A Constraint-Based Approach for Specifying Memory Consistency Models

Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom*
School of Computing
University of Utah
E-mail: {yyang | ganesh | gary}@cs.utah.edu

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Conforming to the underlying memory consistency rules is a fundamental requirement for implementing shared memory systems and writing multiprocessor software. In order to promote understanding and enable rigorous analysis, it is highly desirable that a memory model specification be both declarative and executable.

In this paper, we explore the practicality of analyzing memory consistency models using constraint solving. By precisely defining a memory model as a set of constraints in an axiomatic and compositional style, our approach provides a generic formalism for specifying memory consistency models. Furthermore, this framework offers a unique advantage for program verification, i.e., it enables an accurate and exhaustive analysis of program properties which covers all execution paths for a given shared memory system. With this constraint-base approach, this paper formalizes a collection of classical memory models as well as the memory ordering rules of the Intel Itanium processor families. We also discuss how logic programming and constraint solving can be used to make these axiomatic specifications executable to support automated analysis.

KEYWORDS: memory consistency models, logic programming, constraint solving

1 Introduction

Two emerging trends—the tremendous advances in multiprocessor machines and the integrated support of threads from programming languages such as Java—have combined to make concurrent programming a vitally important software engineering domain. A multiprocessor program relies on a memory consistency model to determine how memory operations should appear to execute. In particular, it specifies what values may be returned by read operations considering various ordering relaxations allowed.

The design of a memory system typically involves a tradeoff between programmability and efficiency. As a natural extension of the uniprocessor model, sequential

* This work was supported in part by Research Grant No. CCR-0081406 (ITR Program) of NSF and SRC Task 1031.001. Our initial study of the Intel Itanium memory ordering rules was published in the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03).
consistency (SC) [Lamport 1979] requires all memory operations to exhibit a common total order that also respects program order. Since SC is very restrictive, many weaker memory models (see [Adve and Gharachorloo 1996] for a survey) have been proposed to provide higher performance. For example, coherence (also known as cache consistency) [Gharachorloo 1995] only enforces SC on a per-variable basis. Parallel RAM (PRAM) [Lipton and Sandberg 1988] allows each observing processor to perceive its own view of memory operation order. Causal consistency [Ahamad et al. 1994] follows a similar policy while enforcing the causal order resulting from data flow. Processor consistency (PC) [Ahamad et al. 1993] (according to Goodman’s version) combines coherence and PRAM in a mutually consistent manner. Some shared memory systems, especially modern ones, are based on hybrid models, meaning programmers can apply special synchronization operations in addition to data operations such as reads and writes. Examples of this category include release consistency [Gharachorloo et al. 1998], entry consistency [Bershad et al. 1991], and location consistency [Gao and Sarkar 1998]. Modern shared memory architectures rely on a rich set of memory access related instructions to provide the flexibility needed by software. For instance, the Intel Itanium™ processor family [Itanium Manual 2000] provides two varieties of loads and stores in addition to fence and semaphore instructions, each associated with different ordering restrictions.

A memory model impacts design decisions taken by system designers, compiler writers, as well as application developers by dictating common rules. Therefore, a memory model specification needs to be clearly understood by all these groups. Programming idioms (commonly used software patterns or algorithms) developed under one model may not work in another. To illustrate, consider Peterson’s algorithm [Peterson 1981] for mutual exclusion shown in Figure 1. The correctness of this well known algorithm depends on the assumption that a processor cannot observe the default value of a flag when checking the loop condition after the flag is set by the other processor. This crucial requirement, however, is broken by many memory systems due to optimization needs. For example, under coherence, flag1 and flag2 may both return false in the loop conditions because coherence only requires a common total order for memory operations involving the same variable. This would allow P1 and P2 to enter the critical section simultaneously, thus break-
Initially, $a = b = c = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = 1;$</td>
<td>$b = 1;$</td>
</tr>
<tr>
<td>$c = 0;$</td>
<td>$c = 2;$</td>
</tr>
<tr>
<td>$r_1 = b;$</td>
<td>$r_2 = a;$</td>
</tr>
</tbody>
</table>

Result: $r_1 = r_2 = 0$

Fig. 2. An execution allowed by coherence and PRAM but prohibited by PC.

Unfortunately, memory ordering specifications are notoriously hard to understand. It is well documented that even experts have often misunderstood classical memory models. As revealed in [Ahmad et al. 1993], for instance, contradictory claims have been made regarding the requirements of processor consistency even though it has been widely cited in the literature. The problem is exacerbated by the increasing variety and complexity of new designs proposed for modern computer systems. For example, Java is the first widely deployed programming language that provides built-in threading support at the language level. However, the existing Java memory model (JMM) is flawed and hard to understand [Pugh 2000] due to the lack of rigor in its specification. We propose a methodology to minimize the chances of such misunderstanding by allowing specifications to be written axiomatically, organized compositionally (comprised of simpler ordering rules), and by supporting direct execution.

Memory model specifications typically fall into two categories: operational or axiomatic (also known as non-operational). An operational specification often forces the reader to understand memory ordering rules in terms of the behaviors of specific data structures. Also an operational specification is a "monolith" that cannot be easily decomposed into its constituent rules. In our experience, and as cited in [Itanium Application Note 2002], non-operational descriptions can allow one to quickly grasp the detailed properties of a design. Therefore, we prefer to use axiomatic memory model specifications written in predicate logic.

The central problem we address in this paper is that most traditional axiomatic specifications are non-executable. The problem gets worse if one wants to exhaustively analyze a whole class of executions allowed by a memory system. For example, consider the validation of a litmus test (a concrete execution) as in Figure 2. This litmus test is 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? To reason about this execution, one has to rely on a hand-proof to argue that the

\[1\] The definition of processor consistency used for the DASH system [Gharachorloo et al. 1998] is quite different from Goodman’s original proposal [Goodman 1989]. We use Goodman’s version in this paper.
two operations $c = 0$ and $c = 2$ cannot be ordered in a consistent way when explaining PRAM and coherence at the same time. While a test program like this is very useful for clarifying subtle implications, hand-proving multiprocessor program behaviors is impractical even with a few memory instructions running on a few processors. Therefore, it is greatly desired that automated analysis be supported. Our use of predicate logic is motivated by its ability to offer succinct and clear expressions. However, the traditional reasoning method employed vis-a-vis predicate logic – namely theorem proving – is too expensive (in terms of human expertise and manual guidance required) for our purposes. This is where our main innovation lies: we enable automatic analysis based on constraint logic programming and/or boolean satisfiability (SAT).

To reiterate, our key contributions are as follows. (i) We introduce a generic specification method that applies predicate logic to define memory models. (ii) We propose two techniques to make these axiomatic specifications executable: one uses Prolog backtracking search, augmented with finite-domain constraint solving, and the other targets the powerful SAT solvers that have recently emerged. To the best of our knowledge, this is the first application of constraint/SAT solving for analyzing memory model compliance. (iii) We formalize a collection of memory models using our approach, including both classical models and a state-of-the-art architectural design, and show how comparative analysis can be conducted in an automated fashion. As far as we know, such a framework that practising engineers can use\(^2\) has not been proposed before.

In the next section, we provide an overview of our framework. Section 3 describes our specification method and formalizes several well known memory models. We define the Itanium memory ordering rules in Section 4. It is followed by a discussion of how to apply the framework for program verification in Section 5. Section 6 compares our approach with related work. Section 7 concludes the paper. Formal specifications of five classical memory models as well as the Itanium memory model are provided in Appendix A and Appendix B, respectively. Sample Prolog code for sequential consistency is provided in Appendix C. Additional information is available at http://www.cs.utah.edu/formal_verification/Nemos.

2 Overview of the Framework

Our framework is called Nemos (Non-operational yet Executable Memory Ordering Specifications). Nemos defines a memory model in a declarative style using a collection of ordering rules. The processing flow of our verification methodology is shown in Figure 3, 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 problem to a constraint satisfaction problem, and (iii) employing a suitable tool to solve the constraint problem automatically.

\(^2\) We have demonstrated our approach at industrial sites, with their engineers being able to use our prototype tool to exercise their designs.
2.1 Specifying the Constraints

We use predicate logic to specify the ordering constraints imposed on an ordering relation order. This approach mirrors the style adopted in modern declarative specifications written by the industry (e.g. [Itanium Application Note 2002]). To make our specifications compositional and executable, our notation differs from previous formalisms in two ways. (i) We employ a modest extension of predicate logic to 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 conveniently added. This allows us to construct a complex model using simpler components. (ii) Our 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.

2.2 Solving the Constraints

Now that the shared memory properties are formalized as machine-recognizable constraints, they can be applied to a finite program execution. This process converts the system requirements from higher order logic to propositional logic.

The Algorithm: Given a test program \( P \), we first derive its execution \( \text{ops} \) from the program text in a preprocessing phase. The initial execution is fully symbolic, that is, \( \text{ops} \) may contain free variables, e.g., for data values and ordering relations. Suppose \( \text{ops} \) has \( n \) operations, there are \( n^2 \) ordering pairs among these operations. We construct a \( n \times n \) adjacency matrix \( M \), where the element \( M_{ij} \) 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 \( M \). The goal is to find a binding of the free variables in \( \text{ops} \) such that it satisfies the conjunction of all requirements or to conclude that no such binding exists. This is automated using a constraint solver.

Two techniques have been explored to implement the algorithm: one applies a constraint solver from FD-Prolog\(^3\) and the other exploits a SAT solver.

\[^3\] FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.
Applying Constraint Logic Programming: Logic programming differs 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 axiomatic specification style. As a result, our formal specifications can be easily encoded using Prolog. 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 [Jaffar and Lassez 1987] for FD 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 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.

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. A slight variant of the Prolog code can let us benefit from SAT solving techniques. Instead of solving constraints using a FD solver, we can let Prolog emit SAT instances through symbolic execution. The resultant formula is then sent to a SAT solver to find out if the litmus test is legal under the memory model.

3 Formalizing Classical Memory Models

This section explains how to specify memory models using Nemos. Examples of some mathematical definitions are elaborated to demonstrate how rigorous specifications can be lucid as well. We first describe some terminology that is used throughout this paper.

Memory Operation A memory operation \( i \) is represented by a tuple \( (Proc, Pc, Op, Var, Data, Source, Id) \), where

- \( p_i = Proc \): issuing processor \( (Proc \in P \cup \{ p_{init} \}) \)
- \( pc_i = Pc \): program counter
- \( op_i = Op \): instruction type \( (Op \in \{ \text{Read}, \text{Write} \}) \)
- \( var_i = Var \): shared variable \( (Var \in V) \)
- \( data_i = Data \): data value
- \( source_i = Source \): source for a read, represented by the ID of the fulfilling write
- \( id_i = Id \): global ID of the operation

Initial Write For each variable \( v \), there is an initial write issued by a special pro-
A Constraint-Based Approach for Specifying Memory Consistency Models

cessor \( p_{\text{init}} \) with the default value of \( v \).

**Execution** An execution, also known as an execution trace, contains all memory operations generated by a program, 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.

### 3.1 Defining Memory Consistency Properties

Since Nemos composes a memory model as a set of ordering rules, a modular definition is naturally supported, where common axioms can be easily shared and reused. Hence, it is possible to develop a “library” of memory consistency properties. This is illustrated by Appendix A.1, where a collection of common memory requirements are formally defined.

**General Ordering Rules (Appendix A.1.1)** As mentioned earlier, general ordering rules are often implicitly required by previous models. In contrast, we capture these key requirements mathematically. As a concrete example, consider the formal definition of \texttt{requireWeakTotalOrder}, which takes \texttt{order} as a parameter and refines its constraint by asserting that every two different operations must be ordered in some way.

\[
\texttt{requireWeakTotalOrder}\ ops\ order \equiv \forall i,j \in \texttt{ops}. \quad \texttt{id} \ i \neq \texttt{id} \ j \Rightarrow (\texttt{order} \ ij \lor \texttt{order} \ ji)
\]

Translating a formal specification to Prolog is fairly straightforward. However, most Prolog systems do not directly support quantifiers. While existential quantification can be realized via Prolog’s backtracking mechanism, we need to implement universal quantification by enumerating the related finite domain (see Appendix C for more details). For instance, \texttt{requireWeakTotalOrder} is encoded as follows, where \texttt{forEachElement} is recursively defined to call \texttt{elementProgram} for every element in the adjacency matrix \( \texttt{Order} \).

\[
\begin{align*}
\texttt{requireWeakTotalOrder}(\texttt{Ops}, \texttt{Order}) :&= \\
&\texttt{length}(\texttt{Ops}, N), \\
&\texttt{forEachElement}(\texttt{Order}, N, \texttt{doWeakTotalOrder}).
\end{align*}
\]

\[
\begin{align*}
\texttt{elementProgram}(\texttt{doWeakTotalOrder}, \texttt{Order}, N, I, J) :&= \\
&\texttt{matrix_elem}(\texttt{Order}, N, I, J, Oij), \\
&\texttt{matrix_elem}(\texttt{Order}, N, J, I, Oji), \\
&(I \neq J \Rightarrow Oij \lor Oji).
\end{align*}
\]

Barring some implementation details, one technique shown by the above example

---

4 In Prolog, variable names start with a capital letter.
is worth noting. That is, the adjacency matrix \textit{Order} is passed in as a finite domain variable. The domain of the elements in \textit{Order} (which is \textit{boolean} in this case) is previously setup in the top level predicate. Providing such domain information significantly reduces the solving time. The searching order among the constraints may also impact performance. In general, it is advantageous to let the solver satisfy the most restrictive goal first.

\textbf{Read Value Rule (Appendix A.1.2)} Memory consistency is essentially defined by observable read values. This is generically captured by \texttt{requireReadValue}. Intuitively, it requires that the value observed by a read must be provided by the “latest” write on the same variable. The constraints imposed on \textit{order} precisely defines how the latest write can be determined.

\textbf{Serialization (Appendix A.1.3)} We define the notion of \textit{serialization}, a common requirement in memory model definitions, in predicate \texttt{requireSerialization}. It requires a circuit-free weak total order among a set of memory operations such that the \textit{read value rule} is also respected.

\textbf{Ordering Relations (Appendix A.1.4)} Ordering relations among memory operations can be induced under certain conditions. Predicate \texttt{requireProgramOrder} defines the condition of program order. Predicate \texttt{requireWriteIntoOrder} establishes an order between a write and a read according to data flow, which is needed to define causal consistency.

\textbf{Auxiliary Predicates (Appendix A.1.5)} 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 \texttt{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 \texttt{mapConstraints}, \textit{order}$_1$ and \textit{order}$_2$ are the respective adjacency matrices of \textit{ops}$_1$ and \textit{ops}$_2$, with \textit{ops}$_2$ being a subset of \textit{ops}$_1$.

\section*{3.2 Defining Five Classical Memory Models}

Appendix A.2 provides formal definitions of five classical memory models based on the primitive ordering properties defined in Appendix A.1.

\textbf{Sequential Consistency (Appendix A.2.1)} Sequential consistency requires a common total order among all operations, in which program order is also respected.

\textbf{Coherence (Appendix A.2.2)} For each variable, coherence requires a serialization among all memory operations involving that variable.

\textbf{PRAM (Appendix A.2.3)} 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.

**Causal Consistency (Appendix A.2.4)** In causal consistency, two operations are ordered if (i) they follow program order; (ii) one operation observes the value provided 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.

**Processor Consistency (Appendix A.2.5)** Our specification of Goodman’s processor consistency is based on the interpretation from [Ahamad et al. 1993]. As in PRAM, each processor must observe an individual serialization (as captured by order\( 2 \)). 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 order\( 1 \)). 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 \texttt{mapConstraints}.

### 4 Formalizing the Intel Itanium Memory Ordering Rules

The original Itanium memory model is informally given in various places in the Itanium architecture manual [Itanium Manual 2000]. Intel later provided an application note [Itanium Application Note 2002] to guide system developers. This document uses a combination of English and informal mathematics to specify a core subset of memory operations in a non-operational style. We demonstrate how to adapt the rules outlined in the application note to our framework in order to enable computer aided analysis. We assume proper address alignment and common address size for all memory accesses, which would be the common case encountered by programmers (even these restrictions could be easily lifted). The detailed definition of the Itanium memory model is presented in Appendix B. This section explains each of the rules.

We extend some definitions for our formal presentation:

**Instructions** - Instructions with memory access or memory ordering semantics. In this paper, 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 \( i = \text{true} \)) or write semantics (isWr \( i = \text{true} \)). 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 finer specification of the ordering properties.

**Operation Tuple** - The operation tuple is extended to describe memory operations allowed by the Itanium architecture. A memory operation \( i \) is represented by a tuple \( \langle \text{P,Pc}, \text{Op,Var,Data,WrId,WrType,WrProc,Reg,UseReg,Id} \rangle \), where
A read instruction or a fence instruction is decomposed into a single operation. A write instruction is decomposed into multiple operations, comprising a local write operation (\(\text{wrType}_i = \text{Local}\)) and a set of remote write operations (\(\text{wrType}_i = \text{Remote}\)) for each target processor (\(\text{wrProc}_i\)), which also includes the issuing processor. Every write operation \(i\) that originates from a single write instruction shares the same program counter (\(\text{pc}_i\)) and write ID (\(\text{WrID}_i\)).

**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.

### 4.1 The Itanium Memory Ordering Rules

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

\[
\text{legal} \ 	ext{ops} \equiv \exists \ \text{order}.
\]

\[
\text{requireLinearOrder} \ \text{ops order} \land
\text{requireWriteOperationOrder} \ \text{ops order} \land
\text{requirePO} \ \text{ops order} \land
\text{requireMemoryDataDependence} \ \text{ops order} \land
\text{requireDataFlowDependence} \ \text{ops order} \land
\text{requireCoherence} \ \text{ops order} \land
\text{requireReadValue} \ \text{ops order} \land
\text{requireAtomicWBRelease} \ \text{ops order} \land
\text{requireSequentialUC} \ \text{ops order} \land
\text{requireNoUCBypass} \ \text{ops order}
\]

Table 1 illustrates the hierarchy of the Itanium memory model definition. Most constraints strictly follow the rules from [Itanium Application Note 2002]. We also explicitly add a predicate \text{requireLinearOrder} to capture the general ordering requirement since [Itanium Application Note 2002] has only English to convey this
A Constraint-Based Approach for Specifying Memory Consistency Models

<table>
<thead>
<tr>
<th>requireLinearOrder</th>
<th>requireMemoryDataDependence</th>
<th>requireReadValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>- requireWeakTotal</td>
<td>- MD:RAW</td>
<td>- validWr</td>
</tr>
<tr>
<td>- requireTransitive</td>
<td>- MD:WAR</td>
<td>- validLocalWr</td>
</tr>
<tr>
<td>- requireAsymmetric</td>
<td>- MD:WAW</td>
<td>- validRemoteWr</td>
</tr>
<tr>
<td>requireWriteOperationOrder</td>
<td>requireDataFlowDependence</td>
<td>requireNoUCBypass</td>
</tr>
<tr>
<td>- local/remote case</td>
<td>- DF:RAR</td>
<td>requireSequentialUC</td>
</tr>
<tr>
<td>- remote/remote case</td>
<td>- DF:RAW</td>
<td>- RAR case</td>
</tr>
<tr>
<td>requirePO</td>
<td>- DF:WAR</td>
<td>- RAW case</td>
</tr>
<tr>
<td>- acquire case</td>
<td>requireCoherence</td>
<td>- WAR case</td>
</tr>
<tr>
<td>- release case</td>
<td>- local/local case</td>
<td>- WAW case</td>
</tr>
<tr>
<td>- fence case</td>
<td>requireAtomicWBRelease</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. The specification hierarchy of the Itanium memory ordering rules.

important ordering property.

General Ordering Requirement (Appendix B.1) This requires order to be a weak total order which is also circuit-free.

Write Operation Order (Appendix B.2) This specifies the ordering among write operations that originate from a single write instruction. It guarantees that no write can become visible remotely before it becomes visible locally.

Program Order (Appendix B.3) This restricts reordering among instructions of the same processor with respect to the program order.

Memory-Data Dependence (Appendix B.4) This restricts reordering among instructions from the same processor when they access common locations.

Data-Flow Dependence (Appendix B.5) This is intended to specify how local data dependency and control dependency should be treated. However, this is an area that is not fully specified in [Itanium Application Note 2002]. Instead of pointing to an informal document as done in [2], we provide a formal specification covering most cases of data dependency, namely establishing data dependency between two memory operations by checking the conflict usages of local registers.5

Coherence (Appendix B.6) This 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

5 We do not cover branch instructions or indirect-mode instructions that also induce data dependency. We provide enough data dependency specification to let designers experiment with straight-line code that uses registers - this is an important requirement to support execution.
Fig. 4. The operations constituting the execution of the litmus test in Fig. 2.

that order.

Read Value (Appendix B.7) This defines what data 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.

Total Ordering 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.

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.

No UC Bypassing (Appendix B.10) This specifies that uncacheable(UC) memory does not allow local bypassing from UC writes.

5 Validating Concurrent Programs

A tool named NemosFinder has been developed in SICStus Prolog [SICStus Prolog site] to enable memory model analysis. The current prototype supports the memory models defined in the Appendix. 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 till it can find an instantiation that satisfies all constraints.

Recall the test program discussed earlier in Figure 2. Its execution trace is displayed in Figure 4. When running under coherence, NemosFinder quickly concludes that the execution is legal, with an output displaying possible adjacency matrices and interleavings shown in Figure 5. A value of 1 for element $M_{ij}$ 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 6. 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
A Constraint-Based Approach for Specifying Memory Consistency Models

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.

5.1 The SAT Approach

As an alternative method, we can convert all memory model constraints to a boolean formula and apply a SAT solver to determine the result. Currently sequential consistency and the Itanium Memory Model have been implemented to support this approach. Our prototype uses the Prolog program as a driver to emit propositional formulae through symbolic execution. After being converted to the DIMACS format, the final formula is sent to a SAT solver, such as ZChaff [Moskewicz et al. 2001] or berkmin [Goldberg and Novikov 2002]. Although the clause generation phase can be detached from the logic programming approach, the ability to have it coexist with FD-Prolog might be advantageous since it allows the two methods to share the same specification base. With the tremendous improvement in SAT solving techniques, this approach offers a promising direction for enhancing scalability.

5.2 Performance Statistics

Although satisfiability problems are NP-complete, the performance in practice has been very good with the support of efficient solvers. The following table summarizes
the results for the test program in Figure 2. Performance is measured on a Pentium 366 MHz PC with 128 MB of RAM running Windows 2000. SICStus Prolog is run under compiled mode.

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>SC</th>
<th>Coherence</th>
<th>PRAM</th>
<th>Causal</th>
<th>Consistency</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>illegal</td>
<td>legal</td>
<td>legal</td>
<td>legal</td>
<td>illegal</td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>180</td>
<td>30</td>
<td>240</td>
<td>390</td>
<td>280</td>
<td></td>
</tr>
</tbody>
</table>

The following table illustrate the performance statistics for exercising the Itanium memory model. These tests are chosen from (Itanium Application Note 2002) and represented by their original table numbers. 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.

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>FD Solver(sec)</th>
<th>Vars</th>
<th>Clauses</th>
<th>SAT(sec)</th>
<th>CNF Gen (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5</td>
<td>illegal</td>
<td>0.49</td>
<td>64</td>
<td>679</td>
<td>0.01</td>
<td>3.67</td>
</tr>
<tr>
<td>Table 10</td>
<td>legal</td>
<td>3.00</td>
<td>100</td>
<td>1280</td>
<td>0.01</td>
<td>8.23</td>
</tr>
<tr>
<td>Table 15</td>
<td>illegal</td>
<td>22.29</td>
<td>576</td>
<td>15706</td>
<td>0.01</td>
<td>211.76</td>
</tr>
<tr>
<td>Table 18</td>
<td>illegal</td>
<td>2.40</td>
<td>144</td>
<td>2125</td>
<td>0.01</td>
<td>15.75</td>
</tr>
<tr>
<td>Table 19</td>
<td>legal</td>
<td>4.81</td>
<td>144</td>
<td>2044</td>
<td>0.01</td>
<td>15.68</td>
</tr>
</tbody>
</table>

For the method using SAT solvers, the clause generation time is noticeably larger than the actual SAT solving time, since the entire formula is encoded at once through symbolic execution and is recursively simplified afterwards. Alternative boolean formula encoding techniques, such as the one discussed in (Seshia et al. 2003), may help speed up this process.

6 Related Work

Formalizing memory consistency models has been a topic of extensive research in the past decade. Collier [Collier 1992] developed a formal theory of memory ordering rules. Using methods similar to Collier’s, Gharachorloo [Gharachorloo 1995] described a generic framework for specifying the implementation conditions of different memory models. Adve [Adve and Hill 1993] proposed a methodology to categorize memory models as various data-race-free classes. Mosberger [Mosberger 1993] surveyed and classified several common models. Kohli et al. [Kohli et al. 1993] proposed a formalism for capturing memory consistency rules and identified parameters that could be varied to produce new models. Raynal et al. [Raynal and Schiper 1996] provided formal definitions for the underlying consistency models for Distributed Shared Memory (DSM) systems. Bernabu-Aubn et al. [Bernabu-Aubn and Cholvi-Juan 1994] and Higham et al. [Higham et al. 1997] proposed different specification frameworks to facilitate memory model comparison. Magalhes et al. [Magalhes and de Melo 1999] formalized a set of memory models, including several hybrid ones. Although these previous efforts have tremendously enhanced the clarity of many memory models, they all suffer from the problem mentioned earlier: it
is hard to experiment with these specifications since they are passive objects and do not support automated verification.

Steinke et al. [Steinke and Nutt 2001] developed a framework that captures the relationship among several existing models based on the idea of orthogonal consistency properties. Although theoretically elegant, their work does not provide direct support for program analysis through execution. In our experience, without the ability to execute specifications, serious bugs can lurk in them, which are hard for human minds to discover. The ability to carry out exhaustive execution based analysis, albeit on small finite executions, greatly helps debug a specification.

Melo et al. [Melo and Chagas 1999] described a visual tool for displaying legal execution histories for SC and PRAM. Permitted executions are selected from a tree resulted 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. In contrast, our method allows us to 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+ [TLA+ site] [Joshi et al. 2003]. These specifications build visibility orders inductively and support the execution of litmus tests. While their approach also precisely specifies the ordering requirement, the manner in which such inductive definitions are constructed will vary from memory model to memory model, making comparisons among them harder. Our method instead relies on primitive relations and directly describes the components to make up a full memory model. This makes our specification easier to understand, and more importantly, to compare against other memory models. This also means we can enable or disable some ordering rules quite reliably without affecting the other primitive ordering rules – a danger in a style which merges all the ordering concerns in a monolithic manner.

Applying formal methods for memory model analysis has been pursued for operational specifications. Park and Dill [Park and Dill 1999] proposed a method for model checking the specifications of the Sparc memory architectures. This approach has been extended to language level memory models, such as the Java Memory Model, for verifying common programming idioms [Yang et al. 2001] [Roychoudhury and Mitra 2002] [Yang et al. 2002]. In our previous work [Yang et al. 2003a], we have developed the UMM (Uniform Memory Model) framework, a generic specification system for operational memory models that provides built-in model checking capability. This paper, on the other hand, is intended to develop verification techniques for non-operational memory models. In [Yang et al. 2003b], we extended Nemos to specify conditional branch instructions and proposed a constraint-based approach for enabling rigorous program analyses such as race detection and atom-
7 Conclusions

The setting in which contemporary memory models are expressed and analyzed needs to be improved. Towards this, we present a novel approach that captures memory model semantics in a declarative constraint-based framework. We showed that it is beneficial to encode the memory consistency requirements as constraint logic programs or, by an extra level of translation, as boolean satisfiability problems. Our 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 our framework.

Our methodology offers several unique benefits to designers and programmers.

- 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. With our framework, one can avoid the error-prone paper-and-pencil approach when reasoning about program behaviors. An executable specification can even 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 specification.

- 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. Since memory ordering rules are isolated as “facets”, one can analyze a model piece by piece. The modular approach also makes the Nemos framework scalable, a requirement for defining complex industrial models.

- Nemos provides a flexible and uniform notation that can be used to cover a wide collection of memory models, which makes comparative analysis easier. Even though this paper is by no means an effort to comprehensively cover existing proposals, other models, e.g., the hybrid models defined in [Magalhes and de Melo 1999], can be adapted easily.

There are many exciting directions for future work. As a proof-of-concept, we have encoded the memory models in the HOL theorem prover [Gordon and Melham 1993]. Such rigorous specifications will allow us to prove generic shared memory properties using theorem proving. To minimize the gap between the formal specifications and the tools that execute them, we plan to apply a Quantified Boolean Formulae (QBF) solver that directly accepts formulae with quantifiers. Also, the structural information of the ordering constraints can potentially be exploited for developing more efficient SAT solving algorithms. Last but not least, a machine-recognizable memory model enables precise semantic analysis for multiprocessor programs. For example, it is possible to formulate important problems such as race conditions as constraint satisfaction problems, which can be exhaustively and automatically investigated.
References


Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom


Appendix A: Formal Specifications of Classical Memory Models

A.1 Common Memory Consistency Properties

A.1.1 General Ordering Rules

**requireWeakTotalOrder** \( \text{ops order} \equiv \forall i, j \in \text{ops} \cdot \text{id} i \neq \text{id} j \Rightarrow (\text{order} i j \lor \text{order} j i) \)

**requireTransitiveOrder** \( \text{ops order} \equiv \forall i, j, k \in \text{ops} \cdot (\text{order} i j \land \text{order} j k) \Rightarrow \text{order} i k \)

**requireAsymmetricOrder** \( \text{ops order} \equiv \forall i, j \in \text{ops} \cdot \text{order} i j \Rightarrow \neg (\text{order} j i) \)

A.1.2 Read Value Rule

**requireReadValue** \( \text{ops order} \equiv \forall k \in \text{ops} \cdot \text{op} k = \text{Read} \Rightarrow (\exists i \in \text{ops} \cdot \text{op} i = \text{Write} \land \text{var} i = \text{var} k \land \text{data} k = \text{data} i \land \text{source} k = \text{id} i \land \neg (\text{order} k i) \land (\forall j \in \text{ops} \cdot \neg (\text{op} j = \text{Write} \land \text{var} j = \text{var} k \land \text{order} i j \land \text{order} j k))) \)

A.1.3 Serialization

**requireSerialization** \( \text{ops order} \equiv \)

\( \neg (\text{requireWeakTotalOrder ops order}) \land \neg (\text{requireTransitiveOrder ops order}) \land \neg (\text{requireAsymmetricOrder ops order}) \land \neg (\text{requireReadValue ops order}) \)

A.1.4 Ordering Relations

**requireProgramOrder** \( \text{ops order} \equiv \forall i, j \in \text{ops} \cdot ((p i = p j \land pc i < pc j) \lor (p i = p_{init} \land p j \neq p_{init})) \Rightarrow \text{order} i j \)

**requireWriteIntoOrder** \( \text{ops order} \equiv \forall i, j \in \text{ops} \cdot (\text{op} i = \text{Write} \land \text{op} j = \text{Read} \land \text{data} j = \text{data} i \land \text{source} j = \text{id} i) \Rightarrow \text{order} i j \)

A.1.5 Auxiliary Predicates

**restrictVar** \( \text{ops v} \equiv \{i \in \text{ops} \mid \text{var} i = v\} \)

**restrictVarWr** \( \text{ops v} \equiv \{i \in \text{ops} \mid \text{var} i = v \land \text{op} i = \text{Write}\} \)

**restrictProc** \( \text{ops proc} \equiv \{i \in \text{ops} \mid p i = \text{proc} \lor (p i \neq \text{proc} \land \text{op} i \neq \text{Read})\} \)

**mapConstraints** \( \text{ops1 order1 ops2 order2} \equiv \forall i, j \in \text{ops1} \cdot (i \in \text{ops2} \land j \in \text{ops2}) \Rightarrow (\text{order1} i j = \text{order2} i j) \)
A.2 Five Well Known Memory Models

A.2.1 Sequential Consistency

\textit{legal ops} \equiv \exists \text{ order} \cdot \\
requireProgramOrder \text{ ops order} \land \\
requireSerialization \text{ ops order} \\

A.2.2 Coherence

\textit{legal ops} \equiv \forall v \in V \cdot (\exists \text{ order} \cdot \\
requireProgramOrder(\text{ restrictVar ops v}) \text{ order} \land \\
requireSerialization(\text{ restrictVar ops v}) \text{ order} ) \\

A.2.3 PRAM

\textit{legal ops} \equiv \forall \text{ proc} \in P \cdot (\exists \text{ order} \cdot \\
requireProgramOrder(\text{ restrictProc ops proc}) \text{ order} \land \\
requireSerialization(\text{ restrictProc ops proc}) \text{ order} ) \\

A.2.4 Causal Consistency

\textit{legal ops} \equiv \forall \text{ proc} \in P \cdot (\exists \text{ order} \cdot \\
requireProgramOrder \text{ ops order} \land \\
requireWriteIntoOrder \text{ ops order} \land \\
requireTransitiveOrder \text{ ops order} \land \\
requireReadValue(\text{ restrictProc ops proc}) \text{ order} \land \\
requireWeakTotalOrder(\text{ restrictProc ops proc}) \text{ order} \land \\
requireAsymmetricOrder(\text{ restrictProc ops proc}) \text{ order} ) \\

A.2.5 Processor Consistency

\textit{legal ops} \equiv \exists \text{ order1} \cdot \\
(\forall v \in V \cdot \text{ requireWeakTotalOrder( restrictVarWr ops v) order1}) \land \\
(\forall \text{ proc} \in P \cdot \exists \text{ order2} \cdot \\
requireProgramOrder(\text{ restrictProc ops proc}) \text{ order2} \land \\
requireSerialization(\text{ restrictProc ops proc}) \text{ order2} \land \\
mapConstraints \text{ ops order1}(\text{ restrictProc ops proc}) \text{ order2} )
Appendix B: Formal Itanium Memory Ordering Rules

legal ops ≡ ∃ order.
requireLinearOrder ops order ∧ requireWriteOperationOrder ops order ∧ requirePO ops order ∧ requireMemoryDataDependence ops order ∧ requireDataFlowDependence ops order ∧ requireCoherence ops order ∧ requireReadValue ops order ∧ requireAtomicWBRelease ops order ∧ requireSequentialUC ops order ∧ requireNoUCBypass ops order

B.1 General Ordering Requirement
requireLinearOrder ops order ≡ requireWeakTotal ops order ∧ requireTransitive ops order ∧ requireAsymmetric ops order
requireWeakTotal ops order ≡ ∀ i, j ∈ ops · id i ≠ id j ⇒ (order i j ∨ order j i)
requireTransitive ops order ≡ ∀ i, j, k ∈ ops · (order i j ∧ order j k) ⇒ order i k
requireAsymmetric ops order ≡ ∀ i, j ∈ ops · order i j ⇒ ¬(order j i)

B.2 Write Operation Order
requireWriteOperationOrder ops order ≡ ∀ i, j ∈ ops · orderedByWriteOperation i j ⇒ order i j
orderedByWriteOperation i j ≡ isWr i ∧ isWr j ∧ wrID i = wrID j ∧ (wrType i = Local ∧ wrType j = Remote ∧ wrProc j = p i ∨ wrType i = Remote ∧ wrType j = Remote ∧ wrProc i = p i ∧ wrProc j ≠ p i)

B.3 Program Order
requirePO ops order ≡ ∀ i, j ∈ ops · (orderedByAcquire i j ∨ orderedByRelease i j ∨ orderedByFence i j) ⇒ order i j
orderedByProgram i j ≡ p i = p j ∧ pc i < pc j
orderedByAcquire i j ≡ orderedByProgram i j ∧ op i = ld · acq
orderedByRelease i j ≡ orderedByProgram i j ∧ op j = st · rel ∧ (isWr i ⇒ (wrType i = Local ∧ wrType j = Local ∨ wrType i = Remote ∧ wrType j = Remote ∧ wrProc i = wrProc j))
orderedByFence i j ≡ orderedByProgram i j ∧ (op i = mf ∨ op j = mf)
B.4 Memory-Data Dependence

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

\text{orderedByRAW } i \ j \lor \text{orderedByWAR } i \ j \lor \text{orderedByWAW } i \ j \Rightarrow
\text{order } i \ j

\text{orderedByMemoryData } i \ j \equiv \text{orderedByProgram } i \ j \land \text{var } i = \text{var } j

\text{orderedByRAW } i \ j \equiv
\text{orderedByMemoryData } i \ j \land \text{isWr } i \land \text{wrType } i = \text{Local} \land \text{isRd } j

\text{orderedByWAR } i \ j \equiv
\text{orderedByMemoryData } i \ j \land \text{isRd } i \land \text{isWr } j \land \text{wrType } j = \text{Local}

\text{orderedByWAW } i \ j \equiv \text{orderedByMemoryData } i \ j \land \text{isWr } i \land \text{isWr } j \land
\text{wrType } i = \text{Local} \land \text{wrType } j = \text{Local} \lor
\text{wrProc } i = p \ i \land \text{wrProc } j = p \ j

B.5 Data-Flow Dependence

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

\text{orderedByLocalDepenence } i \ j \Rightarrow \text{order } i \ j

\text{orderedByLocalDepenence } i \ j \equiv \text{orderedByProgram } i \ j \land \text{reg } i = \text{reg } j \land
\text{isWr } i \land \text{wrType } i = \text{Local} \land \text{useReg } i \land \text{isRd } j \lor
\text{isRd } i \land \text{isWr } j \land \text{wrType } j = \text{Local} \land \text{useReg } j

B.6 Coherence

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

\text{(isWr } i \land \text{isWr } j \land \text{var } i = \text{var } j \land
\text{attribute } (\text{var } i) = \text{WB} \lor \text{attribute } (\text{var } i) = \text{UC} \land
\text{wrType } i = \text{Local} \land \text{wrType } j = \text{Local} \land p \ i = p \ j \lor
\text{wrType } i = \text{Remote} \land \text{wrType } j = \text{Remote} \land \text{wrProc } i = \text{wrProc } j \land \text{order } i \ j)
\Rightarrow

\forall p, q \in \text{ops}.

\text{(isWr } p \land \text{isWr } q \land \text{wrID } p = \text{wrID } i \land \text{wrID } q = \text{wrID } j \land
\text{wrType } p = \text{Remote} \land \text{wrType } q = \text{Remote} \land \text{wrProc } p = \text{wrProc } q) \Rightarrow
\text{order } p \ q)

B.7 Read Value

requireReadValue ops order \equiv \forall j \in \text{ops}.

\text{(isRd } j \Rightarrow (\text{validLocalWr } \text{ops order } j \lor \text{validRemoteWr } \text{ops order } j \lor
\text{validDefaultWr } \text{ops order } j)) \land ((\text{isWr } j \land \text{useReg } j) \Rightarrow \text{validRd } \text{ops order } j)

\text{validLocalWr } \text{ops order } j \equiv \exists i \in \text{ops}.

\text{(isWr } i \land \text{wrType } i = \text{Local} \land \text{var } i = \text{var } j \land p \ i = p \ j \land
\text{data } i = \text{data } j \land \text{order } i \ j) \land
A Constraint-Based Approach for Specifying Memory Consistency Models

$$\neg \exists k \in \text{ops} \cdot \text{isWr } k \land \text{wrType } k = \text{Local} \land \text{var } k = \text{var } j \land \text{p } k = \text{p } j \land \text{order } i \land \text{order } k$$

validRemoteWr ops order j \equiv \exists i \in \text{ops}.

$$\text{isWr } i \land \text{wrType } i = \text{Remote} \land \text{wrProc } i = \text{p } j \land \text{var } i = \text{var } j \land \text{data } j = \text{data } i \land \neg \text{(order } j \text{ i)}) \land$$

$$\neg \exists k \in \text{ops} \cdot \text{isWr } k \land \text{wrType } k = \text{Remote} \land \text{var } k = \text{var } j \land \text{wrProc } k = \text{p } j \land \text{order } i \land \text{order } k$$

validDefaultWr ops order j \equiv

$$\neg \exists i \in \text{ops} \cdot \text{isWr } i \land \text{var } i = \text{var } j \land \text{order } i \land$$

$$(\text{wrType } i = \text{Local} \land \text{p } i = \text{p } j \lor \text{wrType } i = \text{Remote} \land \text{wrProc } i = \text{p } j)) \land \text{data } j = \text{default } (\text{var } j)$$

validRd ops order j \equiv

$$\text{isRd } i \land \text{reg } i = \text{reg } j \land \text{orderedByProgram } i \land \text{data } j = \text{data } i \land$$

$$\neg \exists k \in \text{ops} \cdot \text{isRd } k \land \text{reg } k = \text{reg } j \land$$

$$\text{orderedByProgram } i \land \text{orderedByProgram } k$$

B.8 Total Ordering of WB Releases

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

$$\text{op } i = \text{st } \cdot \text{rel} \land \text{wrType } i = \text{Remote} \land \text{op } k = \text{st } \cdot \text{rel} \land \text{wrType } k = \text{Remote} \land \text{wrID } i = \text{wrID } k \land \text{attribute } (\text{var } i) = \text{WB} \land \text{order } i \land \text{order } k \Rightarrow$$

$$\text{op } j = \text{st } \cdot \text{rel} \land \text{wrType } j = \text{Remote} \land \text{wrID } j = \text{wrID } i$$

B.9 Sequentiality of UC Operations

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

orderedByUC i \land \text{order } j \equiv

$$\text{orderedByProgram } i \land \text{attribute } (\text{var } i) = \text{UC} \land \text{attribute } (\text{var } j) = \text{UC} \land$$

$$\text{isRd } i \land \text{isRd } j \lor$$

$$\text{isRd } i \land \text{isWr } j \land \text{wrType } j = \text{Local} \lor$$

$$\text{isWr } i \land \text{wrType } i = \text{Local} \land \text{isRd } j \lor$$

$$\text{isWr } i \land \text{wrType } i = \text{Local} \land \text{isWr } j \land \text{wrType } j = \text{Local}$$

B.10 No UC Bypassing

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

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

$$\text{isWr } k \land \text{wrType } k = \text{Remote} \land \text{wrProc } k = \text{p } k \land \text{wrID } k = \text{wrID } i \land \text{order } i \land \text{order } j \land \text{order } k \Rightarrow$$

$$\text{wrProc } k \neq \text{p } j \land \text{var } i \neq \text{var } j$$
Appendix C: Excerpts from the Prolog Encoding of SC

/**---------------------------------------------------------------*/
* legal
/**---------------------------------------------------------------*/
legal(Ops):-
    length(Ops,N),
    create_matrix(N,Order),
    domain(Order,0,1),

    requireProgramOrder(Ops,Order),
    requireSerialization(Ops,Order),

    labeling([],Order),
    write_matrix(Order,N),nl,
    write_interleaving(Order,N),nl.

/**---------------------------------------------------------------*/
* requireProgramOrder
/**---------------------------------------------------------------*/
requireProgramOrder(Ops,Order):-
    foreach_elem(Ops,Order,doProgramOrder).

elemProgram(doProgramOrder,Ops,Order,I,J):-
    nth(I,Ops,Oi),
    nth(J,Ops,Oj),
    p(Oi,P_i),
    p(Oj,P_j),
    pc(Oi,PC_i),
    pc(Oj,PC_j),
    length(Ops,N),
    matrix_elem(Order,N,I,J,Oij),
    (P_i #= P_j #\ PC_i #< PC_j) #=> Oij.

/**---------------------------------------------------------------*/
* requireSerialization
/**---------------------------------------------------------------*/
requireSerialization(Ops,Order):-
    requireWeakTotalOrder(Ops,Order),
    requireTransitiveOrder(Ops,Order),
    requireAsymmetricOrder(Ops,Order),
    requireReadValue(Ops,Order).

/**---------------------------------------------------------------*/
* requireWeakTotalOrder
/**---------------------------------------------------------------*/
requireWeakTotalOrder(Ops,Order):-
    length(Ops,N),
    foreach_element(Order,N,doWeakTotalOrder).
elemProgram(doWeakTotalOrder,Order,N,I,J):-
    matrix_elem(Order,N,I,J,Oij),
    matrix_elem(Order,N,J,I,Oji),
    (I \= J #=> Oij \= Oji).

/**---------------------------------------------------------------*/
* requireAsymmetricOrder
/**---------------------------------------------------------------*/
requireAsymmetricOrder(Ops,Order):-
    length(Ops,N),
    foreach_element(Order,N,doAsymmetricOrder).
A Constraint-Based Approach for Specifying Memory Consistency Models

elementProgram(doAsymmetricOrder,Order,N,I,J):-
  matrix_elem(Order,N,I,J,Oij),
  matrix_elem(Order,N,J,I,Oji),
  (#Oij #\ #\Oji).

/*----------------------------------------------------------------
* requireTransitiveOrder
*/
requireTransitiveOrder(Ops,Order):-
  length(Ops,N),
  for_ijk(Order,N,doTransitiveOrder).

ijk_prog(doTransitiveOrder,Order,N,I,J,K):-
  matrix_elem(Order,N,I,J,Oij),
  matrix_elem(Order,N,J,K,Ojk),
  matrix_elem(Order,N,K,I,Oik),
  (Oij #\ Ojk #=> Oik).

/*----------------------------------------------------------------
* requireReadValue
*/
requireReadValue(Ops,Order):-
  for_each_op_1(Order,Ops,doReadValue).

op_prog_1(doReadValue,Order,Ops,K):-
  nth(K,Ops,R),
  const(r,Read),

  op(R,Read) --> checkPreviousWrite(Order,Ops,R); true.

/*/ 
* checkPreviousWrite 
* Checks if there exists a previous candidate write W for a read R. 
*/
checkPreviousWrite(Order,Ops,R):-
  member(W,Ops), % there exists...
  nth(I,Ops,W),
  op(W,Op_w),
  var(W,Var_w),
  id(W,ID_w),
  data(W,Data_w),

  nth(K,Ops,R),
  var(R,Var_r),
  data(R,Data_r),
  source(R,Source_r),

  length(Ops,N),
  matrix_elem(Order,N,K,I,Orw),
  const(w,Write),

  (Op_w #= Write #\ Var_w #= Var_r #\ Data_r #= Data_w #\ Source_r #= ID_w #\ Orw),
  for_each_op_2(Order,Ops,W,R,checkMostRecent).

/*/ 
* checkMostRecent 
*/
op_prog_2(checkMostRecent,Order,Ops,W,R,J):-
  nth(J,Ops,W2),
  nth(I,Ops,X),
  nth(K,Ops,R),
  op(W2,Op_w2),
var(W2,Var_w2),
var(R,Var_r),
length(Ops,N),
matrix_elem(Order,N,I,J,Oww2),
matrix_elem(Order,N,J,K,Ow2r),
const(w,Write),
#
( Op_w2 #= Write #/
  Var_w2 #= Var_r #/
  Oww2 #\ Ow2r ).

/*****************************************************************
* The following implements some auxiliary predicates. The N x N
* elements of the ordering matrix is represented as a list:
* index_ij = i*(N-1) + j, where i and j range from 1 to N.
*****************************************************************/

/*----------------------------------------------------------------
* create_matrix(N,M)
* Creates a matrix M (with unbound variables) with N lines.
*----------------------------------------------------------------------*/
create_matrix(N,M):-
  T is N*N, create_matrix1(0,T,M), !.
create_matrix1(T,T,[]).
create_matrix1(K,T,[_|M]):-
  K1 is K+1, create_matrix1(K1,T,M).

/*----------------------------------------------------------------
* matrix_elem(M,N,I,J,X)
* Gets the value of element (I,J) of matrix M (size N) into X.
*----------------------------------------------------------------------*/
matrix_elem(M,N,I,J,X):-
  K is (I-1)*N+J, nth(K,M,X).

/*----------------------------------------------------------------
* forEachElement(M,N,P)
* Calls elementProgram(P,M,N,I,J) for each element (I,J) of matrix M.
*----------------------------------------------------------------------*/
forEachElement(M,N,P):-
  length(Ops,N), T is N*N,
  for_each_elem_hlpr2(0,T,Ops,M,P), !.

for_each_elem_hlpr2(T,T,_,_,_).
for_each_elem_hlpr2(K,T,M,N,P):-
  K1 is K+1,
  I is (K // N)+1,
  J is (K rem N)+1,
  elementProgram(P,M,N,I,J),
  for_each_elem_hlpr2(K1,T,M,N,P).

/*----------------------------------------------------------------
* forEachElem(Ops,M,P)
* Calls elementProgram(P,Ops,M,I,J) for each element (I,J) of matrix M.
*----------------------------------------------------------------------*/
forEachElem(Ops,M,P):-
  length(Ops,N), T is N*N,
  for_each_elem_hlprx2(0,T,Ops,M,P), !.
for_each_elem_hlpr2(T,T,_,_,_).

for_each_elem_hlpr2(K,T,Ops,M,P):-
  length(Ops,N),
  K1 is K+1,
  I is (K // N)+1,
  J is (K rem N)+1,
  elemProgram(P,Ops,M,I,J),
  for_each_elem_hlpr2(K1,T,Ops,M,P).

/*------------------------------------------------------------------------------------------*/
for_each_op_1(Order,Ops,P):-
  length(Ops,N), for_each_op_hlpr1(N,Order,Ops,P).

for_each_op_hlpr1(0,_,_,_):-!.
for_each_op_hlpr1(I,Order,Ops,P):-
  op_prog_1(P,Order,Ops,I),
  I1 is I-1,
  for_each_op_hlpr1(I1,Order,Ops,P).

/*------------------------------------------------------------------------------------------*/
for_each_op_2(Order,Ops,W,R,P):-
  length(Ops,N), for_each_op_hlpr2(0,N,Order,Ops,W,R,P), !.

for_each_op_hlpr2(0,_,_,_,_,_).
for_each_op_hlpr2(I,T,Order,Ops,W,R,P):-
  I1 is I+1,
  op_prog_2(P,Order,Ops,W,R,I1),
  for_each_op_hlpr2(I1,T,Order,Ops,W,R,P).

/*------------------------------------------------------------------------------------------*/
for_ijk(M,N,P):-
  for_k(0,N,M,P), !.
  for_k(T,T,_,_).
for_k(K,N,M,P):-
  K1 is K + 1,
  for_ij(M,N,K1,P),
  for_k(K1,N,M,P).

for_ij(M,N,K,P):-
  T is N*N, for_ij1(0,T,M,N,K,P), !.
for_ij1(T,T,_,_,_).
for_ij1(X,T,M,N,K,P):-
  X1 is X+1,
  I is (X // N)+1,
  J is (X rem N)+1,
  ijk_prog(P,M,N,I,J,K),
  for_ij1(X1,T,M,N,K,P).