# An Executable Specification and Verifier for Relaxed Memory Order

Seungjoon Park and David L. Dill, *Member, IEEE*

**Abstract**—The Mur$\varphi$ description language and verification system for finite-state concurrent systems is applied to the problem of specifying a family of multiprocessor memory models described in the SPARC Version 9 architecture manual. The description language allows for a straightforward operational description of the memory model which can be used as a specification for programmers and machine architects. The automatic verifier can be used to generate all possible outcomes of small assembly language multiprocessor programs in a given memory model, which is very helpful for understanding the subtleties of the model. The verifier can also check the correctness of assembly language programs including synchronization routines. This paper describes the memory models and their encoding in the Mur$\varphi$ description language. We describe how synchronization routines can be verified and how finite state programs can be analyzed. We also present some interesting findings from the verification and the analysis.

**Index Terms**—Multiprocessors, memory models, formal method, executable specification, automatic verification.

## 1 Introduction

In a shared memory multiprocessor architecture, a *memory model* specifies the semantics of memory operations when multiple processors load and store shared memory locations [1], [2]. The precise details of this model are crucial to several parties. Obviously, the design of the cache coherence scheme must respect the model. Also, processor designers must ensure that, for example, out-of-order issue of memory instructions conforms to the model. Programmers must be aware of the model because, for example, it affects the correctness of synchronization routines. Compiler writers may also have to consider the memory model in some optimizations.

Several memory models for shared-memory multiprocessor architecture have been proposed. An early model, sequential consistency [3], simply required that multiprocessors simulate atomic reads and writes to a common global memory. This model is relatively easy to understand but has strong constraints which hinder high performance implementations. During the past decade, a lot of effort has been made to design weaker memory models, such as processor consistency [4], [5], [6], total store ordering [7], [8], partial store ordering [7], [8], weak ordering [9], [10], release consistency [5], [6], relaxed memory order [11], Digital Equipment Alpha [12], and IBM PowerPC [13].

Weaker memory models are attractive because they allow better more concurrency in memory system and processor implementations, resulting in improved performance. However, weaker memory models are generally very subtle because understanding the behavior of highly concurrent systems is never easy. Even sequential consistency can be counter-intuitive at times [10], [14], [15]. Hence, it is vital to specify a memory model precisely.

Our approach to these problems is to describe the memory model by giving a *maximally general* executable description, using a simple general-purpose description language for concurrent systems called Mur$\varphi$ [16]. Such a description provides a precise specification of the machine architecture, both for implementors and programmers.

It is important to note that the executable description is a maximally general implementation which could be regarded as a *formal specification*. In other words, all the execution traces generated by the operational model are legal under the logical specification *and* all the legal execution traces are generated by the operational model.

The major advantage of using Mur$\varphi$ is that it is also an *automatic formal verification system*. There is a tool that supports exhaustive checking of all the reachable states of a description for deadlocks or violations of user-specified properties. Since Mur$\varphi$ can only deal with finite-state processes, various memory structures must be bounded for automatic verification. Mur$\varphi$ also allows the printing of the state of a system at user-specified points while exploring the reachable states; this feature can be used, for example, to list all of the possible register values that can occur when an example program terminates.

The approach here is different from that used by Collier [1], who infers the behavior of programs from a set of ordering relations, which are not necessarily easy to convert into an executable form. Gharachorloo et al. [6], [18] and Sindhu et al. [8] have used methods similar to Collier's. Our method more closely resembles that of Gibbons et al. [19], who give I/O automata specifications of memory models. The primary differences here are the description languages and, more importantly, our emphasis on support for automatic processing, while verification with I/O automata is generally by hand [20].

• *S. Park is with RIACS, NASA Ames Research Center, Moffett Field, CA . E-mail: park@cs.stanford.edu.*
• *D.L. Dill is with the Department of Computer Science, Stanford University, Stanford, CA . E-mail: dill@cs.stanford.edu.*

*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108227.*

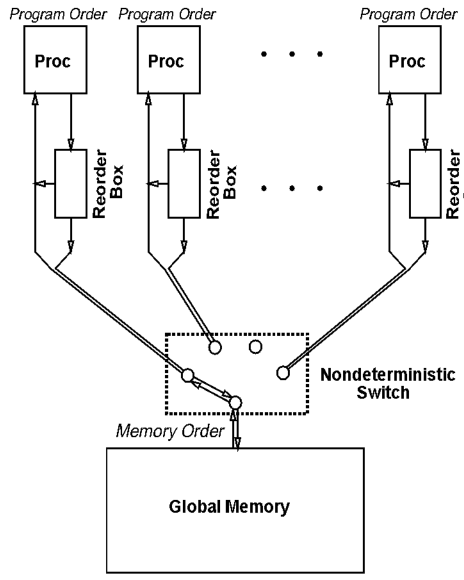

Fig. 1. An abstract memory model for multiprocessors.

We developed an executable memory model during the process of defining the Relaxed Memory Order (RMO) model of the SPARC Architecture Manual Version 9 [11]. RMO is a generalization of the previous SPARC Version 8 memory models, TSO (Total Store Ordering) and PSO (Partial Store Ordering). Intuitively, TSO liberalizes sequential ordering by allowing the performance of stores to be delayed relative to subsequent loads; PSO additionally allows stores to be delayed relative to other stores; and RMO further allows loads to be delayed relative to subsequent loads and stores. In previous work, we developed an executable model for TSO and PSO [21]; however, the executable model of RMO is not a simple generalization of the earlier description.

Developing an executable model of the protocol in Mur$\varphi$ greatly enhanced our understanding and confidence in the design for several reasons. First, writing a precise description points out ambiguities and inconsistencies, even if the description is not executed. Second, we were able to analyze the possible outcomes of illustrative examples and synchronization programs rapidly and automatically when there was a question about the implication of a change in detail of the specification. Third, we could verify the examples in the SPARC-V9 Architecture Manual, which increased our confidence that there were no errors in the code examples associated with the memory models.

## 2 Logical Specification of the Memory Model

Fig. 1 illustrates the intuition behind the SPARC memory models. Note that this is a fictitious description that bears no relation to a reasonable *implementation* of a memory model—it is only intended to capture a programmer-level view of the possible behaviors of memory operations. There is a set of processors, $P_1, P_2, ..., P_n$, each of which has its own cache, and an *abstract reorder box*. Each processor executes instructions in the natural order specified in the program, called *program order*. Instructions may appear to occur in some order other than program order, due to various implementation techniques, such as local caching or out-of-order instruction execution in the processor implementation. This reordering is modeled in Fig. 1 by attaching a *reorder box* to each processor and cache.

Each reorder box is also connected to a common global *memory*. The memory arbitrarily chooses one of the reorder boxes, chooses an instruction from the reorder box subject to ordering constraints that are specified as part of the particular memory model. It then executes actions which depend on the instruction, such as updating memory locations or processor registers. The actions for each instruction are executed atomically—other actions in the system cannot interfere with them. An instruction is said to be *performed* when it is executed by the memory.

The following is a condensed description[1] of the logical specification of the memory model in the SPARC Architecture Version 9 [11]. The logical specification is not executable. In essence, given an instruction trace from each processor consisting of the sequence of instructions and the results of interactions with the memory system, it determines whether the instruction trace is compatible with the memory model. This is the style of specification used by Collier et al. and Frailong [1], [18], [8].

In the remainder of this paper, $X, Y$, and $Z$ refer to memory instructions. $X_A^n$ denotes a memory instruction $X$ on processor $n$ that reads or writes memory address $A$. The processor index and memory address are specified only if needed. Predicates $L(X)$ and $S(X)$ are true if $X$ is a load or a store instruction, respectively. $L(Y)$ and $S(Y)$ can be true simultaneously, when $Y$ is an atomic load/store.

A *program order* is a partial order of instructions that is an interleaving of total orders, one for each processor: Instructions associated with the same processor are always program-ordered, while instructions from different processors are never program ordered. Program order represents the sequence of instructions as issued by each processor. We write $A <_p B$ when instruction $A$ precedes instruction $B$ in program order.

*Memory order* is a total order of all the memory instructions from the processors. Each memory model defines a set of *ordering rules* which constrain legal memory orders. Many memory orders may be consistent with a given program order. This multiplicity of orders reflects nondeterminism in the memory model and yields nondeterministic results when multiprocessor programs are executed. The choice of a particular global memory order determines the values returned by loads. We write $A <_m B$ when instruction $A$ precedes instruction $B$ in a particular memory order; also, in this case, we say "$A$ is performed before $B$." The SPARC-V9 architecture has a special *memory barrier* instruction (*membar*). It explicitly enforces additional constraints on the memory order of certain types of memory instructions preceding and following the *membar*. For instance, `membar{L<S}` requires that all the loads preceding the membar in program order precede the stores following it. The predicate $M(X,Y)$ is used to represent

1. The change we made from the SPARC manual is that we do not differentiate memory transactions from memory instructions. We believe that such a distinction is not necessary at this level of specification.

that $X <_p Y$ and $X$ and $Y$ are ordered by a membar of the corresponding type.

A program in a weak memory model can be made to behave like the same program in a stronger model by inserting membars between appropriate instructions. To simulate PSO under RMO, we may put `membar{L<L,L<S}` immediately following every load, disabling the freedom of RMO to delay the execution of loads until after the following memory instructions.

### 2.1 Ordering Rules

There are some times when ordering constraints from a processor must necessarily constrain the memory order. For example, an instruction loading a register cannot be performed after an instruction storing the resulting register value back to memory. The SPARC-V9 memory model defines a *dependence order* (denoted by $<_d$ ) which captures the data dependence relations among instructions, as one step in the specification of the constraints on memory order. Dependence order is determined from program order as follows:

$A <_d B$, if $A <_p B$ and at least one of the following is true:

- (d1) $A$ and $B$ are control dependent and $S(B)$
- (d2) $A$ writes a register read by $B$
- (d3) $A$ stores a memory location loaded by $B$

We do not define the three kinds of dependence in detail here to avoid dissecting the SPARC instruction set. Precise rules can be defined so that each dependence can be determined between every pair of instructions in a sequence by inspecting the sequence.

Caution is required in the definition of dependence order because it constrains memory order. If dependence order is too strict, it may unnecessarily constrain the range of legal processor implementations. There are two specification issues that should be mentioned. First, the definition (d2) does not define a dependence when two instructions write the same register or when an instruction reads a register then another writes it in order to allow register renaming in the processor implementation. Second, rule (d1) is defined so that there is a control dependence when an instruction affects a branch which is followed by a store, but not when the following instruction is a load. This ensures that the processor is allowed speculative execution of loads after a branch before the branch has been decided. Both of these decisions affect the executable description, which must include register renaming and speculative execution of loads if it is to be maximally general.

A particular memory total order $<_m$ is legal if $X_A <_m Y_B$ whenever one or more of the following conditions holds:

- (m1) $X_A <_d Y_B$ and $L(X_A)$
- (m2) $M(X_A, Y_B)$
- (m3) $A = B$ and $X_A <_p Y_B$ and $S(Y_B)$

Rule (m1) says that if two instructions are data dependent ($<_d$ ) and the first is a load, then they should be performed in order ($<_m$ ). Preceding stores may be delayed even if they are data dependent to following instructions. Rule (m2) describes the ordering constraint imposed by membars. Rule (m3) requires that stores to the same address be performed in program order. The rule also orders a load and a following store to the same address which is not captured by dependence order. This is necessary for processor self-consistency.

### 2.2 Value Axiom

While the ordering rules constrain the performance order of memory instructions, the following axiom defines the value returned by a load, Value ($L_A$), to be

$$\begin{aligned}&\text{Value } (S_A \mid S_A = \text{Max under } <_m \\ &\text{from the set of } \{S_A <_m L_A\} \cup \{S_A <_p L_A\}).\end{aligned}$$

Given a particular memory order, it implies that the value returned by a load is that of the latest store with respect to the memory order that is performed by the shared memory before the load ($\{S_A <_m L_A\}$)or that precedes the load in program order ($\{S_A <_p L_A\}$). Note that the store in the latter case should be the one issued by the same processor which issued the load, since $<_p$ does not order instructions from different processors.

## 3 The Executable Memory Model

The executable specification is intended to be maximally general—not only should it conform to the logical specification, it should generate every possible result that is allowed under the specification. Hence, it is more difficult in some sense to write the executable specification than to describe a particular multiprocessor, because a multiprocessor does not have to take advantage of every degree of freedom allowed by the logical specification. On the other hand, the executable model does not have to represent an efficient or practical solution, so it is much easier to design in that sense.

### 3.1 Mur$\varphi$ Description Language and Verifier System

Mur$\varphi$ is a description language for modeling finite-state asynchronous concurrent systems. There is an automatic verifier for Mur$\varphi$ which generates all of the reachable states of the system while checking for deadlock and other error conditions. Mur$\varphi$ can also check liveness and fairness properties (e.g., progress). The syntax of Mur$\varphi$ is derived from various standard programming languages, especially Pascal and C.

Mur$\varphi$ allows the declaration of familiar data types, including subranges of integers, arrays, records, and user-defined enumerations. A *state* of the described concurrent machine is an assignment to each global variable with a value in the range of the declared type. A Mur$\varphi$ program consists of a collection of *rules*. Each rule has a *condition*, which is Boolean expression referring to the global variables, and an *action*, which is a statement that modifies the values of the variables, yielding a new state.

Execution of a Mur$\varphi$ program begins with one of a set of initial states specified by the user. Then, the following loop is executed forever: Some rule whose condition is satisfied by the current state is chosen and its action evaluated, yielding a new current state. If there are no rules whose conditions are true, the execution halts. Although the action

```
Const
  ProcessorNum : 3;
  -- other constants are omitted.
Type
  Processor : 0 ..  ProcessorNum - 1 ;
  Address   : 0 ..  AddressNum - 1 ;
  Instruction : Enum{ Iload, Istore, Ildstore };
                -- other instructions are omitted.
  IssueIndex : 0 ..  ReorderBoxSize - 1 ;
  ReorderBoxType : Array[IssueIndex] of
                     Record  Instr : Instruction;
                             Addr  : Address;
                             Temp  : TempIndex;  -- and so on.
                     End;
  -- other types are omitted.
Var
  Memory    : Array [Address] of Value;
  Registers : Array [Processor] of Array [Register] of TempIndex;
  Temps  : Array [Processor] of Array [TempIndex] of Value;
  PC     : Array [Processor] of Label;
  CCR    : Array [Processor] of Boolean; -- Condition Code Register
  ReorderBox: Array [Processor] of ReorderBoxType;
  -- other variables are omitted.
```

Fig. 2. Global variables and type declarations for RMO description.

may be a compound statement consisting of a sequences of smaller statements, conditionals, and loops, it is executed *atomically*—no other rule can be executed before the action completes.

When several rule conditions are true at the same time, a choice is made arbitrarily, resulting in several possible executions. The Mur$\varphi$ verifier tries them exhaustively by depth-first or breadth-first search.

One essential construct in Mur$\varphi$ is the *ruleset*, which is used to describe a collection of rules that vary over a parameter. A ruleset can be thought of as nondeterministically selecting a value for the parameter from a set.

Several types of errors can be detected in a Mur$\varphi$ description. There is an *error* statement that can appear in an action. *Invariant* Boolean expressions may also be specified; if the invariant is false in any reachable state, an error is reported. The system can detect *deadlock states*, which are states that have no other states as successors. Finally, Mur$\varphi$ can check many common liveness and fairness properties using a subset of linear-time temporal logic.

If a problem of any type is detected, the verifier prints out a *diagnostic trace*, which is a sequence of states that leads to a state exhibiting the problem. In addition to the error traces, it is possible to print out the values of specified variables using `put` commands. This capability is used to obtain all the possible results of test programs.

### 3.2 RMO Description in Mur$\varphi$

The executable specification follows the intuition of Fig. 1. It describes reordering boxes, global memory with nondeterministic switch, and necessary part of processors. There are shared variables for all of the state of the system, including the processor registers, the memory, and the contents of the reorder boxes. Here, we provide excerpts from the description.

In the first part of Fig. 2, constants are declared for the number of processors, size of reorder box, size of memory, number of registers, and so on. For verification, these constants should be kept very small in order to bound the size of the state space that must be explored.

The first variable, `Memory`, models the global memory by an array of `value` indexed by `Address` type, which is declared to be a subrange of the integers. This description is based on a register renaming scheme, so the registers are a per processor array of temporary indices, which are pointers to temporaries in a register pool. The processor state is modeled using global variables: registers, program counter (PC), and condition code register (CCR). For simplicity, the variables used for speculation on branches are not shown here.

Each processor has a reorder box `ReorderBoxType`, which is an array of records. Each entry of the record keeps information about an instruction: the instruction type, memory address operand, temporary indices of register operands, and constant operand. The reorder box queues up every instruction from its processor in program order.

There are individual processes for the processors in the figure and for the memory. Only one process may execute at a time. The processes modeling the processors issue individual instructions by inserting them at the tail of a reorder box queue, so that instructions in a reorder box are always in program order.

For each instruction type, there is a procedure in the Mur$\varphi$ description that issues the instruction by inserting it in the reorder box. For example, `Load_init(proc, addr, reg)` inserts a load instruction with its operands after all previously issued instructions in the reorder box. The issuing function also handles register renaming, so that the instructions in the reorder box refer to temporary registers, not register names. When branch instructions are issued, a nondeterministic prediction of the branch direction is made. Instructions are then issued based on this

```
Ruleset p:Processor Do
Ruleset i:IssueIndex Do
  -- for all instructions, ReorderBox[p].instruction[i], check this.
  Alias R : ReorderBox[p];
       op : R.Ar[i].Instr Do
    Rule "Execute one of the instructions in minimal set"
       ( i <  R.Count )    -- instruction at i is valid one
     & ( op=Imem -> i=0 )  -- the membar no longer needs to be here
     & -- [ Rule 1 for <m ]
       ( Forall j:IssueIndex Do
           j<i -> ! ( Depend(p,j,i) & Is_wr_reg(p,j) )
         EndForall  )
     & -- [ Rule 2 for <m ]
       ( Forall j:IssueIndex Do
           j<i -> ! Membared(p,j,i)
         EndForall )
     & -- [ Rule 3 for <m ]
       ( Is_store(p,i) -> Forall j:IssueIndex Do
                            ! ( j<i & (Is_store(p,j)|Is_load(p,j) )
                            & mAddress(p,j) = mAddress(p,i) )
                          EndForall )
    ==>
    Begin -- The chosen instruction[p][i] is allowed to be performed.
      Switch op
        Case Iload   :Read(p, i, R.Ar[i].Addr, R.Ar[i].Temp);
        Case Ildstore:Ldstore_perf(p, i, R.Ar[i].Addr, R.Ar[i].Temp);
        Case Istore  :Store_perf(p, R.Ar[i].Temp, R.Ar[i].Addr);
        -- Others are omitted (do appropriate action for each.)
      End; -- Switch
      ArrangeBuffer(p,i);   -- Dequeue the buffer
    End; -- Rule
  End; -- Alias
End; -- Ruleset on IssueIndex
End; -- Ruleset on Processor
```

Fig. 3. RMO ordering constraints rule in Mur$\varphi$.

prediction. If the prediction turns out to be incorrect when the branch instruction is performed, the state of the registers and program counter is restored to what it was when the branch was issued, and the speculative instructions are cancelled. Since the logical specification allows speculative execution and ignores antidependences from register usage, these considerations are necessary to ensure that the executable specification generates every legal program result.

There is also a procedure to perform each type of instruction. These procedures are executed by the global memory and do most of the work of the instructions. For instance, the procedure `Store_perf()` performs a store instruction by writing the contents of the register into the memory location.

The description attempts to provide the most direct possible translation from the ordering rules given in the logical description. We show the Mur$\varphi$ rule implementing the memory order, after explaining some of the low-level predicates that it uses.

The function `Membared()` returns a Boolean value true if the two memory instructions at entry $i$ and $j$ are ordered through a membar. Function `Is_load(p,i)` checks whether the instruction at position $i$ in the reorder box for processor $p$ is a load instruction. Each *or*'ed expression checks if there is a memory barrier of the corresponding type in between the two instructions. The `Alias` command is used to define an abbreviation.

`Dir_Depend()` checks whether two instructions are dependence ordered, as defined in Section 2. At first, it ensures the instruction at $i$ precedes the one at $j$ in program order ($X <_p Y$). The rest of the Boolean expressions correspond to the rules (d1) through (d3). The expression for the rule (d2) calls `Dep_Reg()`, which returns true if the preceding instruction is writing the same register that is read by the following instruction (strictly speaking, the same *temporary* location in the register renaming scheme). Note that the dependence through the condition code register is checked separately, since branch instructions read the condition code register, which is modified by such instructions as test or compare. The third Boolean expression directly translates the rule (d3).

The global memory process nondeterministically selects a processor and an instruction in the processor, which is executed if it is legal to do so. The ordering rules are implemented in a function in the memory process that decides whether an instruction is legal to perform. An instruction is legal to perform only if the ordering rules allow the existence of a memory order in which the instruction is the minimum of all the instructions currently in the reorder box. Each ordering rule from the logical description is translated as directly as possible to a Mur$\varphi$ function, which checks whether the ordering rule is satisfied. For example, there is a recursive function (`Depend()` in Fig. 3) which, given a reorder box and the indices of instructions in it, returns true only if the instructions at those indices are dependence-ordered (this requires inspecting all of the instructions between the two indices).

When an instruction is legal, the memory performs it, which involves doing all of the computation associated with the instruction, including ALU operations and updating registers and/or memory. Then, it is removed from the reorder box.

In essence, a particular memory order is gradually constructed as the specification executes (the instructions that have been performed are memory ordered and those remaining in the reorder boxes have not yet been ordered). The constraints on nondeterministic choices involved in selecting the next instruction ensure that every legal memory order can be generated.

The main rule for the memory order constraints is in Fig. 3. This rule can also be thought of as implementing the behavior of the memory. The rule is embedded in parameterized rulesets that nondeterministically choose a reorder box and an instruction index. It performs the instruction at that reorder box index only if that instruction is allowed to appear first among all the instructions in the ordering box, according to the memory ordering rules.

The condition of the rule is a conjunction of several Boolean expressions. The first Boolean condition ensures that the chosen index of the reorder box is not an empty slot. The second condition requires every membar instruction to remain in the reorder box until all the previous instructions are executed and removed from the box.

The rest of the three Boolean expressions correspond to $<_m$ ordering rules (m1), (m2), and (m3), respectively. Note that the predicate $L(X)$ in the rule (m1) is replaced by the function checking if the instruction is writing to a register, because reorder boxes deal with all kinds of instructions, while the axioms in the previous section are aimed at enforcing orders among memory instructions only. The conditions are not direct translation of the rules, but they ensure that there is no preceding memory instruction which is $<_m$ ordered to the one at entry $i$. If the condition is true, then the chosen instruction is performed calling the corresponding procedure (e.g., `Read()` for a load and `Ldstore_perf()` for a load-store) according to the instruction type, and the instruction is removed from the reorder box.

One subtle point is the avoidance of starvation: The logical description requires that the memory order include every instruction. This implies that the memory must eventually perform every instruction in every reorder box. This requirement is handled in Mur$\varphi$ by requiring, in an infinite computation, that the oldest instruction in every reorder box be performed infinitely often (instructions not satisfying this requirement can be performed an arbitrary finite number of times between oldest instructions).

Since Mur$\varphi$ can only deal with finite-state processes, various memory structures must be bounded. Furthermore, the number of states grows exponentially with many parameters, so even quantities that are bounded in all implementations, such as the number of registers in a processor, are bounded much more sharply in the Mur$\varphi$ description. Bounded quantities include: the number processors, memory values, memory locations, registers, and reorder boxes.

If the Mur$\varphi$ program is considered without the bounds, it is equivalent to the logical specification. The executable specification in Mur$\varphi$ not only conforms to the logical specification but also generates all the possible behaviors allowed under the specification. We have proven this equivalence using a theorem prover. With the bounds, however, the executions of the Mur$\varphi$ program may be a subset of the executions allowed by the logical specification. For some programs, it is often easy to see that small bounds on all parameters allow sufficient resources to enumerate *all* of the possibilities. For larger descriptions, we must trade generality for the ability to verify automatically a bounded description.

## 4 Analyzing Test Programs with an Automatic Verifier

When developing the RMO model, it was very helpful to be able to find all of the possible outcomes of small example multiprocessor programs. The automatic verifier Mur$\varphi$ finds all of the reachable states of the system, so it can list all results very easily. When ordering rules are changed, it is simple to change the executable specification (since the translation is so direct) and run the test programs through the verifier to find out the consequences.

Running test examples in Mur$\varphi$ is different from running them on a real machine. An actual machine may not exercise all possible orderings, either because it does not implement a memory model in full generality or because the orderings are possible but happen not to occur in a particular run.

To make the operational description fully executable, we need to model programs running on the processors. This can be accomplished by adding rules to the processor description which specify which instruction to issue, as a function of the current PC value [21]. We have implemented a simple program that translates assembly-language programs into the appropriate rules for each processor, which are then combined with the executable specification to yield a Mur$\varphi$ description of that particular program running in the memory model.

Suppose we test the program at the top of Fig. 4. This program can be automatically translated to the rules in Fig. 5. The first rule corresponds to the first load of $P_0$ and the next rule to the last store of $P_1$. For readability, we have given symbolic names to memory locations and registers by defining them as constants. Note that the register `V3_1` of $P_1$ contains a constant value 3.

We have added another rule, shown in Fig. 5, which prints out the state of the registers and shared memory when the program terminates after executing all the instructions. Since Mur$\varphi$ does exhaustive searching, each result through every possible interleaved performance ordering is caught by the printing rule and printed out. Indeed, each possible result is printed many times (because it occurs in many different executions), but the results are then postprocessed to eliminate duplicates.

```
      Processor 0              Processor 1

ld  A, %r1                 ld  C, %rx
st #1, B                   membar #LoadLoad #LoadStore
st #2, C                   ld  B, %ry
                           membar #LoadStore
                           st #3, A

--TSO-------------------------------------------------
         A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):1
         A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):1
--PSO-------------------------------------------------
         A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):1
         A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):1
--RMO-------------------------------------------------
         A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):0
         A:3 B:1 C:2 r1(0):3 rx(1):0 ry(1):1
         A:3 B:1 C:2 r1(0):0 rx(1):0 ry(1):1
         A:3 B:1 C:2 r1(0):3 rx(1):2 ry(1):0
         A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):0
         A:3 B:1 C:2 r1(0):3 rx(1):2 ry(1):1
         A:3 B:1 C:2 r1(0):0 rx(1):2 ry(1):1
------------------------------------------------------
```

Fig. 4. An example test program and the corresponding set of possible results generated by the automatic verifier.

Fig. 4 shows the results obtained by running the verifier on the example program under the various SPARC memory models (this is actual program output). We assume an initial value of zero in every memory location and register. Each line lists the contents of the relevant memory locations and registers for a different terminating state of the program. Since all the memory operations of $P_1$ are ordered by membar instructions, they should be performed in the program order. However, the memory operations in $P_0$ can be reordered as far as they satisfy the ordering constraints of each memory model. The result shows that PSO allows more program behaviors than TSO and RMO allows more than those two, as expected.

When a user obtains an unexpected outcome of a test program, a trace can be generated which shows how the outcome can occur using a simple trick. An invariant can be added which asserts that the unexpected state does not occur. When the verifier finds the state, a counterexample trace will be generated automatically that gives the sequence of rules and intermediate states leading from an initial state to the state in question.

## 5 Verifying Illustrated Examples in the Architecture Manual

In many large-scale concurrent programs, the low-level synchronization code (which may even be generated by a compiler) is the only part that depends on the details of the memory model; this code can be carefully crafted to work in a particular memory model, then used elsewhere by programmers who need not be deeply familiar with its internals [22].

The SPARC architecture manual gives several assembly language routines for standard synchronization paradigms, including spin locks (two versions: one using load-store and one using swap), produce-consumer with a bounded buffer,

```
Rule  -- a sample rule corresponding to the load of processor 0
  PC[0]=0 & ReorderBox[0].Count < ReorderBoxSize
==>  begin Load_init(0, A, r1_0); end;

Rule  -- a sample rule corresponding to the store of processor 1
  PC[1]=4 & ReorderBox[1].Count < ReorderBoxSize
==>  begin Store_init(1, V3_1, A); end;

Rule "Print out the states when the program terminates"
  PC[0]=3 & PC[1]=5 &
  Forall p:Processor Do ReorderBox[p].Count = 0 End
==>
Begin
  Put " A:"; Put Memory[A];  -- Put prints the value.
  Put " B:"; Put Memory[B];  -- others are omitted
  Put " ry(1):"; Put Temps[1][Registers[1][ry_1]];
End;
```

Fig. 5. Mur$\varphi$ rule for assembly language programs.

and Dekker's algorithm. There are corresponding memory model versions for each algorithm. Fig. 6 shows the assembly language code for a spin lock using a load-store instruction. This is taken verbatim from the SPARC Architecture Manual, except that two instructions have been inserted (at the label `crit:`), to improve error detection.

In Fig. 6, the "lock held" condition is kept in a specific memory location `lock`. A nonzero value of the lock represents that the lock is held by some process, while a zero value means that the lock is free. An instruction `ldstub` loads a specified memory location and stores a nonzero value to the memory, atomically. The conditional branch `be` is taken if the special register `CCR` set by `tst` is zero. The following instruction (in this case, `nop`) is executed even if the branch is taken because the SPARC has delayed branching. Note that the `membars` enforce ordering between memory instructions in the critical section and others in the synchronization routine.

As shown in the spin lock code, we added two store instructions in critical region; one stores a constant value 1 to a critical memory location and the other also stores 0 in the location. The invariant below is used to check the mutual exclusion property of the spin lock when there are two processors.

```
Invariant "Mutual Exclusion of Memory Access"
! (Memory[CM0] = 1 & Memory[CM1] = 1);
```

Extending this to more processors is straightforward. It also ensures that a lock is not released too early, before the writes to the lock-protected location are completed. The verifier also checks for deadlocks.

The spin lock example described was computationally the most difficult, although all examples required the same order-of-magnitude time and space. When the spin lock example was modeled with three processors and a reorder box size of 10, the verifier explored 55,499 states in 12 minutes on SGI Indy using symmetry reduction method. The time is not proportional to the number of states because a state may be visited several times (depending on the number of incoming edges in the state graph) and because the amount of time for each rule varies with the complexity of the rules.

This spin lock is subject to starvation. It is possible for $P_0$ to be denied the lock forever even when $P_1$ releases the lock

```
---------------------------------------------------------------
 retry:  ldstub  [lock], %l0      -- load-store
         tst     %l0              -- branch if 0
         be      out
         nop
 loop:   ldub    [lock], %l0      -- load
         tst     %l0              -- branch if non-0
         bne     loop
         nop
         ba,a    retry            -- jump
 out:    membar  #LoadLoad #LoadStore
---------------------------------------------------------------
 crit:   stub    #1, [CM(i)]      -- store to a special
         stub    #0, [CM(i)]      -- location of each processor
---------------------------------------------------------------
 unlock: membar  #StoreStore      -- RMO, PSO only
         membar  #LoadStore       -- RMO only
         stub    %g0, [lock]      -- store value zero
---------------------------------------------------------------
```

Fig. 6. Assembly language program for spin lock synchronization.

infinitely often, because $P_1$ happens to be holding the lock whenever $P_0$ tests it. For this reason, Mur$\varphi$ reports a violation of the property `Eventually Memory[CM0]=1`, even though each process is assumed in the description to release its lock infinitely often. However, Mur$\varphi$ finds no violation of the weaker property that "at least one process gets the lock infinitely often," `Eventually (Memory[CM0]=1 | Memory[CM1]=1)`.

Ultimately, no unexpected behavior was found in the synchronization routines when combined with the appropriate models. Also, as expected, the TSO routine failed when combined with the PSO and RMO memory models.

The *state explosion problem,* which is a central problem in finite-state verification, has not been an issue in this effort because of the small size of the assembly language routines. However, it would become a problem for verification of larger programs.

## 6 Conclusion

We believe that this type of operational description strikes an appropriate balance between formality and understandability by programmers and machine architects. Moreover, the availability of formal verification tools allows users to experiment with the effects of the memory model on small assembly-language routines. Also, as we have learned in this experiment, developing an operational description and running the verifier can be very effective at clarifying the subtle details of the models and synchronization routines.

## Acknowledgments

Most of the detailed definition of the RMO model was done by a group consisting of Dennis Allison, Kourosh Gharachorloo, Paul Loewenstein, Andreas Nowatzyk, and the authors. Feedback from these individuals was essential in developing the Mur$\varphi$ model.

## References

[1] W.W. Collier, *Reasoning About Parallel Architectures.* Prentice Hall, 1992.
[2] J. Protić, M. Tomašević, and V. Milutinović, eds., *Distributed Shared Memory: Concepts and Systems.* IEEE CS, 1998,
[3] L Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs," *IEEE Trans. Computers,* vol. 28, no. 9, pp. 690-691, Sept. 1979.
[4] J.R. Goodman, "Cache Consistency and Sequential Consistency," Technical Report 61, SCI Committee, Mar. 1989.
[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 15-26, May 1990.
[6] K. Gharachorloo, "Memory Consistency Models for Shared-Memory Multiprocessors," PhD thesis, Stanford Univ., 1996.
[7] SPARC International, *The SPARC Architecture Manual Version 8.* Prentice Hall, 1992.
[8] P.S. Sindhu, J.-M. Frailong, and M. Cekleov, "Formal Specification of Memory Models," Technical Report CSL-91-11, Xerox Palo Alto Research Center, Dec. 1991.
[9] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l Symp. Computer Architecture,* pp. 434-442, June 1986.
[10] S. Adve and M. Hill, "Weak Ordering—New Definition and Some Implications," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 2-14, 1990.
[11] D. Weaver and T. Germond, eds., *The SPARC Architecture Manual Version 9.* Prentice Hall, 1994.
[12] R. Sites and R. Witek, eds., *Alpha AXP Architecture Reference Manual, second ed..* Digital Press, 1995.
[13] C. May, E. Silha, R. Simpson, and H. Warren, eds., *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann, 1994.
[14] S. Adve and M. Hill, "Implementing Sequential Consistency in Cache-Based Systems," *Proc. Ninth Int'l Symp. Parallel Processing,* pp. 47-50, Aug. 1990.
[15] C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," *Proc. 14th Ann. Int'l Symp. Computer Architecture,* pp. 234-243, 1987.
[16] D.L. Dill, "The Mur$\varphi$ Verification System," *Proc. Computer Aided Verification, Eighth Int'l Conf., CAV '96,* pp. 390-393. Springer-Verlag, July 1996.
[17] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 107-125, Feb. 1995.
[18] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill, "Programming for Different Memory Consistency Models," *J. Parallel and Distributed Computing,* vol. 15, no. 4, pp. 399-407, Aug. 1992.
[19] P. Gibbons, M. Merritt, and K. Gharachorloo, "Proving Sequential Consistency of High-Performance Shared Memories," *Proc. Third ACM Symp. Parallel Algorithms and Architectures,* pp. 292-303, July 1991.
[20] N. Lynch, "I/O Automata: A Model for Discrete Event Systems," *Proc. 22nd Ann. Conf. Information Science and Systems,* Princeton Univ., Mar. 1988.
[21] D.L. Dill, S. Park, and A. Nowatzyk, "Formal Specification of Abstract Memory Models," *Research on Integrated Systems: Proc. 1993 Symp.,* pp. 38-52. MIT Press, Mar. 1993.
[22] M. Ben-Ari, *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

**Seungjoon Park** received the BS degree with honors and the MS degree in electronics engineering from Seoul National University, and the PhD degree in electrical engineering with minor in computer science from Stanford University. From 1996 to 1998, he was an engineering research associate in the Department of Computer Science, Stanford University. Since 1998, he has been a research scientist of RIACS, working in the automated software engineering group of NASA Ames Research Center, California. His research interests include hardware and software verification, formal methods, computer architecture, and computer-aided digital systems design.

**David L. Dill** received the SB in electrical engineering and computer science from the Massachusetts Institute of Technology in 1979 and the MS and PhD from Carnegie Mellon University in 1982 and 1987, respectively. He is an associate professor of computer science and, by courtesy, electrical engineering at Stanford University. He has been on the faculty at Stanford since 1987. His primary research interests relate to the theory and application of formal verification techniques to system designs, including hardware, protocols, and software. He has also done research in asynchronous circuit verification and synthesis and in verification methods for hard real-time systems. He was the chair of the Computer-Aided Verification Conference held at Stanford University in 1994. From July 1995 to September 1996, he was Chief Scientist at 0-In Design Automation. Prof. Dill's PhD thesis, "Trace Theory for Automatic Hierarchical Verification of Speed Independent Circuits" was named as a Distinguished Dissertation by the ACM and published as such by MIT Press in 1988. He was the recipient of a Presidential Young Investigator award from the U.S. National Science Foundation in 1988 and a Young Investigator award from the Office of Naval Research in 1991. He has received Best Paper awards at the International Conference on Computer Design in 1991 and the Design Automation Conference in 1993 and 1998.