop reusable definitions for memory consistency properties. One can even imagine having a memory model API (Application Programming Interface), which can be called by a user for selectively assembling different executable models. The modular approach also makes the Nemos framework scalable, a requirement for defining complex industrial models. Being able to tackle cutting-edge commercial architectures attests to the scalability of this framework.

The memory models specified in predicate logic can be easily coded using the HOL theorem prover [41]. Such rigorous specifications will allow one to prove generic shared memory properties using theorem proving. The Itanium memory ordering rules are provided in the HOL notation in Appendix C.

# CHAPTER 4

## SOLVING MEMORY ORDERING CONSTRAINTS

### 4.1 Chapter overview

This chapter presents a method that converts a memory model analysis problem to an equivalent constraint satisfaction problem and automates the analysis using constraint solving. This approach makes nonoperational memory ordering rules executable, a key feature that has been lacking in previous nonoperational approaches. Two solving methods are discussed: one applies a constraint solver from FD-Prolog<sup>2</sup> and the other exploits a boolean SAT solver.

### 4.2 Introduction

As shown in previous chapters, litmus tests, albeit small, can reveal critical memory model properties to help a user make right decisions in code selection and optimization. Consider the litmus test shown in Table 4.1, which illustrates an important property of processor consistency. The outcome of interest in Table 4.1 should be prohibited by processor consistency even though it is allowed by both coherence and PRAM. This result might come as a surprise to many. After all, isn't processor consistency intended to be a combination of coherence and PRAM? With a careful study, it can be proven that the two operations  $c = 0$  and  $c = 2$  cannot be ordered in a *consistent* way when explaining PRAM and coherence at the same time. Such analysis is very helpful for promoting understanding, but a pencil-and-paper reasoning approach can quickly become infeasible and error-prone when bigger tests are used or more intricate rules are involved. For example, how can one be assured that there does not exist an interacting memory ordering rule which might introduce

---

<sup>2</sup>FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.

**Table 4.1.** An execution allowed by coherence and PRAM but prohibited by PC.Initially,  $a = b = c = 0$ 

P1	P2
$a = 1;$	$b = 1;$
$c = 0;$	$c = 2;$
$r1 = b;$	$r2 = a;$
Result: $r1 = r2 = 0$	

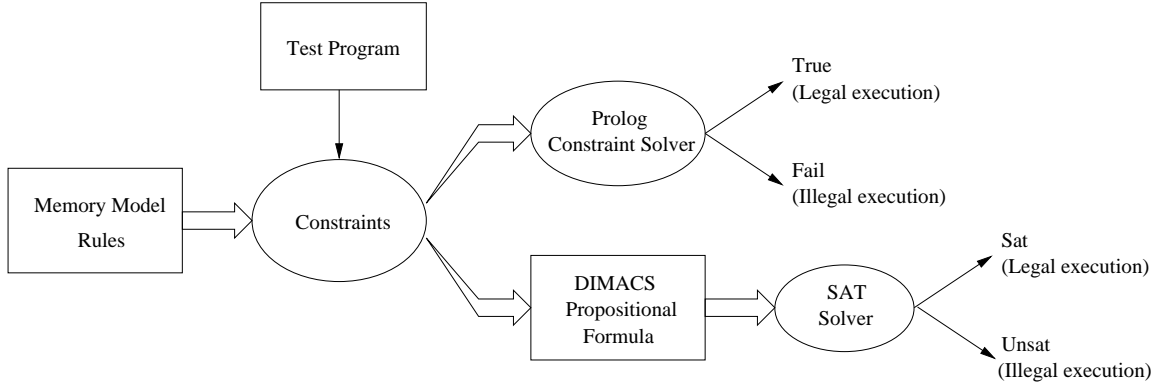
unexpected implications? Therefore, without an automated approach, the usefulness of such analysis is significantly reduced. Also, a large scale design is often composed of simpler components. To avoid being overwhelmed by the overall complexity, a useful technique is to isolate the rules related to specific memory model features so that the model can be analyzed piece by piece.

These issues suggest that a series of useful features is needed from the specification framework to help one better understand the underlying model. Unfortunately, previous rule-based specification methods leave these issues unresolved because they use notations that do not support analysis through execution. Given that designers and programmers need lucid and reliable memory model specifications, and given that memory model specifications can live for decades, it is crucial that progress be made in this regard.

### 4.3 Overview of the solving method

Chapter 3 has shown how to capture memory ordering constraints using higher order predicate calculus. In order to make such a nonoperational specification executable, these ordering constraints need to be instantiated to a finite program execution. This process converts the system requirements from higher order logic back to propositional logic. Existing solving techniques can be subsequently applied to search the existence of an ordering relation that satisfies all constraints.

The processing flow of the methodology is shown in Figure 4.1, which comprises the following steps: (i) capturing the memory consistency model as constraints, (ii) applying these constraints to a given test program and reducing the verification



**Figure 4.1.** The process of making an axiomatic memory model executable.

problem to a constraint satisfaction problem, and (iii) employing a suitable tool to solve the constraint problem automatically.

### 4.3.1 The algorithm

Given a test program  $\mathcal{P}$ , one can derive its execution  $ops$  from the program text in a preprocessing phase. The initial execution is *symbolic*, that is,  $ops$  may contain free variables, e.g., for data values and ordering relations. Suppose  $ops$  has  $n$  operations, there are  $n^2$  ordering pairs among these operations. We construct a  $n \times n$  adjacency matrix  $\mathcal{M}$ , where the element  $\mathcal{M}_{ij}$  indicates whether operations  $i$  and  $j$  should be ordered. We then go through each requirement in the specification and impose the corresponding propositional constraints with respect to the elements of  $\mathcal{M}$ . The goal is to find a binding of the free variables in  $ops$  such that it satisfies the conjunction of all requirements or to conclude that no such binding exists. This is automated using a constraint solver.

### 4.3.2 Applying constraint logic programming (CLP)

Logic programming differs fundamentally from conventional programming in that it describes the logical structure of the problems rather than prescribing the detailed steps of solving them. This naturally reflects the philosophy of the nonoperational specification style. As a result, our formal specifications can be easily encoded using Prolog, a popular logic programming language. Memory ordering constraints can be solved through a conjunction of two mechanisms that FD-Prolog readily provides.

One applies backtracking search for all constraints expressed by logical variables, and the other uses non-backtracking constraint solving based on *arc consistency* [46] for finite domain variables, which is potentially more efficient and certainly more complete (especially under the presence of negation) than with logical variables. This works by adding constraints in a monotonically increasing manner to a constraint store, with the built-in constraint propagation rules of FD-Prolog helping refine the variable ranges (or concluding that the constraints are not satisfiable) when constraints are asserted to the constraint store.

In a sense, the built-in constraint solver from FD-Prolog provides an effective means for bounded software model checking by explicitly exploring all program executions, but symbolically reasoning about the constraints imposed to free variables.

### 4.3.3 Applying boolean satisfiability techniques

The goal of a boolean satisfiability problem is to determine a satisfying variable assignment for a boolean formula or to conclude that no such assignment exists. Converting the constraint satisfaction problem to a boolean SAT problem allows one to benefit from SAT solving techniques. With the tremendous improvement in SAT solving, this approach offers a promising direction for enhancing scalability.

## 4.4 The CLP approach

A tool named *NemosFinder* is developed in SICStus Prolog [5] to enable memory model analysis. *NemosFinder* is written in a modular fashion and is highly configurable. Memory models are defined as sets of predicates, and litmus tests are contained in a separate test file. When a memory model is chosen and a test number is selected, the FD constraint solver attempts all possible orders until it can find an instantiation that satisfies all constraints.

Translating a formal specification to Prolog is relatively straightforward. One caveat, however, is that most Prolog systems do not directly support quantifiers. Existential quantification can be realized via Prolog's backtracking mechanism, but universal quantification needs to be implemented by enumerating the related finite domain. As a concrete example, recall constraint `requireWeakTotalOrder`, which was originally defined as follows:

**requireWeakTotalOrder** *ops order*  $\equiv \forall i, j \in ops.$   
 $\text{id } i \neq \text{id } j \Rightarrow (\text{order } i j \vee \text{order } j i)$

This rule can be encoded using the following Prolog predicates:

```
requireWeakTotalOrder(Ops,Order):-
    length(Ops,N),
    forEachElement(Order,N,doWeakTotalOrder).

elementProgram(doWeakTotalOrder,Order,N,I,J):-
    matrix_elem(Order,N,I,J,Oij),
    matrix_elem(Order,N,J,I,Oji),
    (I #\= J #=> Oij #\= Oji).
```

Notice that in Prolog, variable names start with a capital letter. Predicate `forEachElement` is recursively defined to call `elementProgram` for every element in the adjacency matrix *Order*:

```
forEachElement(Order,N,P):-
    T is N*N,
    forEachElementHelper(0,T,Order,N,P),
    !.

forEachElementHelper(T,T,_,_,_).

forEachElementHelper(K,T,Order,N,P):-
    K1 is K+1,
    I is (K // N)+1,
    J is (K rem N)+1,
    elementProgram(P,Order,N,I,J),
    forEachElementHelper(K1,T,Order,N,P).
```

Barring some implementation details, one technique shown by the above example is worth noting. That is, the adjacency matrix *Order* is passed in as a finite domain (FD) variable.<sup>3</sup> The domain of the elements in *Order* (which is *boolean* in this case) is previously set up in the top level predicate. Providing such domain information significantly reduces the solving time, and hence is crucial for system performance. The search order among the constraints may also impact performance. In general, it is advantageous to let the solver satisfy the most restrictive goal first.

---

<sup>3</sup>Propositional operators for FD variables start with the # sign.

## 4.5 Applying Nemos for verification

This section uses some examples to demonstrate how to apply NemosFinder for memory model analysis.

### 4.5.1 Analyzing classical memory models

Recall the test program in Table 4.1. This test can serve as an example for the analysis of classical memory models. The execution of the test is displayed in Table 4.2. When running under coherence, NemosFinder quickly concludes that the execution is legal, with an output displaying possible adjacency matrices and interleavings shown in Figure 4.2. A value of 1 for element  $\mathcal{M}_{ij}$  in the matrix indicates that the two operations  $i$  and  $j$  are ordered. The result of interest is also legal for PRAM, which is illustrated in Figure 4.3.

If processor consistency is selected, NemosFinder answers that the execution is illegal, indicating that there does not exist an order that can satisfy coherence and PRAM at the same time. The user can play with a memory model and ask “what if” queries by selectively enabling/disabling certain ordering rules. For example, if the user comments out the `mapConstraints` requirement in processor consistency and runs the test again, the result would become legal. This *incremental* and *interactive* test environment can help one to study a model piece by piece and identify the “root cause” of a certain program behavior.

### 4.5.2 Analyzing the Itanium memory model

To illustrate the analysis of the Itanium memory ordering rules, consider Program  $b$  discussed earlier in Figure 1.1. Its instructions are decomposed into operations as shown in Table 4.3. After taking this trace as input, the Prolog tool attempts all

**Table 4.2.** The execution of the litmus test in Table 4.1.

Pinit	P1	P2
(1) write(a,0);	(4) write(a,1);	(7) write(b,1);
(2) write(b,0);	(5) write(c,0);	(8) write(c,2);
(3) write(c,0);	(6) read(b,0);	(9) read(a,0);

	1	4	9
1	0	1	1
4	0	0	0
9	0	1	0

	2	6	7
2	0	1	1
6	0	0	1
7	0	0	0

	3	5	8
3	0	1	1
5	0	0	0
8	0	1	0

(a) Interleaving for  $a$ : 1 9 4 (b) Interleaving for  $b$ : 2 6 7 (c) Interleaving for  $c$ : 3 8 5**Figure 4.2.** Adjacency matrices for the execution in Table 4.2 under coherence.

possible orders until it can find an instantiation that satisfies all constraints. For this particular example, it returns “illegal trace” as the result. If one comments out the `requireProgramOrder` rule in the Itanium memory model definition and examines the trace again, the tool quickly finds a legal ordering matrix and a corresponding interleaving as shown in Table 4.4. Many other experiments can be conveniently performed in a similar way. Therefore, not only does Nemos give people the notation to write rigorous as well as readable specifications, it also allows users to play with the model, asking “what if” queries after selectively enabling/disabling the ordering rules that are crucial to their work.

The built-in predicate `setof` provided by Prolog can also be used to collect all possible return values for read operations. This is achieved by repeatedly backtracking and gradually building up a list of the solutions.

	1	2	3	4	5	6	7	8
1	0	0	0	1	1	1	1	1
2	1	0	0	1	1	1	1	1
3	1	1	0	1	1	1	1	1
4	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	1	1
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

	1	2	3	4	5	7	8	9
1	0	0	0	1	1	1	1	1
2	1	0	0	1	1	1	1	1
3	1	1	0	1	1	1	1	1
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
7	0	0	0	1	1	0	1	1
8	0	0	0	1	1	0	0	1
9	0	0	0	1	1	0	0	0

(a) Interleaving for Process 1:  
3 2 1 4 5 6 7 8(b) Interleaving for Process 2:  
3 2 1 7 8 9 4 5**Figure 4.3.** Adjacency matrices for the execution in Table 4.2 under PRAM.



### 4.6.2 Improved approach

To improve the SAT-based solving method, a new prototype tool (see [38] for details) is developed in Ocaml for analyzing the Itanium memory model. Given a problem instance  $\langle r, ops \rangle$ , where  $r$  is a collection of higher order logic formulae defining the memory model and  $ops$  is a finite execution, the tool transforms the problem instance to a functional program  $p$  that when run generates an equivalent boolean formula  $b$ . This formula is subsequently solved by a SAT solver.

Various techniques have been explored to improve the SAT generation time and solving time. For example, incremental SAT solving is applied. This is based on the observation that  $b$  consists of two parts  $b_1$  and  $b_2$ , where  $b_1$  can be pre-generated knowing only the program length. If we pre-generate  $b_1$  for various lengths, we can load them into an incremental SAT solver (we use the Satzoo incremental solver [1]) and save their process state (called checkpoint) on disk; later when given an execution  $ops$ ,  $b_2$  is generated for it, and fed to a saved image for further execution. Different SAT encoding methods have also been studied.

Another experiment done in [38] involves solving memory ordering constraints using a QBF (Quantified Boolean Formula) solver. This approach helps formally define the problem since a QBF solver directly accepts quantifiers, but only those QBF instances derived from very short executions can, at present, be checked using available QBF satisfiability tools.

With the SAT approach, the prototype tool can easily handle 128 memory operations for the Itanium memory model. With more code tuning, given the steadily improving nature of SAT tools, and by calling SAT tools directly instead of suffering the overhead of file I/O, it is optimistic that the tool can handle well above 128 operations in the near future.

## 4.7 Performance results

Gibbons and Korach [35] have shown that the problem of checking executions against sequential consistency is NP-complete. Generalizing this result, Cantin [18] has shown that the problem of checking executions against memory ordering rules that contain coherence as a sub-rule is NP-hard. Despite the complexity involved

in satisfiability problems, constraint-based methods have become very successful in practice, thanks to the efficient solving techniques developed in recent years.

#### 4.7.1 Analysis of classical memory models

Table 4.5 summarizes the results for the test program in Table 4.1. Performance is measured on a Pentium 366 MHz PC with 128 MB of RAM running Windows 2000. SICStus Prolog is run under compiled mode.

#### 4.7.2 Analysis of the Itanium memory model

Table 4.6 illustrates the performance statistics for exercising the Itanium memory model. These tests are chosen from [7] and represented by their original table numbers. For the method using SAT solvers, the initial approach is applied. The clause generation time is noticeably larger than the actual SAT solving time, since the entire formula is encoded at once through symbolic execution using the Prolog code and is recursively simplified afterwards. Results are measured on a Pentium III 900 MHz machine with 256 MB of RAM running Windows 2000. SICStus Prolog is run under compiled mode. The SAT solver used is ZChaff.

#### 4.7.3 The improved SAT approach

Using the improved SAT-based approach, executions with 32, 64, and 128 operations are tested. The instruction mix was heavily skewed towards *stores* to reflect the worst-case behavior (more rules pertain to stores than loads).

Table 4.7 provides the time of generating SAT instances for formula parts  $b_1$  and  $b_2$ . It also gives the SAT solving time for a monolithic run (the solving time for the full instance), and for running parts  $b_1$  and  $b_2$  separately. All tests are measured on

**Table 4.5.** Performance summary for the test program in Table 4.1.

Memory Model	SC	Coherence	PRAM	Causal Consistency	PC
Result	illegal	legal	legal	legal	illegal
Time (sec)	0.18	0.03	0.24	0.39	0.28

**Table 4.6.** Performance summary for exercising the Itanium memory model.

Test	Result	FD Solver (sec)	Vars	Clauses	SAT (sec)	CNF Gen (sec)
[7, Table 5]	illegal	0.49	64	679	0.01	3.67
[7, Table 10]	legal	3.00	100	1280	0.01	8.23
[7, Table 15]	illegal	22.29	576	15706	0.01	211.76
[7, Table 18]	illegal	2.40	144	2125	0.01	15.75
[7, Table 19]	legal	4.81	144	2044	0.01	15.68

**Table 4.7.** Time for generating SAT instances for formula parts  $b_1$  and  $b_2$ .

#operations	SAT Generation (sec)		SAT Solving (sec)		
	Part $b_1$	Part $b_2$	monolithic	Part $b_1$	Part $b_2$
32	0.51	0.10	0.33	0.69	0.05
64	4.31	0.97	2.73	6.17	0.50
128	34.26	9.10	164.80	145.64	351.10

an AMD Athlon XP2100+ CPU running at 1.733 GHz, and with 1GB memory. The OS is Red Hat Linux Version 9.

## 4.8 Summary

This chapter presents a novel constraint-based approach that makes a nonoperational memory model executable. Prototype tools with different solving methods are developed. Since memory ordering rules are isolated as “facets,” one can analyze a model piece by piece. A Prolog-based utility allows a user to interact with the tool and obtain immediate feedback during analysis. A QBF solver can minimize the specification gap with its capability of handling quantifiers. However, our QBF-based prototype does not scale well. The research of QBF solvers is still at a preliminary stage compared to propositional SAT. Our work can help accelerate its development by providing industrially motivated benchmarks. With the SAT-based approach, we are now in a position to perform practical verification in industrial settings.

## CHAPTER 5

# MEMORY-MODEL-SENSITIVE PROGRAM ANALYSIS

### 5.1 Chapter overview

This chapter proposes “memory-model-sensitive” program analysis, a type of analysis for multithreaded programs that precisely covers all execution paths allowed by the underlying threading environment. In comparison with conventional program analysis, this approach does not offer the same kind of performance and scalability, due to the complexity involved in exact formal reasoning. However, we illustrate that a formal semantics can serve more than documentation purposes — it can be applied as a sound basis for *rigorous* property checking, upon which more scalable methods can be derived. To demonstrate, this chapter formalizes sequential consistency for a source language that supports the use of local variables, computation operations, branch statements, and monitors for mutual exclusion. By additionally formulating correctness properties as constraints, program verification can be conducted in a rigorous manner. This approach is applied to automate three important analyses: execution validation, race detection, and atomicity verification.

### 5.2 Introduction

Existing analyses for multithreaded programs rely on simplifying assumptions about the underlying execution platform. They tend to only concentrate on efficiency or scalability. These are highly worthy goals, but there is also a clear need to formally define how memory models affect familiar notions used in program analysis.

#### 5.2.1 Race detection

One common analysis is *race detection*. Consider program *a* in Figure 5.1, where each thread issues a read and a conditional write. Does this program contain data

Initially, $a = b = 0$		Initially, $a = b = 0$	
Thread 1	Thread 2	Thread 1	Thread 2
$r1 = a;$	$r2 = b;$	$r1 = a;$	$r2 = b;$
$\text{if}(r1 > 0)$	$\text{if}(r2 > 0)$	$\text{if}(r1 > 0)$	$\text{if}(r2 >= 0)$
$b = 1;$	$a = 1;$	$b = 1;$	$a = 1;$

(a) Program a                      (b) Program b

**Figure 5.1.** Are these programs race-free?

races? At the first glance, it may appear that the answer is “yes” since it seems to fit the conventional intuition about a race condition — two operations from different threads (e.g.,  $r1 = a$  in thread 1 and  $a = 1$  in thread 2) attempt to access the same variable without explicit synchronization, and at least one of them is a write. However, a more careful analysis reveals that the control flow of the program, which must be consistent with the read values allowed by the memory model, needs to be considered to determine whether certain operations will ever happen. Therefore, before answering the question, one must clarify what memory model is assumed. With sequentially consistent executions, for instance, the branch conditions in program *a* will never be satisfied. Consequently, the writes can never be executed and the code is race-free.

Now consider program *b* in Figure 5.1, where the only difference is that the branch condition in Thread 2 is changed to  $r2 >= 0$ . Albeit subtle, this change would result in data races even under sequential consistency.

### 5.2.2 Execution validation.

In previous chapters, the common goal of many test programs is to determine whether certain read values are permitted by the memory model. This task can be called *execution validation*. Execution validation can help a programmer understand the memory ordering rules and aid in code selection. Similar to a race analysis, data/control flow must be tracked for execution validation. For multithreaded programs, data/control flow is interwoven with shared memory consistency requirements. This makes it extremely hard and error-prone to hand-prove thread behaviors, even

for small programs.

### 5.2.3 Atomicity verification

The *atomicity* requirement is also frequently needed to ensure that certain operations appear to be executed atomically. As pointed out in the literature (e.g., [30]), the absence of race conditions does not guarantee the absence of atomicity violations, and the existence of race conditions does not necessarily introduce atomicity problems. Consider the program in Table 5.1, where thread 1 and 2 respectively implement the deposit and withdraw transactions for a bank account. This program is race-free because all accesses to the global variable *balance* are protected by the same lock. If these two threads are issued concurrently when *balance* is initially 1, *balance* should remain the same after the program completes if implemented properly. Due to the use of local variables, however, one thread can interleave with another while in a “transient” state. Consequently, the final balance can be 0, 1, or 2 depending on the scheduling, which is clearly not what has been intended. Hence, it would be highly desirable to have a systematic approach to specifying and verifying such programmer expectations.

From these examples, several conclusions can be drawn.

1. The precise thread semantics, in addition to the intrathread program semantics, must be taken into account to enable a rigorous analysis of multithreaded

**Table 5.1.** The transactions are not atomic even though the program is race-free.

Thread 1 (Deposit)	Thread 2 (Withdraw)
Lock <i>l1</i> ;	Lock <i>l1</i> ;
<i>r1</i> = <i>balance</i> ;	<i>r3</i> = <i>balance</i> ;
Unlock <i>l1</i> ;	Unlock <i>l1</i> ;
<i>r2</i> = <i>r1</i> + 1;	<i>r4</i> = <i>r3</i> - 1;
Lock <i>l1</i> ;	Lock <i>l1</i> ;
<i>balance</i> = <i>r2</i> ;	<i>balance</i> = <i>r4</i> ;
Unlock <i>l1</i> ;	Unlock <i>l1</i> ;

software, because a property that is satisfied under one memory model can be easily broken under another.

2. Program properties such as race conditions and atomicity requirements need to be formalized because informal intuitions often lead to inaccurate results.
3. An automatic verification tool with exhaustive coverage is extremely valuable for general software development purposes because thread behaviors are often confusing.

Motivated by these observations we have developed a constraint-based analytical framework for reasoning about multithreaded software. The key insight is that in order to support memory-model-sensitive analysis, one needs to put together three pieces of the puzzle: the program property, the interthread memory model semantics, and the intrathread program semantics. Using a verification tool harnessed with a suitable constraint/SAT solver, we can (i) configure the underlying memory model, (ii) select a program property of interest, (iii) take a test program as input, and (iv) verify the result automatically under all executions. Previous analyses are memory-model-insensitive. The approach presented in this chapter fills the gap by providing a mechanism that makes the thread semantics explicit.

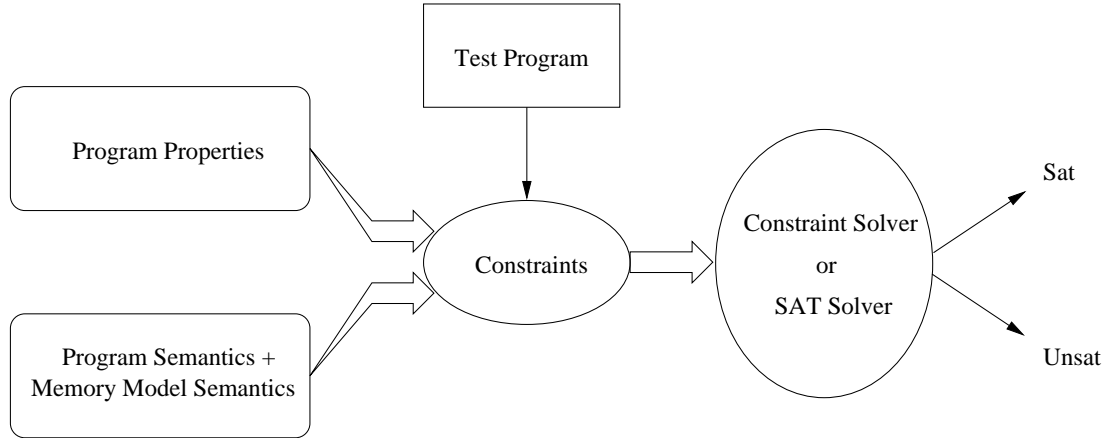
### 5.3 Overview of the methodology

The processing flow of the methodology is shown in Figure 5.2, which comprises the following steps: (i) capturing the semantics of the source language, including both program semantics and thread semantics, as constraints, (ii) formalizing program properties as additional constraints, (iii) applying these constraints to a given test program and reducing the verification problem to a constraint satisfaction problem, and (iv) employing a suitable tool to solve the constraint problem automatically.

### 5.4 The source language

This section develops the formal semantics of sequential consistency for a source language that supports many common programming language constructs.

The choice of using sequential consistency as the basis of our formal development is motivated by two factors: (i) SC is often the implicitly assumed model during software development, i.e., many algorithms and compilation techniques are developed under



**Figure 5.2.** The processing flow for memory-model-sensitive program analysis.

the assumption of SC; (ii) many weak memory models, including the new JMM draft, define pivotal properties such as race-freedom using SC executions. Providing such an executable definition of race conditions will provide a solid foundation upon which a full-featured Java race-detector can be built. Although only SC is formalized here, this framework is generic and allows an arbitrary memory model to be plugged-in for a formal comparative analysis.

Our previous specification of sequential consistency given in Chapter 3 only deals with normal read and write operations. This is sufficient for most processor level memory systems, but it is not enough for describing language level thread activities. In order to handle more realistic programs, it must be extended to support a language that allows the use of local variables, computation operations, control branches, and synchronization operations.

## 5.4.1 Terminology

### 5.4.1.1 Variables

Variables are categorized as *global variables*, *local variables*, *control variables*, and *synchronization variables*. Global and synchronization variables are visible to all threads. Local and control variables are thread local. Control variables do not exist in the original source program — they are introduced by our system as auxiliary variables for control operations. Synchronization variables correspond to the locks

employed for mutual exclusion.

#### 5.4.1.2 Instruction

An *instruction* corresponds to a program statement from the program text. The source language has a syntax similar to Java, with *Lock* and *Unlock* inserted corresponding to the Java keyword *synchronized*. It supports the following instruction types:

Read:	e.g., $r1 = a$
Write:	e.g., $a = 1$ , or $a = r1$
Computation:	e.g., $r1 = r2 + 1$
Control:	e.g., $if(r1 > 0)$
Lock:	e.g., <i>Lock l1</i>
Unlock:	e.g., <i>Unlock l1</i>

#### 5.4.1.3 Execution

An *execution* consists of a set of operation instances generated by program instructions. For example, Table 5.2 displays the execution derived from program b in Figure 5.1 with  $r1 = 1$  and  $r2 = 0$ . We assume that the expression involved in a computation or control operation only uses local variables. If the original instruction performs a computation on global variables, it will be divided into read operations followed by computation operations.

#### 5.4.1.4 Operation tuple

An operation  $i$  is represented by a tuple as follows:

$\langle t, pc, op, var, data, local, localData, cmpExpr, ctrExpr, lock, matchID, id \rangle$ , where

**Table 5.2.** The execution derived from program b in Figure 5.1 with  $r1 = 1$  and  $r2 = 0$ .

Tinit	Thread 1	Thread 2
(1)write(a,0);	(3)read(a,r1,1);	(6)read(b,r2,0);
(2)write(b,0);	(4)control(c1,[r1>0]);	(7)control(c2,[r2>=0]);
	(5)write(b,1,[c1]);	(8)write(a,1,[c2]);

<b>t</b> $i = t$ :	thread ID
<b>pc</b> $i = pc$ :	program counter
<b>op</b> $i = op$ :	operation type
<b>var</b> $i = var$ :	global variable
<b>data</b> $i = data$ :	data value
<b>local</b> $i = local$ :	local variable
<b>localData</b> $i = localData$ :	data value for local variable
<b>cmpExpr</b> $i = cmpExpr$ :	computation expression
<b>ctrExpr</b> $i = ctrExpr$ :	path predicate
<b>lock</b> $i = lock$ :	lock
<b>matchID</b> $i = matchID$ :	ID of the matching lock
<b>id</b> $i = id$ :	global ID of the operation

For every global variable  $a$ , there is a default write operation for  $a$ , with the default value of  $a$  and a special thread ID  $t_{init}$ . We assume Lock and Unlock operations are properly nested. Each trailing Unlock stores the  $id$  of the matching Lock in its  $matchID$  field.

#### 5.4.2 Control flow

It is a major challenge to specify control flow in the context of nondeterministic thread interleavings. This problem is solved by (i) transforming control related operations to auxiliary reads and writes using control variables, and (ii) imposing a set of consistency requirements on the “reads” and “writes” of control variables similar to that of normal reads and writes. The detailed steps are as follows:

- For each branch instruction  $i$ , say  $if(p)$ , add a unique auxiliary control variable  $c$ , and transform instruction  $i$  to an operation  $i'$  with the format of  $c = p$ . Operation  $i'$  is said to be a control operation (**op**  $i' = Control$ ), and can be regarded as an *assignment* to the control variable  $c$ .
- Every operation  $i$  has a  $ctrExpr$  field that stores its *path predicate*, which is a boolean expression on a set of control variables dictating the condition for  $i$  to be executed. An operation  $i$  can be regarded as a *usage* of the involved control variables in the path predicate. Without loops, the path predicate for every operation can be determined during the preprocessing phase. This can be achieved based on a thread local analysis since control variables are thread local.

- An operation  $i$  is *feasible* if its *ctrExpr* field evaluates to *True*. A predicate **fb** is defined to check the feasibility of an operation.
- In the memory ordering rules, feasibility of the involved operations is checked to make sure the consistency of control flow is satisfied.

By converting control blocks to assignments and usages of control variables, we can specify consistency rules for control flow in a fashion similar to data flow.

### 5.4.3 Loops

Loops are not directly supported in this specification. For the purpose of defining a memory model alone, nonetheless, our mechanism for handling control operations is sufficient for loops. This is because the task of a memory model specification can be regarded as answering the question of whether a given execution is allowed by the memory model. For any concrete terminated execution, loops have already been resolved to a finite number of iterations.

To enable a fully automatic and exhaustive program analysis involving loops, it is possible to develop another level of constraints in which the path predicate of an operation can conditionally grow until all constraints are satisfied. Other tools, such as Extended Static Checker for Java (ESC/Java) [29], rely on the user to supply loop invariants — loops without invariants are handled in a manner that is unsound but still useful. This approach can be adopted by our system as well. In practice, our experience shows that unwrapping a loop into fixed iterations is often effective (although this loses soundness) for property checking.

## 5.5 Language semantics

The semantics of the source language is defined as a collection of constraints. This section explains each of the rules. As shown below, predicate **legalSC** is the overall constraint that defines the requirement of sequential consistency on an execution  $ops$  in which the operations follow an ordering relation  $order$ .

$$\begin{aligned} \mathbf{legalSC} \ ops \ order &\equiv \\ &\mathbf{requireProgramOrder} \ ops \ order \wedge \\ &\mathbf{requireReadValue} \ ops \ order \wedge \\ &\mathbf{requireComputation} \ ops \ order \wedge \\ &\mathbf{requireMutualExclusion} \ ops \ order \wedge \end{aligned}$$

**requireWeakTotalOrder**  $ops\ order \wedge$   
**requireTransitiveOrder**  $ops\ order \wedge$   
**requireAsymmetricOrder**  $ops\ order$

### 5.5.1 Program order rule

Constraint **requireProgramOrder** specifies that operations should respect program order, which is formalized by **orderedByProgram**. In addition, the default writes are ordered before other operations. Note that predicate **fb** is used to ensure the feasibility of the involved operations.

**requireProgramOrder**  $ops\ order \equiv \forall i, j \in ops.$

$(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge (\mathbf{orderedByProgram}\ i\ j \vee \mathbf{t}\ i = t_{init} \wedge \mathbf{t}\ j \neq t_{init})) \Rightarrow$   
 $\mathbf{order}\ i\ j$

### 5.5.2 Read value rules

Constraint **requireReadValue** enforces the consistency of data flow across reads and writes. The assignments and usages of local variables (thread local data dependence) and control variables (thread local control dependence) follow the similar guideline to ensure consistent data transfer. Therefore, **requireReadValue** is decomposed into three subrules for global reads, local reads, and control reads, respectively. Because we apply *unique* control variables, **controlReadValue** does not need to be concerned with potential overwriting assignments.

**requireReadValue**  $ops\ order \equiv$   
**globalReadValue**  $ops\ order \wedge$   
**localReadValue**  $ops\ order \wedge$   
**controlReadValue**  $ops\ order$

**globalReadValue**  $ops\ order \equiv \forall k \in ops.$

$(\mathbf{fb}\ k \wedge \mathbf{isRead}\ k) \Rightarrow$   
 $(\exists i \in ops. \mathbf{fb}\ i \wedge \mathbf{op}\ i = \mathbf{Write} \wedge \mathbf{var}\ i = \mathbf{var}\ k \wedge$   
 $\mathbf{data}\ i = \mathbf{data}\ k \wedge \neg(\mathbf{order}\ k\ i) \wedge$   
 $(\neg \exists j \in ops. \mathbf{fb}\ j \wedge \mathbf{op}\ j = \mathbf{Write} \wedge \mathbf{var}\ j = \mathbf{var}\ k \wedge$   
 $\mathbf{order}\ i\ j \wedge \mathbf{order}\ j\ k))$

**localReadValue**  $ops\ order \equiv \forall k \in ops. \mathbf{fb}\ k \Rightarrow$

$(\forall e \in (\mathbf{getLocals}\ k).$   
 $(\exists i \in ops. (\mathbf{fb}\ i \wedge \mathbf{isAssign}\ i \wedge \mathbf{local}\ i = \mathbf{getVar}\ e \wedge$   
 $\mathbf{data}\ i = \mathbf{getData}\ e \wedge \mathbf{orderedByProgram}\ i\ k) \wedge$   
 $(\neg \exists j \in ops. (\mathbf{fb}\ j \wedge \mathbf{isAssign}\ j \wedge \mathbf{local}\ j = \mathbf{getVar}\ e \wedge$

**orderedByProgram**  $i j \wedge \text{orderedByProgram } j k$ )))))

**controlReadValue**  $ops \text{ order} \equiv \forall k \in ops.$   
 $(\forall e \in (\text{getCtrls } k).$   
 $(\exists i \in ops. \text{op } i = \text{Control} \wedge \text{var } i = \text{getVar } e \wedge$   
 $\text{data } i = \text{getData } e \wedge \text{orderedByProgram } i k))$

### 5.5.3 Computation rule

Constraint **requireComputation** enforces the program semantics. It is not directly related to the memory ordering, but is needed for analyzing realistic code. It requires that for every operation involving computations (i.e., when the operation type is *Computation* or *Control*), the resultant data must be obtained by properly evaluating the expression in the operation. For brevity, we omit some details of the standard program semantics that are usually well understood. For example, we use a predicate **eval** to indicate that standard process should be followed to evaluate an expression. Similarly, **getLocals** and **getCtrls** are used to parse the *cmpExpr* and *ctrExpr* fields to obtain a set of ⟨variable, data⟩ entries involved in the expressions (these entries represent the local/control variables that the operation depends on and their associated data values), which can be subsequently processed by **getVar** and **getData**.

**requireComputation**  $ops \text{ order} \equiv \forall k \in ops.$   
 $((\text{fb } k \wedge \text{op } k = \text{Computation}) \Rightarrow$   
 $(\text{data } k = \text{eval } (\text{cmpExpr } k))) \wedge$   
 $((\text{fb } k \wedge \text{op } k = \text{Control}) \Rightarrow$   
 $(\text{data } k = \text{eval } (\text{ctrExpr } k)))$

### 5.5.4 Mutual exclusion rule

Constraint **requireMutualExclusion** enforces mutual exclusion for operations enclosed by matched Lock and Unlock operations.

**requireMutualExclusion**  $ops \text{ order} \equiv \forall i, j \in ops.$   
 $(\text{fb } i \wedge \text{fb } j \wedge \text{matchLock } i j) \Rightarrow$   
 $(\neg \exists k \in ops. \text{fb } k \wedge \text{isSync } k \wedge$   
 $\text{lock } k = \text{lock } i \wedge \text{t } k \neq \text{t } i \wedge \text{order } i k \wedge \text{order } k j)$

### 5.5.5 General ordering rules

These constraints require *order* to be transitive, total, and asymmetric.

**requireWeakTotalOrder**  $ops\ order \equiv \forall i, j \in ops.$   
 $(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge \mathbf{id}\ i \neq \mathbf{id}\ j) \Rightarrow (\mathbf{order}\ i\ j \vee \mathbf{order}\ j\ i)$

**requireTransitiveOrder**  $ops\ order \equiv \forall i, j, k \in ops.$   
 $(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge \mathbf{fb}\ k \wedge \mathbf{order}\ i\ j \wedge \mathbf{order}\ j\ k) \Rightarrow \mathbf{order}\ i\ k$

**requireAsymmetricOrder**  $ops\ order \equiv \forall i, j \in ops.$   
 $(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge \mathbf{order}\ i\ j) \Rightarrow \neg(\mathbf{order}\ j\ i)$

### 5.5.6 Auxiliary definitions.

Several helper predicates are defined.

**fb**  $i \equiv (\mathbf{eval}\ (\mathbf{ctrExpr}\ i) = \mathit{True})$

**orderedByProgram**  $i\ j \equiv (\mathbf{t}\ i = \mathbf{t}\ j \wedge \mathbf{pc}\ i < \mathbf{pc}\ j)$

**isAssign**  $i \equiv (\mathbf{op}\ i = \mathit{Computation} \vee \mathbf{op}\ i = \mathit{Read})$

**isSync**  $i \equiv (\mathbf{op}\ i = \mathit{Lock} \vee \mathbf{op}\ i = \mathit{Unlock})$

**matchLock**  $i\ j \equiv \mathbf{op}\ i = \mathit{Lock} \wedge \mathbf{op}\ j = \mathit{Unlock} \wedge \mathbf{matchID}\ j = \mathbf{id}\ i$

The following predicates are not explicitly defined here since they are typically well understood.

<b>eval</b> $exp$ :	evaluate $exp$ with standard program semantics;
<b>getLocals</b> $k$ :	parse $k$ and get the set of local variables that $k$ depends on, with their associated data values;
<b>getCtrs</b> $k$ :	parse the path predicate of $k$ and get the set of control variables that $k$ depends on, with their associated data values;
<b>getVar</b> $e$ :	get variable from a $\langle \text{variable}, \text{data} \rangle$ entry;
<b>getData</b> $e$ :	get data from a $\langle \text{variable}, \text{data} \rangle$ entry.

After the language semantics is defined, program properties can be formalized as additional constraints to enable memory-model-sensitive analysis. The following section discusses three important applications of this approach.

## 5.6 Applications

### 5.6.1 Execution validation

A direct application of the formal language specification is for execution validation, which formalizes the task of conducting litmus tests. As previously mentioned, studying thread behaviors with small code fragments is very helpful for understanding the implications of a threading model. Many memory model proposals rely on a collection of litmus tests to illustrate critical properties. Section 2.6 shows that it is effective to abstract a common programming pattern as a litmus test to support verification. In [38], execution validation is also applied in post-silicon verification for commercial multiprocessors.

Defining the legality of thread behaviors is the common goal for all memory model specifications, but the ability to automatically validate an execution has been lacking in previous declarative specification methods. Our system supports such an analysis by allowing a user to add annotations about the read values, and verifying those assertions automatically via constraint solving.

Constraint `validateExecution` verifies whether a given execution *ops* is legal under the formal model.

**validateExecution** *ops*  $\equiv (\exists \textit{order}. \textit{legalSC } \textit{ops } \textit{order})$

Concrete examples will be discussed in Section 5.7 to demonstrate how to apply such a formal specification to enable computer aided analysis.

### 5.6.2 Race detection

Race conditions are usually inadvertently introduced and may lead to unexpected behaviors that are hard to debug. Therefore, catching these potential defects is highly useful for developing reliable software. Furthermore, many relaxed memory systems guarantee that race-free programs behave in the same way as sequentially consistent programs, which allows programmers to rely on their intuitions about SC during software development. This also makes race-detection even more important in practice.

The race definition given here is according to [10], which has also been adopted by the new JMM draft [4]. In these proposals, a *happens-before* order (based on

Lamport’s *happened-before* order [51] for message passing systems) is used for formalizing *concurrent* memory accesses. Further, data-race-free programs (also referred to as *correctly synchronized programs*) are defined as being free of conflicting and concurrent accesses under all *sequentially consistent* executions. The reason for using SC executions to define data races is to make it easier for a programmer to determine whether a program is correctly synchronized.

Constraint `detectDataRace` is defined to catch any potential data races. This constraint attempts to find a total order `scOrder` and a happens-before order `hbOrder` such that there exists a pair of conflicting operations that are not ordered by `hbOrder`. This formalizes the notion of “data races under sequentially consistent executions.”

**detectDataRace**  $ops \equiv \exists scOrder, hbOrder.$   
**legalSC**  $ops\ scOrder \wedge$   
**requireHbOrder**  $ops\ hbOrder\ scOrder \wedge$   
**mapConstraints**  $ops\ hbOrder\ scOrder \wedge$   
**existDataRace**  $ops\ hbOrder$

Happens-before order is defined in `requireHbOrder`. Intuitively, it states that two operations are ordered by happens-before order if (i) they are program ordered, (ii) they are ordered by synchronization operations, or (iii) they are transitively ordered by a third operation.

**requireHbOrder**  $ops\ hbOrder\ scOrder \equiv$   
**requireProgramOrder**  $ops\ hbOrder \wedge$   
**requireSyncOrder**  $ops\ hbOrder\ scOrder \wedge$   
**requireTransitiveOrder**  $ops\ hbOrder$

Since sequential consistency requires a total order among all operations, the happens-before edges induced by synchronization operations must follow this total order. This is captured by `requireSyncOrder`. Similarly, `mapConstraints` is used to make sure `scOrder` is consistent with `hbOrder`.

**requireSyncOrder**  $ops\ hbOrder\ scOrder \equiv \forall i\ j \in ops.$   
 $(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge \mathbf{isSync}\ i \wedge \mathbf{isSync}\ j \wedge \mathbf{scOrder}\ i\ j) \Rightarrow \mathbf{hbOrder}\ i\ j$

**mapConstraints**  $ops\ hbOrder\ scOrder \equiv \forall i\ j \in ops.$   
 $(\mathbf{fb}\ i \wedge \mathbf{fb}\ j \wedge \mathbf{hbOrder}\ i\ j) \Rightarrow \mathbf{scOrder}\ i\ j$

With a precise definition of happens-before order, a race condition can be formalized in constraint `existDataRace`. A race is caused by two feasible operation that

are (i) *conflicting*, i.e., they access the same variable from different threads ( $\mathbf{t} i \neq \mathbf{t} j$ ) and at least one of them is a write, and (ii) *concurrent*, i.e., they are not ordered by happens-before order.

**existDataRace**  $ops \text{ hbOrder} \equiv \exists i, j \in ops.$   
 $\mathbf{fb} i \wedge \mathbf{fb} j \wedge \mathbf{t} i \neq \mathbf{t} j \wedge \mathbf{var} i = \mathbf{var} j \wedge$   
 $(\mathbf{op} i = \mathit{Write} \wedge \mathbf{op} j = \mathit{Write} \vee$   
 $\mathbf{op} i = \mathit{Write} \wedge \mathbf{op} j = \mathit{Read} \vee$   
 $\mathbf{op} i = \mathit{Read} \wedge \mathbf{op} j = \mathit{Write}) \wedge$   
 $\neg(\mathbf{hbOrder} i j) \wedge \neg(\mathbf{hbOrder} j i)$

To support race analysis for the new JMM proposal, this race definition needs to be extended, e.g., by adding semantics for volatile variable operations — which should be a relatively straightforward process.

### 5.6.3 Atomicity verification

Atomicity ensures certain atomic transactions. If atomicity can be verified, a compiler may ignore the fine-grained interleavings and apply standard sequential compilation techniques when treating an atomic block. However, race-freedom is neither *necessary* nor *sufficient* to ensure atomicity. As shown by the example in Table 5.1, a monitor-style mutual exclusion mechanism, if used improperly, cannot guarantee atomicity even if the code is race-free. Therefore, a different mechanism is needed to specify and verify atomicity.

For this purpose, we allow a programmer to annotate an atomic block by enclosing it with keywords *AtomicEnter* and *AtomicExit*. To simplify some implementation details, we assume that the annotations are properly inserted. For the operation tuple, we add three more fields: *abEnter*, *abExit*, and *matchAbID*.

<b>abEnter</b> $i = abEnter :$	if $i$ is the start of an atomic block;
<b>abExit</b> $i = abExit :$	if $i$ is the end of an atomic block;
<b>matchAbID</b> $i = matchAbID :$	ID of the matching start of the atomic block.

During the preprocessing phase, we set up the operation  $i$  that immediately *follows* an *AtomicEnter* with **abEnter**  $i = \mathit{True}$ . Similarly, we set up the operation  $j$  that immediately *precedes* the matching *AtomicExit* with **abExit**  $j = \mathit{True}$ . We also

record the *id* of *i* into the *matchAbID* field of *j* (**matchAbID** *j* = **id** *i*). Given an execution *ops* transformed from an annotated program, constraint **verifyAtomicity** is used to catch atomicity violations.

**verifyAtomicity** *ops*  $\equiv \exists$  *order*.  
**legalSC** *ops order*  $\wedge$   
**existAtomicityViolation** *ops order*

**existAtomicityViolation** *ops order*  $\equiv$   
 $\exists i, j, k \in ops.$   
**fb** *i*  $\wedge$  **fb** *j*  $\wedge$  **fb** *k*  $\wedge$   
**abEnter** *i*  $\wedge$  **abExit** *j*  $\wedge$   
**id** *i* = **matchAtID** *j*  $\wedge$  **id** *i*  $\neq$  **id** *j*  $\wedge$   
**isViolation** *k i*  $\wedge$   
 $\neg(\mathbf{order} \ k \ i) \wedge \neg(\mathbf{order} \ j \ k)$

**isViolation** *k i*  $\equiv (\mathbf{t} \ k \neq \mathbf{t} \ i)$

The definition of **existAtomicityViolation** is generic, in that **isViolation** can be fine-tuned to capture other desired semantics. For illustration purposes, we only provide a simple but strong requirement here. It states that no operation from another thread can be interleaved between the atomic block. In practice, it is benign to interleave certain operations as long as the effect cannot be observed. For example, it might be desirable to define a “variable window” (a set of variables manipulated within an atomic block) and detect an atomicity violation only when the intruding operation “overlaps” the variable window.

## 5.7 Implementation

Constraint-based analyses can be quickly prototyped using a constraint logic programming language. A tool named *DefectFinder* has been written in SICStus Prolog to test the proposed techniques.

### 5.7.1 Constraint generation

As pointed out in Section 4.4, translating the constraints in predicate logic to Prolog is straightforward. For instance, the original definition and Prolog implementation for **requireWeakTotalOrder** are shown as follows.

Definition in predicate logic:

```

requireWeakTotalOrder ops order  $\equiv \forall i, j \in ops.$ 
    (fb  $i \wedge$  fb  $j \wedge$  id  $i \neq$  id  $j) \Rightarrow$  (order  $i j \vee$  order  $j i$ )

```

Definition in Prolog:

```

requireWeakTotalOrder(Ops,Order,FbList):-
    forEachElem(Ops,Order,FbList,doWeakTotalOrder).

elemProg(doWeakTotalOrder,Ops,Order,FbList,I,J):-
    const(feasible,Feasible),
    length(Ops,N),
    matrix_elem(Order,N,I,J,0ij),
    matrix_elem(Order,N,J,I,0ji),
    nth(I,FbList,Fi),
    nth(J,FbList,Fj),
    (Fi  $\# =$  Feasible  $\#/\ \wedge$  Fj  $\# =$  Feasible  $\#/\ \wedge$  I  $\# \neq$  J)
     $\# \Rightarrow$  (0ij  $\# \neq$  0ji).

```

### 5.7.2 Concurrency analysis

To illustrate how the tool works, recall program  $b$  in Figure 5.1. Consider the problem of checking whether  $r1 = 1$  and  $r2 = 0$  is allowed by sequential consistency. Table 5.2 displays the corresponding execution derived from the program text (it only shows the operation fields relevant to this example). When constraint `validateExecution` is imposed on this execution, `DefectFinder` immediately finds a legal order that satisfies the constraint and outputs its adjacency matrix, as shown in Table 5.3. Here, the value  $X$  of a matrix element means the ordering relation between  $i$  and  $j$  has not been instantiated based on the accumulated constraints. In general, there usually exist many  $X$  entries where alternative interleavings are allowed. If desired, a Prolog predicate *labeling* can be called to instantiate all variables. Our tool also outputs a possible interleaving  $1\ 2\ 6\ 7\ 8\ 3\ 4\ 5$  which is automatically derived from this matrix.

If the execution with another output,  $r1 = 0$  and  $r2 = 1$ , is checked, the tool would quickly determine that it is illegal since no ordering relation can be found to

**Table 5.3.** The adjacency matrix for the execution shown in Table 5.2 under sequential consistency.

	1	2	3	4	5	6	7	8
1	0	X	1	1	1	1	1	1
2	X	0	1	1	1	1	1	1
3	0	0	0	1	1	0	0	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	1	1	1	0	1	1
7	0	0	1	1	1	0	0	1
8	0	0	1	1	1	0	0	0

satisfy all constraints.

Applying DefectFinder for a different application simply involves selecting the corresponding goal. For example, if the programs in Figure 5.1 are checked for race conditions, the tool would report that program *a* is race-free and program *b* is not, in which case the conflicting operations and an interleaving that leads to the race conditions are displayed.

Similarly, when the program in Table 5.1 is verified for race conditions, our utility would report that it is race-free. However, an atomicity violation would be detected if the transaction is annotated by an atomic block. Having detected this defect, the user can subsequently modify the code and do the test again. For instance, if a transaction is protected by a single Lock/Unlock pair and both transactions use the same lock, the bug would be removed.

### 5.7.3 Performance results

Our tool has been applied to analyze a large collection of litmus tests. Table 5.4 summarizes the performance results of the examples discussed in this section. These analyses are performed using a Pentium 366 MHz PC with 128 MB of RAM running Windows 2000. SICStus Prolog is run under compiled mode.

## 5.8 Summary

This chapter demonstrates how to verify high level correctness properties in a multithreaded program. Unlike previous approaches, which need to assume a

**Table 5.4.** Performance statistics.

Test Program	Property	Result	Time (sec)
Program a in Figure 5.1	r1=1 and r2=0? Race Conditions	illegal no races	6.790 6.810
Program b in Figure 5.1	r1=1 and r2=0? Race Conditions	legal has races	0.401 0.811
Program in Table 5.1	Race Conditions Atomicity	no races violated	18.940 2.955

simplified memory model, our method helps a user to perform rigorous program analysis against the exact thread semantics. This approach offers the following benefits:

- It is rigorous. Based on formal definitions of program properties and memory model rules, this approach enables a precise semantic analysis. Specifications developed in such a rigorous manner can also be sent to a theorem proving utility, such as the HOL theorem prover [41], for proving generic properties.
- It is automatic. This approach allows one to take advantage of the tremendous advances in constraint/SAT solving techniques. The executable thread semantics can also be treated as a “black box” whereby the users are not necessarily required to understand all the details of the model to benefit from the tool.
- It is generic. Since this method is not limited to a specific synchronization mechanism (e.g., lock-based method), it can be applied to reason about various correctness properties for any threading model, all using the same framework.

In terms of scalability, there are two aspects involved. One is the complexity of the shared memory system that can be modelled. The other is the size of programs that can be analyzed. For the former aspect, Nemos scales well with its compositional specification style. As for the latter aspect, our CLP prototype tool currently only handles small litmus tests. However, there is still a lot of room for improvement. For instance, one can add a “constraint configuration” component that automatically filters out or reorders certain rules according to the input program, e.g., rules regard-

ing control flow can be excluded if the program does not involve branch statements. More efficient solving algorithms may also help improve our approach.

## CHAPTER 6

### CONCLUSIONS

#### 6.1 Thesis summary

The setting in which contemporary memory models are expressed and analyzed needs to be improved. Toward this, this dissertation develops techniques to make memory model specifications clear and executable, and shows that it is both beneficial and feasible to capture not only architectural level but also language level memory models using the same framework. The importance of doing so is growing, especially with the advent of multiprocessor architectures on which the language level thread semantics needs to be supported in the most efficient manner. In summary, this dissertation has made contributions in the following areas.

##### 6.1.1 Operational specification techniques

The design and application of the UMM framework is discussed. With this framework, a designer can specify and analyze shared memory consistency requirements using a systematic approach as follows: (i) formally define thread semantics as guarded commands using the UMM transition system, (ii) encode such a specification, along with idiom-driven test programs, using a model checker, and (iii) automatically enumerate all executions of the test programs to verify pivotal memory model properties. Making memory models executable greatly reduces the likelihood of having overlooked corner cases.

##### 6.1.2 Nonoperational specification techniques

The Nemos framework is presented to capture memory model semantics in a declarative and executable style. The memory consistency requirements can be encoded as constraint logic programs or as boolean satisfiability problems. These techniques are demonstrated through the formalization of a collection of memory

models. Being able to provide such a wide coverage and tackle cutting-edge commercial architectures attests to the flexibility and scalability of this framework. Nemos is designed with a special emphasis to support verification, allowing one to plug in existing constraint or SAT solvers for exercising parallel programs with respect to the underlying memory model. The compositional specification style allows one to analyze a model piece by piece. The modular approach also makes the Nemos framework scalable.

### 6.1.3 Reasoning about multithreaded programs

Defining a precise concurrency model is important for enabling a variety of analyses. A novel approach that handles both program semantics and memory model semantics in a declarative constraint-based framework is introduced. Three concrete applications — execution validation, race detection, and atomicity verification — demonstrate the effectiveness of applying such a memory-model-sensitive analysis tool for verifying multithreaded programs that, albeit small, can be extremely difficult to analyze by hand. This analytical framework is particularly useful in helping people understand the underlying concurrency model and conduct verification for common programming patterns. The capability of studying program correctness under relaxed memory models is also essential in verifying critical components of important programs such as JVMs and garbage collectors that run on weak memory systems.

## 6.2 Future directions

### 6.2.1 Framework enhancements

The scalability of our analytical frameworks can be improved by using techniques such as predicate abstraction, branch refinement, or assume-guarantee. Programming pattern annotation and inference techniques can also help facilitate program abstraction. The current verification prototype tools can be augmented with a first order decision procedure package for checking more realistic multithreaded programs. Also, the constraint solving or SAT solving algorithms for the constraint-based framework can be improved. In particular, the domain specific constraints may provide useful structural information that can be exploited for developing faster solving algorithms.

### 6.2.2 Memory-model-sensitive compilers

Compilation techniques for multithreaded programs should be explored. The specification methods presented in this dissertation might be useful for enabling precise dependence analysis and reachability analysis. In addition to checking correctness properties, our frameworks can be extended to automatically generate optimized code. Such a tool may be used to synthesize critical instructions of concurrent programs comprising common synchronization primitives. There exists a rich body of previous work in parallelizing compilers that convert a sequential program to parallel code, but little has been done in memory-model-sensitive compilers that can fully take advantage of the optimization opportunities allowed by a relaxed memory system.

### 6.2.3 Other application domains

Although this dissertation has focused on multithreaded programming, the techniques developed here serve a common goal — how to properly specify certain requirements and verify that an implementation satisfies the specification. Therefore, our methods are also promising in solving problems in other application domains. For example, one interesting area is embedded systems. Embedded systems usually impose certain functional, resource, or real-time constraints, which can be formalized and automatically checked via constraint solving.

# APPENDIX A

## SPECIFYING $JMM_{MP}$ USING UMM

To enable formal verification,  $JMM_{MP}$  is adapted using UMM. The core JMM semantics formalized here is primarily based on  $JMM_{MP}$  [56] as of January 11, 2002.

### A.1 Terminology

#### A.1.1 Variables

In the Java memory model, a *global variable* refers to a static field of a loaded class, an instance field of an allocated object, or an element of an allocated array. It can be further categorized as a *normal*, *volatile*, or *final* variable. A *local variable* corresponds to a Java local variable or an operand stack location.

#### A.1.2 Instructions

The instruction tuple is extended to carry local variable and locking information. Since the proposed semantics does not enforce a *unique* per-thread visibility order for any observing thread, a write does not need to be decomposed. An instruction  $i$  is denoted by  $\langle t, pc, op, var, data, local, useLocal, lock, time \rangle$ , where

$t(i) = t$ :	issuing thread;
$pc(i) = pc$ :	program counter;
$op(i) = op$ :	operation type;
$var(i) = var$ :	variable;
$data(i) = data$ :	data value;
$local(i) = local$ :	local variable;
$useLocal(i) = useLocal$ :	tag to indicate if the write value is provided by $local(i)$ ;
$lock(i) = lock$ :	lock;
$time(i) = time$ :	global time stamp, <i>incremented each time</i> an instruction is added to GIB.

### A.1.3 The extended memory abstraction

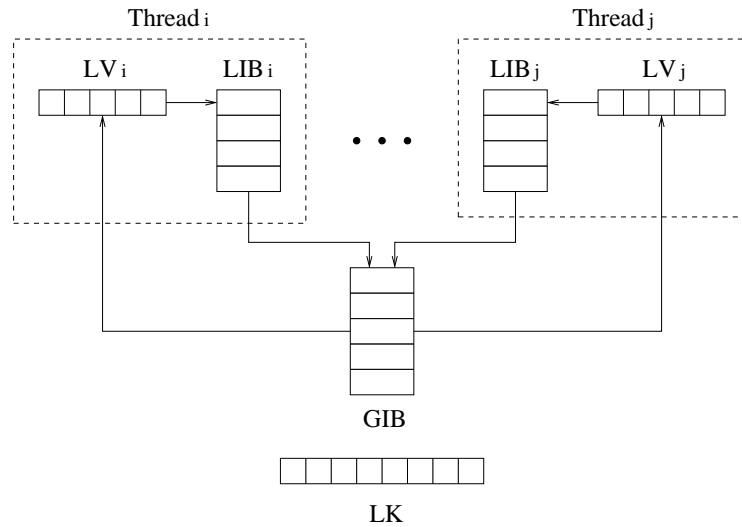
To capture the additional requirements regarding local variables and locks in the Java memory model, the conceptual architecture of the UMM transition system is slightly extended. Figure A.1 shows the abstract machine for modelling the Java memory model. In addition to the local instruction buffer, each thread  $k$  also maintains a *local variable array*  $LV_k$ . Each element  $LV_k[v]$  contains the data value of the local variable  $v$ . To maintain the locking status, a dedicated global *lock array*  $LK$  is also added. Each element  $LK[l]$  is a tuple  $\langle count, owner \rangle$ , where *count* is the number of recursive lock acquisitions and *owner* is the owning thread.

## A.2 Initial conditions

LIB initially contains instructions from each thread in program order. GIB initially contains the default write instructions for every variable  $v$  (with the default value of  $v$ , a special thread ID  $t_{init}$ , and a *time* field of 0). The *count* fields in  $LK$  are initially set to 0.

## A.3 Transition table for the Java memory model

Java memory operations are defined in the transition table given in Table A.1. A *read* operation on a global variable corresponds to the Java program instruction



**Figure A.1.** Extended conceptual architecture for the Java memory model.

**Table A.1.** Transition table for the alternative Java memory model.

Event	Condition	Action
readNormal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{ReadNormal} \wedge$ $(\exists w \in \text{GIB} : \text{legalNormalWrite}(i, w))$	$\text{LV}_{t(i)}[\text{local}(i)] := \text{data}(w);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
writeNormal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{WriteNormal}$	<b>if</b> ( $\text{useLocal}(i)$ ) $i.\text{data} := \text{LV}_{t(i)}[\text{local}(i)];$ <b>end;</b> $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
lock	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{Lock} \wedge$ $(\text{LK}[\text{lock}(i)].\text{count} = 0 \vee$ $\text{LK}[\text{lock}(i)].\text{owner} = t(i))$	$\text{LK}[\text{lock}(i)].\text{count} :=$ $\text{LK}[\text{lock}(i)].\text{count} + 1;$ $\text{LK}[\text{lock}(i)].\text{owner} := t(i);$ $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
unlock	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{Unlock} \wedge$ $(\text{LK}[\text{lock}(i)].\text{count} > 0 \wedge$ $\text{LK}[\text{lock}(i)].\text{owner} = t(i))$	$\text{LK}[\text{lock}(i)].\text{count} :=$ $\text{LK}[\text{lock}(i)].\text{count} - 1;$ $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
readVolatile	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{ReadVolatile} \wedge$ $(\exists w \in \text{GIB} : \text{legalVolatileWrite}(i, w))$	$\text{LV}_{t(i)}[\text{local}(i)] := \text{data}(w);$ $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
writeVolatile	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{WriteVolatile}$	<b>if</b> ( $\text{useLocal}(i)$ ) $i.\text{data} := \text{LV}_{t(i)}[\text{local}(i)];$ <b>end;</b> $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
readFinal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{ReadFinal} \wedge$ $(\exists w \in \text{GIB} : \text{legalFinalWrite}(i, w))$	$\text{LV}_{t(i)}[\text{local}(i)] := \text{data}(w);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
writeFinal	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{WriteFinal}$	<b>if</b> ( $\text{useLocal}(i)$ ) $i.\text{data} := \text{LV}_{t(i)}[\text{local}(i)];$ <b>end;</b> $\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$
freeze	$\exists i \in \text{LIB}_{t(i)} : \text{ready}(i) \wedge$ $op(i) = \text{Freeze}$	$\text{GIB} := \text{append}(\text{GIB}, i);$ $\text{LIB}_{t(i)} := \text{delete}(\text{LIB}_{t(i)}, i);$

with a format of  $r1 = a$ . It always stores the data value in the target local variable. A *write* operation on a global variable can have two formats,  $a = r1$  or  $a = 1$ , depending on whether the *useLocal* tag is set. *Lock* and *unlock* instructions are



does not impose any synchronization effect. Condition *ready* enforces the bypassing policy of the memory model as well as local data dependence. The helper function *notRedundant(j)* returns *true* if instruction *j* does have a synchronization effect.

Data dependence imposed by the usage of conflicting local variables is expressed in condition *localDependent*. The helper function *isWrite(i)* returns *true* if the operation type of *i* is *WriteNormal*, *WriteVolatile*, or *WriteFinal*. Similarly, *isRead(i)* returns *true* if the operation of *i* is *ReadNormal*, *ReadVolatile*, or *ReadFinal*.

$$\begin{aligned} \text{ready}(i) \equiv & \\ & \neg \exists j \in \text{LIB}_{t(i)} : pc(j) < pc(i) \wedge \\ & (\text{localDependent}(i, j) \vee \\ & \text{BYPASS}[op(j)][op(i)] = \text{No} \vee \\ & \text{BYPASS}[op(j)][op(i)] = \text{RdtLk} \wedge \text{notRedundant}(j)) \end{aligned}$$

$$\begin{aligned} \text{localDependent}(i, j) \equiv & \\ & t(j) = t(i) \wedge \text{local}(j) = \text{local}(i) \wedge \\ & (\text{isWrite}(i) \wedge \text{useLocal}(i) \wedge \text{isRead}(j) \vee \\ & \text{isWrite}(j) \wedge \text{useLocal}(j) \wedge \text{isRead}(i) \vee \\ & \text{isRead}(i) \wedge \text{isRead}(j)) \end{aligned}$$

## A.5 Visibility ordering requirement for the JMM

JMM<sub>MP</sub> applies an ordering constraint similar to location consistency. As captured in condition *LCOrder*, two instructions are ordered if one of the following cases holds:

1. they are ordered by program order,
2. they are synchronized by the same lock or the same volatile variable, or
3. there exists another operation that can transitively establish the order.

$$\begin{aligned} \text{LCOrder}(i1, i2) \equiv & \\ & (t(i1) = t(i2) \wedge pc(i1) > pc(i2) \vee t(i1) \neq t_{init} \wedge t(i2) = t_{init}) \vee \\ & \text{synchronized}(i1, i2) \vee \\ & (\exists i' \in \text{GIB} : \text{time}(i') > \text{time}(i2) \wedge \text{time}(i') < \text{time}(i1) \wedge \\ & \text{LCOrder}(i1, i') \wedge \text{LCOrder}(i', i2)) \end{aligned}$$

The synchronization mechanism is formally captured in condition *synchronized*. Instruction *i1* can be synchronized with a previous instruction *i2* via a release/acquire process, where a lock is first released by *t(i2)* after *i2* is issued and later acquired by *t(i1)* before *i1* is issued. Release can be triggered by an *Unlock* or a *WriteVolatile* instruction. Acquire can be triggered by a *Lock* or a *ReadVolatile* instruction.

$$\begin{aligned}
\text{synchronized}(i1, i2) &\equiv \\
&\exists l, u \in \text{GIB} : \\
&(op(l) = \text{Lock} \wedge op(u) = \text{Unlock} \wedge lock(l) = lock(u) \vee \\
&op(l) = \text{ReadVolatile} \wedge op(u) = \text{WriteVolatile} \wedge var(l) = var(u)) \wedge \\
&t(l) = t(i1) \wedge (t(u) = t(i2) \vee t(i2) = t_{init}) \wedge \\
&time(i2) \leq time(u) \wedge time(u) < time(l) \wedge time(l) \leq time(i1)
\end{aligned}$$

After establishing the ordering relationship by condition *LCOrder*, the requirement of serialization is enforced in *legalNormalWrite*. Informally, a write  $w$  cannot provide its value to a read  $r$  if there exists another overwriting write  $w'$  in the ordering path.

$$\begin{aligned}
\text{legalNormalWrite}(r, w) &\equiv \\
&op(w) = \text{WriteNormal} \wedge var(w) = var(r) \wedge \\
&(t(w) = t(r) \rightarrow pc(w) < pc(r)) \wedge \\
&(\neg \exists w' \in \text{GIB} : op(w') = \text{WriteNormal} \wedge var(w') = var(r) \wedge \\
&LCOrder(r, w') \wedge LCOrder(w', w))
\end{aligned}$$

The *mutual exclusion* effect of Lock and Unlock operations is enforced by tracking the *count* and *owner* fields of each lock as specified in the transition table.

## A.6 Volatile variable semantics

We require sequential consistency for all volatile variable operations. With the uniform notation of UMM, predefined memory requirements can be easily reused. Hence, the formal definition of sequential consistency described in Section 2.3 is applied to define *ReadVolatile* and *WriteVolatile* operations. The bypassing table shown in Table A.2 prohibits any reordering among volatile operations. Condition *legalVolatileWrite*, which follows *legalWrite* in Sequential Consistency, defines the legal results for *ReadVolatile* operations.

$$\begin{aligned}
\text{legalVolatileWrite}(r, w) &\equiv \\
&op(w) = \text{WriteVolatile} \wedge var(w) = var(r) \wedge \\
&(\neg \exists w' \in \text{GIB} : op(w') = \text{WriteVolatile} \wedge var(w') = var(r) \wedge \\
&time(r) > time(w') \wedge time(w') > time(w))
\end{aligned}$$

## A.7 Final variable semantics

In Java, a final field can either be a primitive value or a reference to another object. When it is a reference, the Java language requires only that the reference itself cannot be modified in the program after its initialization but the fields of the

object it points to do not have the same guarantee.  $\text{JMM}_{\text{MP}}$  proposes to add a special rule to those nonfinal fields that are referenced by a final variable: if such a field is assigned in the constructor, its default value cannot be observed by another thread after normal construction. To achieve this,  $\text{JMM}_{\text{MP}}$  uses a special mechanism to “synchronize” initialization information from the constructing thread to the final reference and eventually to the elements contained by the final reference. However, without explicit support of immutability from the Java language, this mechanism makes the memory semantics substantially more complicated because synchronization information needs to be carried by every variable.

Since our goal is to illustrate the UMM methodology, finding the most reasonable solution for final field semantics is an orthogonal task. To make our Java memory model specification complete, yet not to distract readers with the details specific to certain semantics, a straightforward definition for final fields is given here. It is different from  $\text{JMM}_{\text{MP}}$  in that it only requires the final field itself to be a constant after being frozen. The visibility rule for final fields is shown in condition *legalFinalWrite*. The default value of the final field (when  $t(w) = t_{\text{init}}$ ) can only be observed if the final field is not frozen. In addition, the constructing thread cannot observe the default value after the final field is initialized.

$$\begin{aligned}
 \text{legalFinalWrite}(r, w) \equiv & \\
 & op(w) = \text{WriteFinal} \wedge var(w) = var(r) \wedge \\
 & (t(w) = t_{\text{init}} \rightarrow \\
 & ((\neg \exists i1 \in \text{GIB} : op(i1) = \text{Freeze} \wedge var(i1) = var(r)) \wedge \\
 & (\neg \exists i2 \in \text{GIB} : op(i2) = \text{WriteFinal} \wedge var(i2) = var(r) \wedge t(i2) = t(r))))
 \end{aligned}$$

## APPENDIX B

### ITANIUM MEMORY ORDERING RULES (IN PREDICATE LOGIC)

$\text{legal ops} \equiv \exists \text{ order}.$

$\text{requireLinearOrder ops order} \wedge \text{requireWriteOperationOrder ops order} \wedge$   
 $\text{requireProgramOrder ops order} \wedge$   
 $\text{requireMemoryDataDependence ops order} \wedge$   
 $\text{requireDataFlowDependence ops order} \wedge \text{requireCoherence ops order} \wedge$   
 $\text{requireReadValue ops order} \wedge \text{requireAtomicWBRelease ops order} \wedge$   
 $\text{requireSequentialUC ops order} \wedge \text{requireNoUCBypass ops order}$

#### B.1 General ordering requirement

$\text{requireLinearOrder ops order} \equiv$

$\text{requireWeakTotal ops order} \wedge \text{requireTransitive ops order} \wedge$   
 $\text{requireAsymmetric ops order}$

$\text{requireWeakTotal ops order} \equiv \forall i, j \in \text{ops}. \text{id } i \neq \text{id } j \Rightarrow (\text{order } i j \vee \text{order } j i)$

$\text{requireTransitive ops order} \equiv \forall i, j, k \in \text{ops}. (\text{order } i j \wedge \text{order } j k) \Rightarrow \text{order } i k$

$\text{requireAsymmetric ops order} \equiv \forall i, j \in \text{ops}. \text{order } i j \Rightarrow \neg(\text{order } j i)$

#### B.2 Write operation order

$\text{requireWriteOperationOrder ops order} \equiv \forall i, j \in \text{ops}.$

$\text{orderedByWriteOperation } i j \Rightarrow \text{order } i j$

$\text{orderedByWriteOperation } i j \equiv \text{isWr } i \wedge \text{isWr } j \wedge \text{wrID } i = \text{wrID } j \wedge$

$(\text{wrType } i = \text{Local} \wedge \text{wrType } j = \text{Remote} \wedge \text{wrProc } j = \text{p } i \vee$

$\text{wrType } i = \text{Remote} \wedge \text{wrType } j = \text{Remote} \wedge$

$\text{wrProc } i = \text{p } i \wedge \text{wrProc } j \neq \text{p } i)$

### B.3 Program order

**requireProgramOrder**  $ops\ order \equiv \forall i, j \in ops.$   
 $(\text{orderedByAcquire } i\ j \vee \text{orderedByRelease } i\ j \vee \text{orderedByFence } i\ j) \Rightarrow$   
 $\text{order } i\ j$

**orderedByProgram**  $i\ j \equiv \mathbf{p}\ i = \mathbf{p}\ j \wedge \mathbf{pc}\ i < \mathbf{pc}\ j$

**orderedByAcquire**  $i\ j \equiv \text{orderedByProgram } i\ j \wedge \mathbf{op}\ i = \mathit{ld.acq}$

**orderedByRelease**  $i\ j \equiv \text{orderedByProgram } i\ j \wedge \mathbf{op}\ j = \mathit{st.rel} \wedge$   
 $(\mathbf{isWr}\ i \Rightarrow (\mathbf{wrType}\ i = \mathit{Local} \wedge \mathbf{wrType}\ j = \mathit{Local} \vee$   
 $\mathbf{wrType}\ i = \mathit{Remote} \wedge \mathbf{wrType}\ j = \mathit{Remote} \wedge \mathbf{wrProc}\ i = \mathbf{wrProc}\ j))$

**orderedByFence**  $i\ j \equiv \text{orderedByProgram } i\ j \wedge (\mathbf{op}\ i = \mathit{mf} \vee \mathbf{op}\ j = \mathit{mf})$

### B.4 Memory-data dependence

**requireMemoryDataDependence**  $ops\ order \equiv \forall i, j \in ops.$   
 $(\text{orderedByRAW } i\ j \vee \text{orderedByWAR } i\ j \vee \text{orderedByWAW } i\ j) \Rightarrow$   
 $\text{order } i\ j$

**orderedByMemoryData**  $i\ j \equiv \text{orderedByProgram } i\ j \wedge \mathbf{var}\ i = \mathbf{var}\ j$

**orderedByRAW**  $i\ j \equiv$   
 $\text{orderedByMemoryData } i\ j \wedge \mathbf{isWr}\ i \wedge \mathbf{wrType}\ i = \mathit{Local} \wedge \mathbf{isRd}\ j$

**orderedByWAR**  $i\ j \equiv$   
 $\text{orderedByMemoryData } i\ j \wedge \mathbf{isRd}\ i \wedge \mathbf{isWr}\ j \wedge \mathbf{wrType}\ j = \mathit{Local}$

**orderedByWAW**  $i\ j \equiv \text{orderedByMemoryData } i\ j \wedge \mathbf{isWr}\ i \wedge \mathbf{isWr}\ j \wedge$   
 $(\mathbf{wrType}\ i = \mathit{Local} \wedge \mathbf{wrType}\ j = \mathit{Local} \vee$   
 $\mathbf{wrType}\ i = \mathit{Remote} \wedge \mathbf{wrType}\ j = \mathit{Remote} \wedge$   
 $\mathbf{wrProc}\ i = \mathbf{p}\ i \wedge \mathbf{wrProc}\ j = \mathbf{p}\ i)$

### B.5 Data-flow dependence

**requireDataFlowDependence**  $ops\ order \equiv \forall i, j \in ops.$   
 $\text{orderedByLocalDependence } i\ j \Rightarrow \text{order } i\ j$

**orderedByLocalDependence**  $i\ j \equiv \text{orderedByProgram } i\ j \wedge \mathbf{reg}\ i = \mathbf{reg}\ j \wedge$   
 $(\mathbf{isRd}\ i \wedge \mathbf{isRd}\ j \vee$   
 $\mathbf{isWr}\ i \wedge \mathbf{wrType}\ i = \mathit{Local} \wedge \mathbf{useReg}\ i \wedge \mathbf{isRd}\ j \vee$   
 $\mathbf{isRd}\ i \wedge \mathbf{isWr}\ j \wedge \mathbf{wrType}\ j = \mathit{Local} \wedge \mathbf{useReg}\ j)$

## B.6 Coherence

**requireCoherence**  $ops\ order \equiv \forall i, j \in ops.$   
 $(\text{isWr } i \wedge \text{isWr } j \wedge \text{var } i = \text{var } j \wedge$   
 $(\text{attribute } (\text{var } i) = WB \vee \text{attribute } (\text{var } i) = UC) \wedge$   
 $(\text{wrType } i = Local \wedge \text{wrType } j = Local \wedge \mathbf{p } i = \mathbf{p } j \vee$   
 $\text{wrType } i = Remote \wedge \text{wrType } j = Remote \wedge \text{wrProc } i = \text{wrProc } j) \wedge$   
 $\text{order } i\ j)$   
 $\Rightarrow$   
 $(\forall p, q \in ops.$   
 $(\text{isWr } p \wedge \text{isWr } q \wedge \text{wrID } p = \text{wrID } i \wedge \text{wrID } q = \text{wrID } j \wedge$   
 $\text{wrType } p = Remote \wedge \text{wrType } q = Remote \wedge \text{wrProc } p = \text{wrProc } q) \Rightarrow$   
 $\text{order } p\ q)$

## B.7 Read value

**requireReadValue**  $ops\ order \equiv \forall j \in ops.$   
 $(\text{isRd } j \Rightarrow (\text{validLocalWr } ops\ order\ j \vee \text{validRemoteWr } ops\ order\ j \vee$   
 $\text{validDefaultWr } ops\ order\ j)) \wedge ((\text{isWr } j \wedge \text{useReg } j) \Rightarrow \text{validRd } ops\ order\ j)$

**validLocalWr**  $ops\ order\ j \equiv \exists i \in ops.$   
 $(\text{isWr } i \wedge \text{wrType } i = Local \wedge \text{var } i = \text{var } j \wedge \mathbf{p } i = \mathbf{p } j \wedge$   
 $\text{data } i = \text{data } j \wedge \text{order } i\ j) \wedge$   
 $(\neg \exists k \in ops. \text{isWr } k \wedge \text{wrType } k = Local \wedge \text{var } k = \text{var } j \wedge \mathbf{p } k = \mathbf{p } j \wedge$   
 $\text{order } i\ k \wedge \text{order } k\ j)$

**validRemoteWr**  $ops\ order\ j \equiv \exists i \in ops.$   
 $(\text{isWr } i \wedge \text{wrType } i = Remote \wedge \text{wrProc } i = \mathbf{p } j \wedge \text{var } i = \text{var } j \wedge$   
 $\text{data } j = \text{data } i \wedge \neg(\text{order } j\ i)) \wedge$   
 $(\neg \exists k \in ops. \text{isWr } k \wedge \text{wrType } k = Remote \wedge \text{var } k = \text{var } j \wedge \text{wrProc } k = \mathbf{p } j \wedge$   
 $\text{order } i\ k \wedge \text{order } k\ j)$

**validDefaultWr**  $ops\ order\ j \equiv$   
 $(\neg \exists i \in ops. \text{isWr } i \wedge \text{var } i = \text{var } j \wedge \text{order } i\ j \wedge$   
 $(\text{wrType } i = Local \wedge \mathbf{p } i = \mathbf{p } j \vee \text{wrType } i = Remote \wedge \text{wrProc } i = \mathbf{p } j)) \wedge$   
 $\text{data } j = \text{default } (\text{var } j)$

**validRd**  $ops\ order\ j \equiv \exists i \in ops.$   
 $(\text{isRd } i \wedge \text{reg } i = \text{reg } j \wedge \text{orderedByProgram } i\ j \wedge \text{data } j = \text{data } i) \wedge$   
 $(\neg \exists k \in ops. \text{isRd } k \wedge \text{reg } k = \text{reg } j \wedge$   
 $\text{orderedByProgram } i\ k \wedge \text{orderedByProgram } k\ j)$

## B.8 Total ordering of WB releases

**requireAtomicWBRelease**  $ops\ order \equiv \forall i, j, k \in ops.$   
 $(\text{op } i = st.rel \wedge \text{wrType } i = Remote \wedge \text{op } k = st.rel \wedge \text{wrType } k = Remote \wedge$   
 $\text{wrID } i = \text{wrID } k \wedge \text{attribute } (\text{var } i) = WB \wedge \text{order } i\ j \wedge \text{order } j\ k) \Rightarrow$   
 $(\text{op } j = st.rel \wedge \text{wrType } j = Remote \wedge \text{wrID } j = \text{wrID } i)$

## B.9 Sequentiality of UC operations

$\text{requireSequentialUC } ops \text{ order} \equiv \forall i, j \in ops. \text{orderedByUC } i \ j \Rightarrow \text{order } i \ j$

$\text{orderedByUC } i \ j \equiv$

$\text{orderedByProgram } i \ j \wedge \text{attribute } (\text{var } i) = UC \wedge \text{attribute } (\text{var } j) = UC \wedge$   
 $(\text{isRd } i \wedge \text{isRd } j \vee$   
 $\text{isRd } i \wedge \text{isWr } j \wedge \text{wrType } j = Local \vee$   
 $\text{isWr } i \wedge \text{wrType } i = Local \wedge \text{isRd } j \vee$   
 $\text{isWr } i \wedge \text{wrType } i = Local \wedge \text{isWr } j \wedge \text{wrType } j = Local)$

## B.10 No UC bypassing

$\text{requireNoUCBypass } ops \text{ order} \equiv \forall i, j, k \in ops.$

$(\text{isWr } i \wedge \text{wrType } i = Local \wedge \text{attribute } (\text{var } i) = UC \wedge \text{isRd } j \wedge$   
 $\text{isWr } k \wedge \text{wrType } k = Remote \wedge \text{wrProc } k = p \ k \wedge \text{wrID } k = \text{wrID } i \wedge$   
 $\text{order } i \ j \wedge \text{order } j \ k) \Rightarrow$   
 $(\text{wrProc } k \neq p \ j \vee \text{var } i \neq \text{var } j)$

# APPENDIX C

## ITANIUM MEMORY ORDERING RULES

### (IN HOL)

```
legal ops =
  ?order.
  requireLinearOrder ops order /\
  requireWriteOperationOrder ops order /\
  requireProgramOrder ops order /\
  requireMemoryDataDependence ops order /\
  requireDataFlowDependence ops order /\
  requireCoherence ops order /\
  requireReadValue ops order /\
  requireAtomicWBRelease ops order /\
  requireSequentialUC ops order /\
  requireNoUCBypass ops order

requireLinearOrder ops order =
  requireWeakTotal ops order /\
  requireTransitive ops order /\
  requireAsymmetric ops order

requireWeakTotal ops order =
  !i j. i IN ops /\ j IN ops /\ ~(i.id = j.id) ==>
  order i j \/ order j i

requireTransitive ops order =
  !i j k. i IN ops /\ j IN ops /\ k IN ops ==>
  order i j /\ order j k ==> order i k

requireAsymmetric ops order =
  !i j. i IN ops /\ j IN ops /\ (i.id = j.id) ==> ~order i j

requireWriteOperationOrder ops order =
  !i j. i IN ops /\ j IN ops /\ orderedByWriteOperation i j ==>
  order i j

orderedByWriteOperation i j =
  isWr i /\ isWr j /\ (i.wrID = j.wrID) /\
  ((i.wrType = Local) /\ (j.wrType = Remote) /\ (j.wrProc = i.proc) \/
```

```

(i.wrType = Remote) /\ (j.wrType = Remote) /\
(i.wrProc = i.proc) /\ ~(j.wrProc = i.proc)

requireProgramOrder ops order =
  !i j. i IN ops /\ j IN ops ==>
    orderedByAcquire i j \/ orderedByRelease i j \/ orderedByFence i j ==>
      order i j

orderedByProgram i j =
  (i.proc = j.proc) /\ (i.pc < j.pc)

orderedByAcquire i j =
  orderedByProgram i j /\ (i.op = LdAcq)

orderedByRelease i j =
  orderedByProgram i j /\ (j.op = StRel) /\
  (isWr i ==>
    (i.wrType = Local) /\ (j.wrType = Local) \/
    (i.wrType=Remote) /\ (j.wrType=Remote) /\ (i.wrProc=j.wrProc))

orderedByFence i j =
  orderedByProgram i j /\ ((i.op = Mf) \/ (j.op = Mf))

requireMemoryDataDependence ops order =
  !i j. i IN ops /\ j IN ops ==>
    (orderedByRAW i j \/ orderedByWAR i j \/ orderedByWAW i j) ==>
      order i j

orderedByMemoryData i j =
  orderedByProgram i j /\ (i.var = j.var)

orderedByRAW i j =
  orderedByMemoryData i j /\ isWr i /\ isRd j /\ (i.wrType = Local)

orderedByWAR i j =
  orderedByMemoryData i j /\ isRd i /\ isWr j /\ (j.wrType = Local)

orderedByWAW i j =
  orderedByMemoryData i j /\ isWr i /\ isWr j /\
  ((i.wrType = Local) /\ (j.wrType = Local) \/
  (i.wrType = Remote) /\ (j.wrType = Remote) /\
  (i.wrProc = i.proc) /\ (j.wrProc = i.proc))

requireDataFlowDependence ops order =
  !i j. i IN ops /\ j IN ops ==>
    orderedByLocalDependence i j ==> order i j

```

```

orderedByLocalDependence i j =
  orderedByProgram i j /\ (i.reg = j.reg) /\
  ((isRd i /\ isRd j) \/
  (isWr i /\ isRd j /\ (i.wrType = Local) /\ i.useReg) \/
  (isRd i /\ isWr j /\ (j.wrType = Local) /\ j.useReg))

requireCoherence ops order =
  !i j. i IN ops /\ j IN ops ==>
  isWr i /\ isWr j /\ (i.var = j.var) /\ order i j /\
  ((attr_of i.var = WB) \/ (attr_of i.var = UC)) /\
  ((i.wrType=Local) /\ (j.wrType=Local) /\ (i.proc=j.proc) \/
  (i.wrType=Remote) /\ (j.wrType=Remote) /\ (i.wrProc=j.wrProc)) ==>
  !p q. p IN ops /\ q IN ops ==>
  isWr p /\ isWr q /\ (p.wrID = i.wrID) /\ (q.wrID = j.wrID) /\
  (p.wrType = Remote) /\ (q.wrType = Remote) /\ (p.wrProc = q.wrProc)
  ==> order p q

requireReadValue ops order =
  !j. j IN ops ==>
  (isRd j ==> validLocalWr ops order j \/ validRemoteWr ops order j \/
  validDefaultWr ops order j) /\
  (isWr j /\ j.useReg ==> validRd ops order j)

validLocalWr ops order j =
  ?i. i IN ops /\ isWr i /\ (i.wrType = Local) /\ (i.var = j.var) /\
  (i.proc = j.proc) /\ order i j /\ (i.data = j.data) /\
  ~?k. k IN ops /\ isWr k /\
  (k.wrType = Local) /\ (k.var=j.var) /\ (k.proc = j.proc) /\
  order i k /\ order k j

validRemoteWr ops order j =
  ?i. i IN ops /\ isWr i /\ (i.wrType = Remote) /\ (i.wrProc = j.proc) /\
  (i.var = j.var) /\ ~order j i /\ (i.data = j.data) /\
  ~?k. k IN ops /\ isWr k /\ (k.wrType = Remote) /\ (k.var = j.var) /\
  (k.wrProc = j.proc) /\ order i k /\ order k j

validDefaultWr ops order j =
  (~?i. i IN ops /\ isWr i /\ (i.var = j.var) /\ order i j /\
  (i.wrType = Local) /\ (i.proc = j.proc) \/
  (i.wrType = Remote) /\ (i.wrProc = j.proc)) /\
  (j.data = default_of j.var)

validRd ops order j =
  ?i. i IN ops /\ isRd i /\ (i.reg = j.reg) /\
  orderedByProgram i j /\ (i.data = j.data)
  /\ ~?k. k IN ops /\ isRd k /\ (k.reg = j.reg) /\
  orderedByProgram i k /\ orderedByProgram k j

```

```

requireAtomicWBRelease ops order =
  !i j k. i IN ops /\ j IN ops /\ k IN ops ==>
    (i.op = StRel) /\ (i.wrType = Remote) /\
    (k.op = StRel) /\ (k.wrType = Remote) /\
    (i.wrID = k.wrID) /\ (attr_of i.var = WB) /\ order i j /\ order j k ==>
    (j.op = StRel) /\ (j.wrType = Remote) /\ (j.wrID = i.wrID)

requireSequentialUC ops order =
  !i j. i IN ops /\ j IN ops /\ orderedByUC i j ==> order i j

orderedByUC i j =
  orderedByProgram i j /\ (attr_of i.var = UC) /\ (attr_of j.var = UC) /\
  (isRd i /\ isRd j \/\
  isRd i /\ isWr j /\ (j.wrType=Local) \/\
  isWr i /\ isRd j /\ (i.wrType=Local) \/\
  isWr i /\ isWr j /\ (i.wrType=Local) /\ (j.wrType=Local))

requireNoUCBypass ops order =
  !i j k. i IN ops /\ j IN ops /\ k IN ops ==>
  isWr i /\ isRd j /\ isWr k /\ (i.wrType = Local) /\
  (k.wrType = Remote) /\ (k.wrProc = k.proc) /\
  (k.wrID = i.wrID) /\ (attr_of i.var = UC) /\
  order i j /\ order j k ==>
  ~(k.wrProc = j.proc) \/\ ~(i.var = j.var)

```

## REFERENCES

- [1] Satzoo Incremental SAT Solver. Author: Niklas Een. Also competed in SAT'03. [http://www.math.chalmers.se/~een/Satzoo/An\\_Extensible\\_SATsolver.ps.gz](http://www.math.chalmers.se/~een/Satzoo/An_Extensible_SATsolver.ps.gz).
- [2] Intel Itanium architecture software developer's manual. <http://developer.intel.com/design/itanium/manuals.htm>.
- [3] Java memory model discussion group. <http://www.cs.umd.edu/~pugh/java/memoryModel/archive>.
- [4] Java Specification Request (JSR) 133: Java memory model and thread specification revision. <http://jcp.org/jsr/detail/133.jsp>.
- [5] Sicstus Prolog. <http://www.sics.se/sicstus>.
- [6] Tla+. <http://research.microsoft.com/users/lamport/tla/tla.html>.
- [7] A formal specification of Intel Itanium processor family memory ordering. *Application Note, Document Number: 251429-001*, October, 2002.
- [8] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.
- [9] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.
- [10] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 234–243, 1991.
- [11] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 93)*, 1993.
- [12] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, 1995.
- [13] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in relay ladder logic programs. *LNCS*, 1384:184–200, 1998.
- [14] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, 2000.

- [15] J. M. Bernabu-Aubn and V. Cholvi-Juan. Formalizing memory coherency models. *Journal of Computing and Information*, May 1994.
- [16] P. Bishop and N. Warren. *Java in Practice: Design Styles and Idioms for Effective Java*. Addison-Wesley, 1999.
- [17] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*, 2001.
- [18] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, 2003.
- [19] P. Chatterjee, H. Sivaraj, and G. Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: refinement via model-checking. In *Computer-Aided Verification (CAV'02)*, Copenhagen, Denmark, July 2002. <http://floc02.diku.dk/CAV/program.html>.
- [20] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of PLDI*, 2002.
- [21] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, 1992.
- [22] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, March 1996.
- [23] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
- [24] D. Dill. The stanford murphi verifier. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, New Jersey, July 1996. Springer-Verlag. Tool demo.
- [25] D. Dill, S. Park, and A. Nowatzky. Formal specification of abstract memory models. In *the 1993 Symposium for Research on Integrated Systems*, pages 38–52, March 1993.
- [26] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991.
- [27] C. Flanagan. Automatic software model checking using CLP. In *Proceedings of ESOP*, 2003.
- [28] C. Flanagan and S. N. Freund. Type-based race detection for Java. *Proceedings of PLDI*, pages 219–232, 2000.

- [29] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java, 2002.
- [30] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI*, 2003.
- [31] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.
- [32] G. Gao and V. Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical report, 16, CAPSL, University of Delaware, 1998.
- [33] R. Gerth. Introduction to sequential consistency and the lazy caching algorithm. *Distributed Computing*, 1995.
- [34] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, 1995.
- [35] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, Aug. 1997.
- [36] E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. In *Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 2002.
- [37] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations for Java Memory Model. In *ACM Transactions On Computer Systems*, vol. 18, No. 4, pages 333–386, November 2000.
- [38] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Proceedings of Computer Aided Verification (CAV'04)*, 2004.
- [39] M. Gordon. A formalization of a simplified subset of the alpha shared memory model. <ftp://ftp.cl.cam.ac.uk/hvg/papers/AARM.dvi.gz>.
- [40] M. Gordon. Memory access semantics for a multiprocessor instruction set. <ftp://ftp.cl.cam.ac.uk/hvg/papers/AlphaProg.dvi.gz>.
- [41] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [42] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, chapter 17. Addison-Wesley, 1996.
- [43] Y. Gurevich, W. Schulte, and C. Wallace. Investigating Java concurrency using abstract state machines. Technical Report 2000-04, University of Delaware, December 1999.
- [44] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

- [45] L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 349–356, 1997.
- [46] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.
- [47] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla+. *Formal Methods in System Design*, (2):125–131, March 2003.
- [48] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA '92)*, pages 13–21, May 1992.
- [49] P. Kohli, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume I - Architecture, pages I-332–I-335, Boca Raton, FL, 1993. CRC Press.
- [50] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.
- [51] L. Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.
- [52] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
- [53] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Dept. of Computer Science, Princeton University, Sept. 1988.
- [54] J.-W. Maessen, Arvind, and X. Shen. Improving the Java Memory Model using CRF. In *Proceedings of Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–12, October 2000.
- [55] A. C. Magalhes and A. de Melo. Defining uniform and hybrid memory consistency models on a unified framework. Thirty-second Annual Hawaii International Conference on System Sciences, Volume 8.
- [56] J. Manson and W. Pugh. Semantics of multithreaded Java. Technical Report UMIACS-TR-2001-09, 2002.
- [57] J. Manson and W. Pugh. Core semantics of multithreaded Java. In *ACM Java Grande Conference*, June 2001.
- [58] A. C. Melo and S. C. Chagas. Visual-MCM: Visualising execution histories on multiple memory consistency models. *Lecture Notes in Computer Science*, 1557:500–509, 1999.

- [59] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 235–244, 1991.
- [60] D. Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.
- [61] R. H. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.
- [62] R. H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. Technical Report CS-TR-1991-1039, 1991.
- [63] D. Park, U. Stern, and D. Dill. Java model checking. In *the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, 2000.
- [64] S. Park and D. L. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227–235, 1999.
- [65] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 47–57, 1996.
- [66] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), June 1981.
- [67] W. Pugh. Fixing the Java memory model. In *Java Grande*, pages 89–98, 1999.
- [68] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. In *Proceedings Int Conf on Parallel and Distributed Computing (PDCS'96)*, pages 125–130, Dijon, France, 1996.
- [69] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998.
- [70] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, 2001.
- [71] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *International Conference on Software Engineering*, 2002.
- [72] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [73] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of PLDI*, pages 285–297, 1989.

- [74] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *the 26th International Symposium On Computer Architecture*, Atlanta, Georgia, May 1999.
- [75] R. C. Steinke and G. J. Nutt. A lattice based framework of shared memory consistency models. In *The 21st International Conference on Distributed Computing Systems*, April 2001.
- [76] N. Sterling. Warlock - a static data race analysis tool. *USENIX Winter Technical Conference*, pages 97–106, 1993.
- [77] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java Model Checker. In *Post-CAV Workshop on Advances in Verification, Chicago*, 2000.
- [78] C. von Praun and T. Gross. Object-race detection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 70–82, 2001.
- [79] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*. Prentice Hall, 1994.
- [80] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Analyzing the CRF Java Memory Model. In *the 8th Asia-Pacific Software Engineering Conference*, 2001.
- [81] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Joint ACM Java Grande - ISCOPE Conference*, 2002.
- [82] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Rigorous concurrency analysis of multithreaded programs. Technical Report UUCS-03-026, University of Utah, November 2003.
- [83] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: An operational memory model specification framework with integrated model checking capability. *Concurrency and Computation: Practice and Experience*, 17(5–6):465–487, 2005.
- [84] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the intel itanium memory ordering rules using logic programming and sat. In *CHARME*, pages 81–95, 2003. LNCS 2860.
- [85] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium*, 2004. Accepted.
- [86] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36, 2002. LNCS 2402.