Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models*

Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind
School of Computing, University of Utah
{yyang | ganesh | gary | slind}@cs.utah.edu

Abstract

Conforming to the underlying memory consistency rules is a fundamental requirement for implementing shared memory systems and writing multiprocessor programs. In order to promote understanding and enable automated verification, it is highly desirable that a memory model specification be both declarative and executable. We have developed a specification framework called *Nemos* (Non-operational yet Executable Memory Ordering Specifications), which employs a uniform notation based on predicate logic to define shared memory semantics in an *axiomatic* as well as *compositional* style. In this paper, we present this framework and discuss how constraint logic programming and SAT solving can be used to make these axiomatic specifications executable for memory model analysis, thus supporting precise specification and automatic execution in the same framework. To illustrate our approach, this paper formalizes a collection of well known memory models, including sequential consistency, coherence, PRAM, causal consistency, and processor consistency.

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 consistency (SC) [1] 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 [2] for a survey) have been proposed to provide higher performance. For example, coherence [3]

*This work was supported in part by Research Grant No. CCR-0081406 (ITR Program) of NSF and SRC Task 1031.001.

```
P1                                  P2

flag1 = true;                       flag2 = true;
turn = 2;                           turn = 1;
while(turn == 2 && flag2)           while(turn == 1 && flag1)
    ;                                   ;
< critical section >                < critical section >
flag1 = false;                      flag2 = false;
```

Figure 1: Peterson's algorithm for mutual exclusion.

(also known as cache consistency) only enforces SC on a per-variable basis. Parallel RAM (PRAM) [4] allows each observing processor to perceive its own view of memory operation order. Causal consistency [5] follows a similar policy while enforcing the causal order resulting from data flow. Processor consistency (PC) [6] (we use Goodman's version in this paper) 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 [7], entry consistency [8], and location consistency [9].

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. Consider, for example, Peterson's algorithm [10] 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. Therefore, memory model rules can impact system correctness in a fundamental way. Unfortunately, specifications of memory ordering rules are notoriously hard to understand. For example, it is well documented in the literature that even experts have often misunderstood classical memory models [6]. The problem is exacerbated by the increasing variety and complexity of new designs proposed for modern computer systems. 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 [11], 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[1].

[1]Actually our specifications employ a mild extension falling in a limited fragment of higher order logic.

Initially, $a = b = c = 0$

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

Result: $r1 = r2 = 0$

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

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 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. (iii) We formalize a collection of well known memory models using our approach, and show how comparative analysis can be conducted in an automated fashion. To the best of our knowledge, such a framework that practising engineers can use[2], has not been proposed before.

The rest of the paper is organized as follows. In Section 2, we provide an overview of our framework. Section 3 describes our specification method. It is followed by a discussion of how to apply the framework for program verification in Section 4. Section 5 compares our approach with related work. Section 6 concludes the paper. Formal specifications of five well known memory models are provided in the Appendix.

[2]We have demonstrated our approach at industrial sites, with their engineers being able to use our prototype tool to exercise their designs.

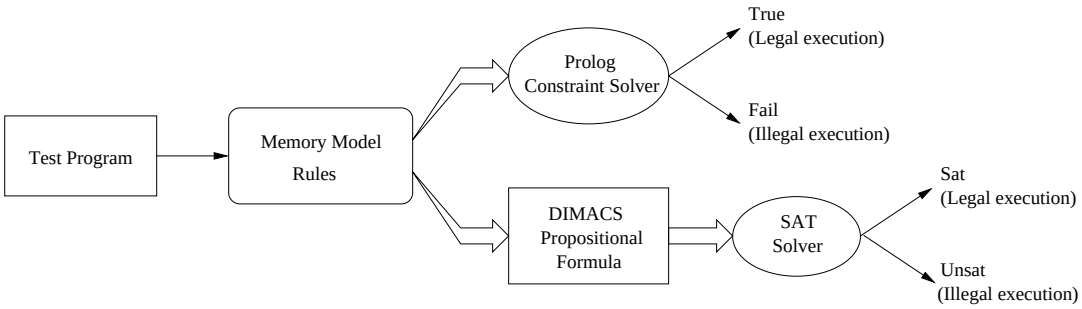

Figure 3: The process of making an axiomatic memory model executable.

2 Overview of the Framework

2.1 The Specification Method

Nemos (Non-operational yet Executable Memory Ordering Specifications) is designed as a specification framework for creating uniform executable memory model definitions in an axiomatic style. To specify a memory model, the framework defines a complete set of ordering constraints imposed on an ordering relation *order*. This approach mirrors the style adopted in modern declarative specifications written by the industry, such as [11]. Our notation differs from traditional formalisms in two ways. First, we employ a slight 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. Second, our specifications are fully explicit about all ordering properties, including previously "hidden" requirements such as totality, transitivity, and circuit-freedom, so that our constraints can completely characterize a memory model. Without explicating such hidden requirements, a specification is not complete for execution.

2.2 Making Axiomatic Specifications Executable

Now that the shared memory properties are formalized as machine-recognizable constraints, they can be *instantiated* over a finite program execution. This process converts the memory model requirements from higher order logic to propositional logic. Consequently, the legality of the execution can be automatically checked by solving a satisfiability problem.

The Algorithm: Given a finite execution *ops* with n operations, there are n^2 ordering pairs among the operations, constituting an adjacency matrix $\mathcal{M}$, where the element $\mathcal{M}_{ij}$ indicates whether operations i and j should be ordered. We go through each ordering rule in the memory model specification and impose the corresponding propositional constraints with respect to the elements of $\mathcal{M}$. Then we check the satisfiability of the conjunction of all ordering requirements. If such an $\mathcal{M}$ exists, the trace *ops* is legal, and a valid interleaving can be derived from $\mathcal{M}$. Otherwise, *ops* is not allowed by the memory model.

A pictorial representation of our methodology is shown in Figure 3. 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.

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* [12] 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.

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 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. Five commonly cited memory models are formally defined in the Appendix, with additional information available at `http://www.cs.utah.edu/formal_verification/Nemos`. We first describe some terminology that is used throughout this paper.

Memory Operation A *memory operation* `i` is represented by a tuple $\langle p, c, o, v, d, s, g \rangle$, where

proc $i = p$:	issuing processor ($p \in P \bigcup \{p_{init}\}$)
pc $i = c$:	program counter
op $i = o$:	instruction type ($o \in \{Read, Write\}$)
var $i = v$:	shared variable ($v \in V$)
data $i = d$:	data value
source $i = s$:	the source for a read, represented by the ID of the fulfilling write
id $i = g$:	global ID of the operation

[3]FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.

Initial Write For each variable v, there is an *initial write* issued by a special processor p_{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.

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 `requireWeakTotalOrder`, which takes *order* as a parameter and refines its constraint by asserting that every two different operations must be ordered in some way.

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

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. For instance, `requireWeakTotalOrder` is encoded as follows, where `forEachElement` is recursively defined to call `elementProgam` for every element in the adjacency matrix *Order*[4].

```
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).
```

Read Value Rule (Appendix A.1.2) Memory consistency is essentially defined by observable read values. This is generically captured by predicate `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 *order* precisely defines how the latest write can be determined.

[4]In Prolog, variable names start with a capital letter.

Serialization (Appendix A.1.3) We define the notion of *serialization*, a commonly used requirement in memory model definitions, 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.

Ordering Relations (Appendix A.1.4) 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.

Auxiliary Predicates (Appendix A.1.5) Sometimes the serialization requirement 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 adjacency matrices of *ops1* and *ops2*, with *ops2* being a subset of *ops1*.

3.2 Defining Memory Models

Appendix A.2 provides formal definitions of five classical memory models based on the primitive ordering properties defined in Appendix A.1. In a separate report [13], we have applied the same method to formalize the Intel Itanium Memory Model [11], demonstrating the scalability of our approach for complex industrial designs.

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

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

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 [6]. 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

```
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);
```

Figure 4: The operations constituting the execution of the litmus test in Fig. 2.

the same time in a mutually consistent manner. This critical requirement, which may be easily overlooked, is clearly spelled out by `mapConstraints`.

4 Validating Concurrent Programs

A tool named *NemosFinder* has been developed in SICStus Prolog [14] to enable memory model analysis. The current prototype supports the memory models defined in the Appendix and the Itanium Memory Model. 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 $\mathcal{M}_{ij}$ 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 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 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.

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

(a) Interleaving for a: *1 9 4*

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

(b) Interleaving for b: *2 6 7*

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

(c) Interleaving for c: *3 8 5*

Figure 5: Adjacency matrices for the execution shown in Fig. 4 under coherence.

	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

(a) Interleaving for Process 1:
3 2 1 4 5 6 7 8

	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

(b) Interleaving for Process 2:
3 2 1 7 8 9 4 5

Figure 6: Adjacency matrices for the execution shown in Fig. 4 under PRAM.

4.1 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` [15] or `berkmin` [16]. 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.

4.2 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.

Memory Model	SC	Coherence	PRAM	Causal Consistency	PC
Result	illegal	legal	legal	legal	illegal
Time (ms)	180	30	240	390	280

5 Related Work

Formalizing memory consistency models has been a topic of extensive research in the past decade. Collier [17] developed a formal theory of memory ordering rules. Using methods similar to Collier's, Gharachorloo [3] described a generic framework for specifying the implementation conditions of different memory models. Adve [18] proposed a methodology to

categorize memory models as various data-race-free classes. Mosberger [19] surveyed and classified several common models. Kohli et al. [20] proposed a formalism for capturing memory consistency rules and identified parameters that could be varied to produce new models. Raynal et al. [21] provided formal definitions for the underlying consistency models for Distributed Shared Memory (DSM) systems. Bernabu-Aubn et al. [22] and Higham et al. [23] proposed different specification frameworks to facilitate memory model comparison. Magalhes et al. [24] 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. 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. [25] 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+ [26, 27]. 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 [28] 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 [29, 30, 31]. In our previous work [32], 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. Our formalization of the Itanium Memory Model [13] demonstrated the potential payoff of our approach for industrial chip design and verification.

6 Conclusions

We have presented a formal framework for specifying and analyzing non-operational memory consistency models, which 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 [24]) 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 [33]. 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

[1] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] Kourosh Gharachorloo. Memory consistency models for shared-memory multiprocessors. Technical report, CSL-TR-95-685.

[4] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, 1988.

[5] Mustaque Ahamad, Gil Neigher, James Burns, Prince Kohli, and Philip Hutto. Causal memory: Definitions, implementation and programming. Technical report, GIT-CC-93/55, Georgia Institute of Technology, July 1994.

[6] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 93)*, 1993.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[8] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[9] Guang Gao and Vivek Sarkar. Location consistency - a new memory model and cache consistency protocol. Technical report, 16, CAPSL, University of Delaware, 1998.

[10] G. L. Peterson. Myths about the mutual exclusion problem. In *Information Processing Letters*, volume 12, June 1981.

[11] A formal specification of Intel Itanium processor family memory ordering. *Application Note, Document Number: 251429-001*, October, 2002.

[12] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

[13] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *CHARME 2003*, October 2003.

[14] Sicstus prolog. http://www.sics.se/sicstus.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *39th Design Automation Conference*, Las Vegas, June 2001.

[16] 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.

[17] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, 1992.

[18] 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.

[19] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[20] Prince Kohli, Git Neiger, and Mustaque 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.

[21] 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.

[22] Jos M. Bernabu-Aubn and Vicente Cholvi-Juan. Formalizing memory coherency models. *Journal of Computing and Information*, May 1994.

[23] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 349–356, 1997.

[24] Alba Cristina Magalhes and Alves de Melo. Defining uniform and hybrid memory consistency models on a unified framework.

[25] Alba Cristina Melo and Simone Cintra Chagas. Visual-MCM: Visualising execution histories on multiple memory consistency models. *Lecture Notes in Computer Science*, 1557:500–509, 1999.

[26] Tla+. http://research.microsoft.com/users/lamport/tla/tla.html.

[27] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking cache-coherence protocols with tla+. In *Formal Methods in System Design*, number 2, pages 125–131, March 2003.

[28] Seungjoon Park and David L. Dill. An executable specification and verifier for Relaxed Memory Order. *IEEE Transactions on Computers*, 48(2):227–235, 1999.

[29] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Analyzing the CRF Java Memory Model. In *the 8th Asia-Pacific Software Engineering Conference*, 2001.

[30] Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded Java semantics for program verification. In *International Conference on Software Engineering*, 2002.

[31] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Joint ACM Java Grande - ISCOPE Conference*, 2002.

[32] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. A generic operational memory model specification framework for multithreaded program verification. Technical report, UUCS-03-015, University of Utah, 2003.

[33] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

Appendix: Formal Memory Model Specifications

A.1 Common Memory Consistency Properties

A.1.1 General Ordering Rules

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

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

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

A.1.2 Read Value Rule

requireReadValue *ops order* $\equiv \forall k \in ops.$ **op** $k = Read\ \Rightarrow$
$\quad$ ($\exists i \in ops.$ **op** $i = Write\ \wedge$ **var** $i =$ **var** $k\ \wedge$
$\quad$ **data** $k =$ **data** $i\ \wedge$ **source** $k =$ **id** $i\ \wedge\ \neg$(**order** $k\ i$) $\wedge$
$\quad$ ($\forall j \in ops.\ \neg$(**op** $j = Write\ \wedge$ **var** $j =$ **var** $k\ \wedge$ **order** $i\ j\ \wedge$ **order** $j\ k$)))

A.1.3 Serialization

requireSerialization *ops order* $\equiv$
$\quad$ **requireWeakTotalOrder** *ops order* $\wedge$
$\quad$ **requireTransitiveOrder** *ops order* $\wedge$
$\quad$ **requireAsymmetricOrder** *ops order* $\wedge$
$\quad$ **requireReadValue** *ops order*

A.1.4 Ordering Relations

requireProgramOrder *ops order* $\equiv \forall\, i, j \in ops.$
$\quad$ ((**proc** $i =$ **proc** $j\ \wedge$ **pc** $i <$ **pc** j) $\vee$ (**proc** $i = p_{init}\ \wedge$ **proc** $j \neq p_{init}$)) $\Rightarrow$ **order** $i\ j$

requireWriteIntoOrder *ops order* $\equiv \forall\, i, j \in ops.$
$\quad$ (**op** $i = Write\ \wedge$ **op** $j = Read\ \wedge$ **data** $j =$ **data** $i\ \wedge$ **source** $j =$ **id** i) $\Rightarrow$ **order** $i\ j$

A.1.5 Auxiliary Predicates

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

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

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

mapConstraints *ops*1 *order*1 *ops*2 *order*2 $\equiv \forall\, i, j \in ops1.$
$\quad$ ($i \in ops2\ \wedge\ j \in ops2$) $\Rightarrow$ (**order1** $i\ j =$ **order2** $i\ j$)

A.2 Five Well Known Memory Models

A.2.1 Sequential Consistency

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

A.2.2 Coherence

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

A.2.3 PRAM

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

A.2.4 Causal Consistency

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

A.2.5 Processor Consistency

legal *ops* $\equiv \exists$ *order*1.
$(\forall\ v \in V.$ **requireWeakTotalOrder** (**restrictVarWr** *ops v*) *order*1$) \wedge$
$(\forall\ p \in P.\ \exists$ *order*2.
requireProgramOrder (**restrictProc** *ops p*) *order*2 $\wedge$
requireSerialization (**restrictProc** *ops p*) *order*2 $\wedge$
mapConstraints *ops order*1 (**restrictProc** *ops p*) *order*2)