Optimistic Synchronization-Based State-Space Reduction

Scott D. Stoller* Ernie Cohen†

13 March 2006

Abstract

Reductions that aggregate fine-grained transitions into coarser transitions can significantly reduce the cost of automated verification, by reducing the size of the state space. We propose a reduction that can exploit common synchronization disciplines, such as the use of mutual exclusion for accesses to shared data structures. Exploiting them using traditional reduction theorems requires checking that the discipline is followed in the original (*i.e.*, unreduced) system. That check can be prohibitively expensive. This paper presents a reduction that instead requires checking whether the discipline is followed in the reduced system. This check may be much cheaper, because the reachable state space is smaller.

1 Introduction

For many concurrent software systems, a straightforward model of the system has such a large and complicated state space that automated verification, by automated theorem-proving or state-space exploration (model checking), is infeasible. *Reduction* is an important technique for reducing the size of the state space by aggregating transitions into coarser-grained transitions.

When exploring the state space of a concurrent system, context switches between threads are typically allowed before each transition. A simple example of a reduction for concurrent systems is to inhibit context switches before transitions that access only unshared variables. This effectively increases the granularity of transitions. Thus, one can regard this and similar reductions as defining a *reduced system*, which is a coarser-grained version of the original system. The reduced system may have dramatically fewer states than the original system. A *reduction theorem* asserts that certain properties are preserved by the transformation.

We consider a more powerful reduction that exploits common synchronization disciplines. For example, in a system that uses mutual exclusion on accesses to some shared variables—called *protected variables*—our reduction inhibits context switches before transitions that access only unshared variables and protected variables. Such transitions are called *invisible* transitions; other transitions are called *visible* transitions. Informally, this reduction is safe because protected variables cannot be accessed concurrently, and allowing context switches before the synchronization operations (lock acquire, *etc.*) that protect them is sufficient. The model-checking experiments

*This work was supported in part by NSF under Grants CCR-9876058, CCR-0205376, and CNS-0509230 and ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Address: Computer Science Dept., Stony Brook University, Stony Brook, NY 11794-4400. Email: stoller@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~stoller/

†Microsoft Corp. Email: ernie.cohen@acm.org

reported in [Sto02] are based on a similar reduction, which decreased memory usage (which is proportional to the number of states) by a factor of 25 or more. Such reductions can also decrease the computational cost of the automated theorem-proving needed for thread-modular verification [FFQ02, FQS02].

Traditional reduction theorems, such as [Lip75, CL98, Coh00], can also exploit such synchronization disciplines. However, a hypothesis of these traditional theorems is that the allegedly protected variables are indeed protected (by synchronization that enforces mutual exclusion) in the original (*i.e.*, unreduced) system. How can we establish this? Static analyses like [FF01] can automatically provide a conservative approximation but sometimes return "don't know". For general finite-state systems, it might seem that the only way to automatically obtain exact information about whether the synchronization discipline is followed (*i.e.*, the selected variables are actually protected) is to express this condition as a history property and check it by state-space exploration of the original system. But this would be about as expensive as checking correctness requirements on the original system, making the reduction almost pointless.

Our reduction theorem implies that one can determine exactly during state-space exploration of the *reduced* system whether the synchronization discipline is followed in the original system.

For generality, the reduction theorem is expressed without explicit reference to mutual exclusion or synchronization. It is expressed in terms of a predicate q, which in the application of the reduction theorem to mutual exclusion synchronization is chosen to be the history predicate "the synchronization discipline has been violated". The theorem assumes that the transition relation of each thread is partitioned into invisible transitions and visible transitions, as described above. This allows us to define the reduced system, in which context switches are allowed only immediately before visible transitions. The reduction theorem states that if the original system has a reachable state in which q holds, then so does the reduced system, provided the invisible transitions satisfy several conditions, most notably that (*i*) a transition cannot enable or disable invisible transitions of other threads, (*ii*) as long as q is false (in other words, as long as the synchronization discipline is followed), a transition commutes to the right of an invisible transition of another thread, and (*iii*) invisible transitions cannot falsify q (in other words, they cannot hide a violation of the synchronization discipline).

In the application to mutual exclusion synchronization, informally, the first condition above holds because synchronization operations that may block are visible; the second condition holds because, in the absence of violations of the synchronization discipline, the set of variables accessed by a transition is disjoint from the set of variables accessed by an immediately following invisible transition of another thread, because accesses to a protected variable by different threads are separated by intervening synchronization operations; and the third condition holds because, once the synchronization discipline has been violated, it remains violated for the rest of the execution. Note that one needs to prove only once that the hypotheses of the reduction theorem hold when instantiated for mutual exclusion synchronization; this establishes applicability of the reduction theorem to all systems that use such synchronization. The role of this proof is analogous to the role of proofs needed with traditional partial-order methods to show validity of a proposed independence relation on operations of a data type (*e.g.*, queues or locks).

To apply the reduction theorem to a system that uses mutual exclusion synchronization, the user guesses which variables are protected (this determines which transitions are visible, as described above) and how they are protected. The latter is done by supplying *exclusive access predicates* [FQ03]. For each protected variable x and each thread i, there is an exclusive access predicate e_i^x. The synchronization discipline requires that e_i^x hold in states from which thread i can execute a transition that accesses x. Mutual exclusion is expressed by the requirement that, for every variable x and every two distinct threads i and j, e_i^x and e_j^x are mutually exclusive (*i.e.*, cannot hold simultaneously). Locks, by themselves or in the form of monitors, are probably the most widely used synchronization mechanism. For systems that use them, we describe in Section 9 how to automatically guess which variables are protected (by monitors) and determine the associated exclusive access predicates.

Our reduction theorem is designed to be used together with traditional reduction theorems. Suppose a traditional reduction theorem asserts that some property ϕ is preserved by the reduction if the original system follows the synchronization discipline. After checking that the reduced system follows the discipline and satisfies ϕ, one can use our reduction theorem to conclude that the original system follows the discipline, and then use the traditional reduction theorem to conclude that the original system satisfies ϕ.

A simple example of checking the hypotheses of a reduction during state-space exploration of the *reduced* system is mentioned in [HP95]. There, the hypotheses to be checked are whether specified processes ever access specified variables. Proving soundness in that case is relatively easy, because the hypotheses are unaffected by re-ordering of the events in an execution.

The reduction in [Sto02] is similar in spirit to the one in this paper. The main contributions of this paper relative to [Sto02] are a reduction that applies to systems that use arbitrary synchronization mechanisms to achieve mutual exclusion (the results in [Sto02] apply only when monitors are used), and significantly shorter and cleaner proofs, based on ω-algebra. Similar results could presumably be proved in a transition-system framework, like the one in [God96], but our experience attempting to do that suggests that the algebraic framework makes the proofs easier to discover, shorter, and cleaner.

The main contribution of this paper compared to an earlier version [SC03] is a more liberal definition of "invisible transition", which allows some synchronization operations—for example, the release, notify, and notifyAll operations on monitors—to be classified as invisible. The definition in [SC03] forces, roughly speaking, all synchronization operations to be classified as visible.

Our method and traditional partial-order methods (*e.g.*, stubborn sets [Val97], ample sets [CGP99], and persistent sets [God96]) both exploit independence (commutativity) of transitions, but our method can establish independence of transitions—and hence achieve a reduction—in many cases where traditional partial-order methods cannot. Traditional partial-order methods, as implemented in tools such as Spin [Hol97] and VeriSoft [God97], use two kinds of information to determine independence of transitions: program-specific information, obtained by static analysis, about which processes may perform which operations on which objects (*e.g.*, only process P_2 sends messages on channel C_1), and manually supplied program-independent information about dependencies between operations on selected datatypes (*e.g.*, a send operation on a full channel is disabled until a receive operation is performed on that channel).

Our method has two main advantages over traditional partial-order methods. First, our method can exploit more complicated program-specific information to determine independence of transitions, *e.g.*, the invariant that a particular variable is always protected by particular synchronization constructs. Such invariants take into account the context (specifically, synchronization context) in which operations are performed. In contrast, traditional partial-order methods are based on analysis of which operations are performed by each thread with little regard for the context in which the operations occur. Second, our method does not rely on any conservative static analysis. In contrast, traditional partial-order methods rely on conservative static analysis to determine which processes may perform which operations on which objects; for example, static analysis may be used to determine whether more than one thread can invoke a given operation on the queue accessed by a given program statement. For programs in relatively simple modeling languages, inexpensive and precise static analysis of such properties is feasible. For programs that contain references (or pointers), arrays, procedure calls, and dynamic thread creation, conservative static analyses will generally be imprecise. This imprecision will cause opportunities for reduction to be overlooked, decreasing the effectiveness of the traditional partial-order method. Since our method does not rely on any conservative static analysis, it has no difficulty with references, *etc.*

Section 2 presents some motivating examples. Section 3 introduces omega algebra, which is a simple and powerful framework for reductions. Section 4 presents and proves the reduction theorem. The theorem is expressed in a very general algebraic style and is applicable to a variety of system models, *e.g.*, shared variables or message passing. Section 5 defines a simple model of concurrent systems with shared variables, and Section 6 defines a synchronization discipline based on mutual exclusion. Section 7 shows that the reduction theorem applies in that context. Section 8 presents a methodology for using the reduction. Section 9 describes how the methodology can be automated for systems that use monitors for synchronization. Section 10 uses a simple example to compare our reduction with traditional partial-order methods.

2 Motivating Examples

This section describes three examples of systems for which the current reduction is more effective (at reducing the number of explored states) than traditional partial-order methods and the reduction in [Sto02]. For the first example, we explain why in some detail; explanations for the other examples are roughly similar. These examples are based mainly on descriptions in [SBN$^+$97] of code in real systems.

Semaphores. A user thread gets a buffer from a buffer pool, sends request to a device driver thread, supplying the operation type (read or write), a buffer, and a semaphore as arguments, and then waits for completion of the operation by invoking down() on the semaphore. The device driver thread receives the request, performs the operation (reading or writing the buffer as appropriate), and then calls up() on the semaphore. The buffers can be classified as protected variables, allowing transitions that access them to be classified as invisible by our reduction.

For concreteness, consider a system with two user threads and one driver thread, running the following pseudo-code. The ellipses represent the actual device access and other operations. Each thread's local variables are subscripted by a thread identifier. Uppercase letters denote control points.

```
user1 : $^{A}b_1 = \text{getBuf}();\, ^{B}\text{sendRequest}(\text{READ}, b_1, s_1);\, ^{C}\text{down}(s_1);\, ^{D}\text{read}(b_1);\, ^{E}\cdots$
user2 : $^{A}b_2 = \text{getBuf}();\, ^{B}\text{write}(b_2);\, ^{C}\text{sendRequest}(\text{WRITE}, b_2, s_2);\, ^{D}\text{down}(s_2);\, ^{E}\cdots$
driver : $^{A}$while (true)
            $^{B}\text{receiveRequest}(op_d, b_d, s_d);$
            $^{C}\text{if } (op_d = \text{READ}) \cdots {}^{D}\text{write}(b_d) \cdots \text{else} \cdots {}^{E}\text{read}(b_d) \cdots$
            $^{F}\text{up}(s_d)$
```

Thread user1 has exclusive access to the buffer to which b_1 points when user1 is at control point B, D, or E. Other threads have similar exclusive access predicates for buffers. Let pc_i denote the program counter of thread i. The exclusive access predicates for buffers are:

$$\begin{aligned}
e^b_{\text{user1}} &= *b_1 = b \wedge pc_1 \in \{B, D, E\} \\
e^b_{\text{user2}} &= *b_2 = b \wedge pc_2 \in \{B, C, E\} \\
e^b_{\text{driver}} &= *b_d = b \wedge pc_d \in \{C, D, E, F\}
\end{aligned}$$

Consider a state s_0 in which $pc_1 = C \wedge pc_2 = B \wedge pc_d = D$. With the reduction in this paper, buffers are protected variables, so reads and writes of buffers are invisible, and our reduction inhibits context switches before them. Accesses to unshared variables, such as op_d, are also invisible. Thus, the driver will receive the request, test the condition on op_d, and write to the buffer without any intervening context switches. In contrast, traditional partial-order methods, even sophisticated ones, will allow a context switch before the driver's access to the buffer; this increases the number of explored states. For concreteness, consider selective search using persistent sets computed by the conditional stubborn set algorithm (CSSA) [God96]. A persistent set in a state s is a subset of the enabled transitions in s that satisfies certain conditions. The selective search explores, from each state, only a persistent set of transitions. The conditions in the definition of persistent set ensure that this preserves certain properties of the state space, such as reachability of deadlocks. CSSA is parameterized by a statically determined binary dependence relation, called might-be-the-first-to-interfere-with, on operations. In this example, static alias analysis determines that b_2 and b_d may be aliased, *i.e.*, they may point to the same buffer (at the same or different times). Consequently, the might-be-the-first-to-interfere-with relation relates each write(b_2) operation with each write(b_d) operation, and so on. In state s_0, user2 and the driver have enabled transitions that perform write(b_2) and write(b_d), respectively, so CSSA includes transitions of both threads in the persistent set. The reduction in [Sto02], which is based on analysis of locks, is not effective for this system, because it uses semaphores.

Memory Re-use. Some systems re-use objects (or structures) by placing them on a free list when they are not in use. These objects may be protected by different locks each time they are re-used, violating the locking discipline of [Sto02]. For example, consider a file system in which blocks in a file are protected by the lock associated with (the i-node of) that file, and blocks on the free list are protected by the lock associated with the free list. A block may be in a different file, and hence protected by a different lock, each time it is re-used. Let m_F denote the lock associated with the free list. Let m_f denote the lock associated with file f. The exclusive access predicate e_i^b for a block b might be

$$(\text{onFreeList}(b) \wedge m_F.\mathit{owner} = i) \vee (\exists \text{ file } f : \text{allocatedTo}(b, f) \wedge m_f.\mathit{owner} = i)$$

Master-Worker Paradigm. In the master-worker paradigm, a master thread assigns tasks to worker threads. Typically, each task is represented by an object created by the master thread and passed to a worker thread. The master thread does not access a task object after passing it to a worker. Task objects can be classified as protected. Suppose each worker thread w has a field w.task that refers to the worker's task. For a task object x, the exclusive access predicate e_{master}^x holds before x has been passed to a worker thread, and e_w^x holds when $w.\text{task} = x$.

3 Omega Algebra

An *omega algebra* is an algebraic structure over the operators (listed in order of increasing precedence) 0 (nullary), 1 (nullary), + (binary infix), $\cdot$ (binary infix, usually written as simple juxtaposition), $\star$ (binary infix, same precedence as $\cdot$), * (unary suffix), and $^\omega$ (unary suffix), satisfying the following axioms[1]:

$$\begin{array}{rclrcll}
(x+y)+z & = & x+(y+z) & x \leq y & \Leftrightarrow & x+y=y & \\
x+y & = & y+x & & & & \\
x+x & = & x & x^* & = & 1+x+x^*\ x^* & \\
0+x & = & x & x\ y \leq x & \Rightarrow & x\ y^* = x & (\text{* ind R}) \\
x\ (y\ z) & = & (x\ y)\ z & x\ y \leq y & \Rightarrow & x^*\ y = y & (\text{* ind L}) \\
0\ x = x\ 0 & = & 0 & & & & \\
1\ x = x\ 1 & = & x & x \star y & = & x^\omega + x^*\ y & \\
x\ (y+z) & = & x\ y + x\ z & x^\omega & = & x\ x^\omega & \\
(x+y)\ z & = & x\ z + y\ z & x \leq y\ x + z & \Rightarrow & x \leq y \star z & (\star\ \text{ind})
\end{array}$$

(Here, as throughout the paper, in displayed formulas and theorems variables w, x, y, z are implicitly universally quantified over all omega algebra terms.) In parsing formulas, $\cdot$ and $\star$ associate to the

[1]The axioms are equivalent to Kozen's axioms for Kleene algebra [Koz94], plus the three axioms for omega terms.

right; e.g., $u\ v \star x \star y$ parses and expands to $(u \cdot (v^\omega + v^* \cdot (x^\omega + x^* \cdot y)))$. In proofs, we use the hint "(distributivity)" to indicate application of the distributivity laws, and the hint "(hyp)" to indicate the use of hypotheses. In induction steps that use induction, we use the hint "$t_1\ t_2 \leq t_1$; (* ind R)" to indicate use of the first induction axiom (with t_1 for x and t_2 for y), and dually for the second induction axiom. If x_i is a finite collection of terms over the range of i, we write $(+i : x_i)$ and $(\cdot i : x_i)$ for the sum and product, respectively, of these terms.

These axioms are sound and complete for the usual equational theory of omega-regular expressions; more precisely, completeness holds only for *standard* terms, where the first arguments to $\cdot$, $^\omega$, and $\star$ are regular. Thus, we make free use, without proof, of familiar equations from the theory of (omega-)regular languages (e.g., $x^*\ x^* = x^*$, $(1+x)^* = x^*$), indicated by the hint "(regular algebra)". When an (in)equality appears as a hint without other reference, this hint is implicit.

y is a *complement* of x iff $x\ y = 0 = y\ x$ and $x + y = 1$. It is easy to show that complements (when they exist) are unique and that complementation is an involution; a *predicate* is an element of the algebra with a complement. In this paper, p and q (possibly with subscripts) range over predicates, with complements $\overline{p}$ and $\overline{q}$. It is easy to show that the predicates form a Boolean algebra, with $+$ as disjunction, $\cdot$ as conjunction, 0 as *false*, 1 as *true*, complementation as negation, and $\leq$ as implication. Equations true in all Boolean algebras (e.g., $p\ q = q\ p$) are freely used in proofs, indicated by the hint "(Boolean algebra)"; such a hint implicitly carries the claim that all of the terms in the hint denote predicates.

The omega algebra axioms support several interesting programming models, where (intuitively) 0 is magic[2], 1 is skip, $+$ is chaotic nondeterministic choice, $\cdot$ is sequential composition, $\leq$ is refinement, x^* is executed by executing x any finite number of times, and x^ω is executed by executing x an infinite number of times. The results of this paper are largely motivated by the *relational model*, where terms denote binary relations over a state space, 0 is the empty relation, 1 is the identity relation, $\cdot$ is relational composition, $+$ is union, * is reflexive-transitive closure, $\leq$ is subset, and x^ω relates an input state s to an output state if there is an infinite sequence of states starting with s, with consecutive states related by x. Thus, x^ω relates an input state to either all states or none, and $x^\omega = 0$ iff x is well-founded. Predicates are identified with the identity relation on the set of states in their domain; thus, a predicate can be executed, as a no-op, from the states in which it holds. Define $\top = 1^\omega$. One can show that $\top$ is the maximal element under $\leq$, and in the relational model, it relates all pairs of states (because it relates an input state s to an output state if there is an infinite sequence of states starting with s and with consecutive states related by the identity relation, and there is such a sequence).

In addition to equational identities of regular languages, we will use the following standard theorems (more sophisticated theorems of this type appear in [Coh00]). Algebraic lemmas and theorems in this paper are presented as numbered equations followed by some vertical space followed

[2]magic is the program that has no possible executions (and so satisfies every possible specification). Of course, it cannot be implemented.

by a formal proof.

(1) $x\ y \le y\ z \Rightarrow x^*\ y \le y\ z^*$

$$\begin{array}{lll}
x^*\ y & \le & \{1 \le z^* \hfill\} \\
x^*\ y\ z^* & = & \{x\ y\ z^* \le y\ z^* \ \ \text{(below)};\ \ (\text{* ind L})\} \\
y\ z^* & &
\end{array}$$

$$\begin{array}{lll}
x\ y\ z^* & \le & \{x\ y \le y\ z \ \ \text{(hyp)} \hfill\} \\
y\ z\ z^* & \le & \{z\ z^* \le z^* \hfill\} \\
y\ z^* & &
\end{array}$$

□

(2) $y\ x \le x\ y \Rightarrow (x+y)^* = x^*\ y^*$

$$\begin{array}{lll}
(x+y)^* & \le & \{1 \le x^*\ y^* \hfill\} \\
x^*\ y^*\ (x+y)^* & = & \{x^*\ y^*\ (x+y) \le x^*\ y^* \ \ \text{(below)};\ \ (\text{* ind R})\} \\
x^*\ y^* & \le & \{\text{(regular algebra)} \hfill\} \\
(x+y)^* & &
\end{array}$$

Since the first and last terms are equal, the first and third terms are equal.

$$\begin{array}{lll}
x^*\ y^*\ (x+y) & = & \{\text{(distributivity)} \hfill\} \\
x^*\ y^*\ x + x^*\ y^*\ y & \le & \{y\ x \le x\ y \ \ \text{(hyp), so}\ \ y^*\ x \le x\ y^*\ (1)\} \\
x^*\ x\ y^* + x^*\ y^*\ y & \le & \{x^*\ x \le x^*;\ \ y^*\ y \le y^* \hfill\} \\
x^*\ y^* & &
\end{array}$$

□

(3) $y\ x \le (x+1)\ y \Rightarrow (x+y)^* = x^*\ y^*$

$$\begin{array}{lll}
(x+y)^* & = & \{\text{(regular algebra)} \hfill\} \\
(x+1+y)^* & = & \{y\ (x+1) \le (x+1)\ y \ \ \text{(below)};\ \ (2)\} \\
(x+1)^*\ y^* & = & \{(x+1)^* = x^* \hfill\} \\
x^*\ y^* & &
\end{array}$$

$$\begin{array}{lll}
y\ (x+1) & = & \{\text{(distributivity)} \hfill\} \\
y\ x + y\ 1 & \le & \{y\ x \le (x+1)\ y \ \ \text{(hyp)};\ \ y\ 1 = y = 1\ y\} \\
(x+1)\ y + 1\ y & = & \{1 \le x+1 \hfill\} \\
(x+1)\ y & &
\end{array}$$

□

4 A Reduction Theorem

We consider systems composed of a fixed, finite, nonempty set of concurrent processes (each perhaps internally concurrent and nondeterministic). Variables i and j range over process indices. Each process i has a visible action v_i and an invisible action u_i[3], where the invisible action is constrained to neither receive information from other processes nor to send information to other processes so as to create a race condition in the recipient. This constraint is guaranteed only so long as some global synchronization policy is followed. For example, in a system where processes are synchronized using locks, either visible or invisible actions of process i might modify variables that are either local to process i or protected by locks held by process i, release locks, or send asynchronous messages to other processes; but only visible actions can acquire locks or wait for a condition to hold. Note that violation of the synchronization discipline (e.g., an action accessing a shared variable without first obtaining an appropriate lock) might cause a race condition between an invisible action and the actions of another process, violating the constraint on invisible actions.

To avoid introducing temporal operators, we introduce a Boolean history variable q that records whether the synchronization discipline has been violated at some point in the execution. Predicate p_i means that process i cannot perform an invisible action, *i.e.*, that u_i is disabled. Let p be the conjunction of the p_i's:

(4) $\quad p = (\cdot i : p_i).$

A state satisfying p is called *visible*; thus, in a visible state, all invisible transitions are disabled.

We now define several actions, formalized in the definitions (5)–(11) below. An M_i action consists of a visible action of process i followed by a sequence of invisible actions of process i. An N_i action is an M_i action that is "maximal" (i.e., further u_i actions are disabled) and that finishes in a state where the synchronization discipline has not been violated. N_i is effectively the transition relation of thread i in the reduced system. Additional conditions will imply that executing an N action in a visible state results in a visible state; thus, in the reduced system, context switches occur only in visible states. A u (respectively v, M, N) action is a u_i (respectively, v_i, M_i, N_i) action of some process i. Finally, an R action is executable iff (*i*) the discipline has been violated, or (*ii*) such a violation is possible after execution of a single M action. Like x^ω, R relates each initial state to either all final states or none.

(5) $\quad M_i \; = \; v_i \, u_i^*$

(6) $\quad N_i \; = \; M_i \, p_i \, \overline{q}$

(7) $\quad u \; = \; (+i : u_i)$

(8) $\quad v \; = \; (+i : v_i)$

(9) $\quad M \; = \; (+i : M_i)$

[3]Note that u_i and v_i can be sums of nondeterministic actions that correspond to individual transitions of process i.

(10) $N = (+i : N_i)$

(11) $R = (1 + M)\ q\ \top$

Our reduction theorem says that if the original system can reach a violation of the synchronization discipline starting from some visible state, then the reduced system can also reach a violation starting from the same initial state, except that the violation might occur partway through the last transition of the reduced system (*i.e.*, the last transition might be an M action rather than an N action). The transition relations of the original and reduced systems are $u + v$ and N, respectively. Thus, the conclusion of the reduction theorem is $p\ (u + v)^*\ q \le N^*\ R$. This says that if a state s_2 satisfying q is reachable from a visible state s_1 in the original system—in other words, $\langle s_1, s_2 \rangle$ is in the relation $p\ (u + v)^*\ q$—then $\langle s_1, s_2 \rangle$ is also in $N^*\ R$. Expanding the definition of R and recognizing that $\top$ is the full relation, this means that for some s_3, $\langle s_1, s_2 \rangle$ is in $N^*\ (1 + M)\ q$, *i.e.*, a state satisfying q is reachable from s_1 in the reduced system, except that the last transition might be incomplete (*i.e.*, it might be an M instead of an N).

The hypotheses of our reduction theorem are as follows, formalized in formulas (13)–(21) below. It is impossible to execute invisible actions of a single process forever without violating the discipline (13); in other words, the process eventually executes a visible transition, violates the discipline, or gets stuck. This hypothesis is needed to show that, if a violation occurs in the original system when nultiple threads are in invisible states, all threads except the one causing the violation can be advanced to visible states (or to an earlier violation) in a finite number of steps; thus, it suffices to allow a single M transition in the conclusion of the reduction.

An action cannot enable or disable an invisible action of another process; specifically, p_i holds after u_j or v_j iff it holds before (14),(15). In the absence of a violation, an action commutes to the right of an invisible action of another process; specifically, if executing u_j or v_j followed by u_i leads from a state s_1 to a state s_2, and we try to move the u_j or v_j to the right by executing it after the u_i, then one of three outcomes must occur: a violation occurs after the u_i, a violation occurs after the u_j or v_j, or we reach the same state s_2 (16),(17).

The next two hypotheses say that p_i holds iff u_i is disabled. The first of them says that p_i implies u_i is disabled; specifically, no state is reachable by executing u_i from a state where p_i holds (18). The second of them says that u_i is disabled implies p_i; specifically, in every state s_1, either u_i is enabled (leading to some state s_2, which $\top$ relates to s_1) or p_i holds (recall that predicates are modeled as subsets of the identity relation) (19).

Visible and invisible actions of a process cannot be simultaneously enabled; specifically, no state is reachable by executing v_i from a state satisfying p_i (20). Invisible actions cannot hide violations of the discipline, *i.e.*, if q holds before u_i, then q holds after u_i; specifically, the subset of u_i containing pairs whose first state satisfies q is a subset of the subset of u_i containing pairs whose second state satisfies q (21).

Define, for any x,

(12) $[x] = x + q\ \top + x\ q\ \top$

Intuitively, $[x]$ behaves like x, except that it is allowed to behave arbitrarily if started in a state where the discipline has been violated, and may behave arbitrarily after performing x if x results in a state where the discipline has been violated.

(13) $\quad (u_i\ \overline{q})^\omega = 0$

(14) $\quad i \neq j \Rightarrow u_j\ p_i = p_i\ u_j$

(15) $\quad i \neq j \Rightarrow v_j\ p_i = p_i\ v_j$

(16) $\quad i \neq j \Rightarrow u_j\ u_i \leq u_i\ [u_j]$

(17) $\quad i \neq j \Rightarrow v_j\ u_i \leq u_i\ [v_j]$

(18) $\quad p_i\ u_i = 0$

(19) $\quad 1 \leq p_i + u_i\ \top$

(20) $\quad \overline{p}_i\ v_i = 0$

(21) $\quad q\ u_i \leq u_i\ q$

Our reduction theorem can be used to check not only the synchronization discipline, but also the invariance of any other predicate I such that violations of I cannot be hidden by invisible actions. To see this, note that, except for (21), the conditions above are all monotonic in q. Thus, if all the conditions above (including (21)) are satisfied for a predicate q, and there is a predicate I such that $I\ u_i \leq u_i\ I$ for each i, then all the conditions are still satisfied if q is replaced with $q + I$.

The proof below can be viewed as formalizing the following construction, which starts from an execution that violates the discipline and produces an execution of the reduced system that also violates the discipline. First, we try to move invisible u_i actions to the left of u_j and v_j actions, where $i \neq j$, starting from the left (*i.e.*, from the leftmost u_i action that immediately follows a u_j or v_j action). The u_i action cannot make it all the way to the beginning of the execution (since $p\ u_i = 0$), so it must eventually run into either another u_i or a v_i. Repeating this produces an execution in which a sequence of M actions leads to a violation of the discipline.

Next, we try to turn all but the last of these M actions into N actions, starting from the next to last M action. In general, we will have done this for some number of M actions, so we will have an execution that ends with $N^*\ R$. Now try to convert the last M_i before the $N^*\ R$ suffix into an N action. Suppose this M_i action ends with u_i enabled. u_i must then also be enabled later when the discipline is first violated (because (14) and (15) imply N_j does not affect enabledness of u_i, and (20) implies N_i is disabled when u_i is enabled), so we add a u_i action just after the violation and try to push it backward (through the $N^*\ (1 + M)$). This may create additional violations of the discipline, but there will always be an $N^*\ R$ to the right of the new u_i. Eventually, u_i makes it back to the M_i, extending M_i with another u_i. By (13), u_i's cannot continue forever without violating the discipline, so repeating this extension process eventually either gives us a violation right after M_i (in which case we have produced a new $N^*\ R$ action, so we can discard everything after it) or lead to the u_i's being disabled, in which case we have succesfully turned the M_i action into an N action and again turned the extended execution into an execution that ends with $N^*\ R$. Repeating this for each M_i action, moving from right to left, produces the desired execution of the reduced system.

Theorem 1 *Let P be a finite set, and let i and j range over P. For all u_i and v_i, using definitions (4)–(11), if hypotheses (13)–(21) hold, then $p\ (u+v)^*\ q \leq N^*\ R$.*

Proof. The proof below is top-down; in other words, we prove lemmas used in a proof after the proof itself. Thus, within the proof of formula n, we may use only formulas with labels greater than n and results proved before now (*i.e.*, formulas with labels less than (22), the label on the top-level proof below). The top-level proof works as follows: push u's left (lines 1-2) where they are eliminated by the initial p (line 3), push M's to the left of R's (line 4), condense the R's to a single R (lines 5-6), and finally turn the M's into N's (lines 7-8).

(22) $p\ (u+v)^*\ q \leq N^*\ R$

$$\begin{array}{lcl}
p\ (u+v)^*\ q & \leq & \{v \leq M+R\ (23) \} \\
p\ (u+M+R)^*\ q & \leq & \{(M+R)\ u \leq (1+u)\ (M+R)\ (24); (3)\} \\
p\ u^*\ (M+R)^*\ q & \leq & \{p\ u^* \leq 1\ (25)\} \\
(M+R)^*\ q & \leq & \{R\ M \leq (M+1)\ R\ (27);\ (3)\} \\
M^*\ R^*\ q & \leq & \{R^* = 1+R\ (28)\} \\
M^*\ (1+R)\ q & \leq & \{(1+R)\ q \leq R\ (29)\} \\
M^*\ R & \leq & \{1 \leq N^*\} \\
M^*\ N^*\ R & = & \{M\ N^*\ R \leq N^*\ R\ (30);\ (\text{* ind L})\} \\
N^*\ R & &
\end{array}$$

□

A v is either an M or an R:

(23) $v \leq M+R$

$$\begin{array}{lcl}
v & = & \{(8)\} \\
(+i : v_i) & \leq & \{1 \leq {u_i}^*\} \\
(+i : v_i\ {u_i}^*) & = & \{v_i\ u_i^* = M_i\ (5)\} \\
(+i : M_i) & = & \{(9)\} \\
M & \leq & \{(\text{regular algebra})\} \\
M+R & &
\end{array}$$

□

A u moves to the left of an M or R (but may disappear in the process):

(24) $(M+R)\ u \leq (1+u)\ (M+R)$

$$\begin{array}{lcl}
(M+R)\ u & = & \{\text{(distributivity)}\} \\
M\ u + R\ u & \leq & \{R\ u \leq R\ (34)\} \\
M\ u + R & = & \{M = (+j : M_j)\ (9);\ u = (+i : u_i)\ (7)\} \\
(+j : M_j)\ (+i : u_i) + R & = & \{\text{(distributivity)}\} \\
(+i, j : M_j\ u_i) + R & \leq & \{M_j\ u_i \leq (p_i + u_i)\ (M_j + R))\ (38)\} \\
(+i, j : (p_i + u_i)\ (M_j + R)) + R & \leq & \{p_i \leq 1\} \\
(+i, j : (1 + u_i)\ (M_j + R)) + R & = & \{\text{(distributivity)}\} \\
(+i : 1 + u_i)\ (+j : M_j + R) + R & = & \{\text{(distributivity)}\} \\
(1 + (+i : u_i))\ ((+j : M_j) + R) + R & \leq & \{(+i : u_i) = u\ (7); (+j : M_j) = M\ (9)\} \\
(1 + u)\ (M + R) + R & = & \{R \leq (1 + u)\ (M + R)\} \\
(1 + u)\ (M + R) & &
\end{array}$$

□

A p swallows up u's to the right:

(25) $p\ u^* \leq 1$

$$\begin{array}{lcl}
p\ u^* & = & \{p\ u \leq p\ (26);\ (\text{* ind R})\} \\
p & \leq & \{\text{(Boolean algebra)}\} \\
1 & &
\end{array}$$

□

A p swallows up a single u to the right:

(26) $p\ u \leq p$

$$\begin{array}{lcl}
p\ u & = & \{u = (+i : u_i)\ (7)\} \\
p\ (+i : u_i) & = & \{\text{(distributivity)}\} \\
(+i : p\ u_i) & \leq & \{p \leq p_i\ (4),\ \text{(Boolean algebra)}\} \\
(+i : p_i\ u_i) & = & \{p_i\ u_i = 0\ (18)\} \\
(+i : 0) & = & \{\text{(distributivity)}\} \\
0 & \leq & \{\text{(regular algebra)}\} \\
p & &
\end{array}$$

□

An M moves left past an R (possibly disappearing in the process):

(27) $R\ M \leq (M + 1)\ R$

$$\begin{array}{lcl}
R\ M & \leq & \{(34)\} \\
R & \leq & \{1 \leq M + 1\} \\
(M + 1)\ R & &
\end{array}$$

□

A sequence of R's can be reduced to at most one R:

(28) $R^* = 1 + R$

$$\begin{array}{lll}
R^* & = & \{z^* = 1 + z + z\ z^* \qquad\qquad\} \\
1 + R + R\ R^* & = & \{R\ R \leq R\ (34);\ (^*\ \text{ind R})\} \\
1 + R + R & = & \{(\text{regular algebra}) \qquad\quad\} \\
1 + R & \leq & \{(\text{regular algebra}) \qquad\quad\} \\
R^* & &
\end{array}$$

Since the first and last terms are equal, the first and fourth terms are equal.
□

(29) $(1 + R)\ q \leq R$

$$\begin{array}{lll}
(1 + R)\ q & = & \{(\text{distributivity}) \} \\
q + R\ q & \leq & \{R\ q \leq R\ (34) \} \\
q + R & \leq & \{q = 1\ q \leq (1 + M)\ q \leq (1 + M)\ q\ \top = R\ (11)\} \\
R & &
\end{array}$$

□

An $N^*\ R$ action swallows up M_i actions to its left:

(30) $M_i\ N^*\ R \leq N^*\ R$

The following proof says that an $N^*\ R$ action can be used to generate u_i actions to its left until it either produces a discipline violation (q) or until it has produced enough u_i's to turn the M_i to its left into an N:

$$\begin{array}{lll}
M_i\ N^*\ R & \leq & \{N^*\ R \leq (u_i\ \overline{q})N^*\ R + (p_i + u_i\ q)N^*\ R\ (32); \} \\
 & & \{(\star\ \text{ind})\ \text{w. } x := n^*\ R,\ y := u_i\ \overline{q},\ z := (p_i + u_i\ q)N^*\ R\} \\
M_i\ (u_i\ \overline{q}) \star (p_i + u_i\ q)\ N^*\ R & \leq & \{(u_i\ \overline{q})^\omega = 0\ (13);\ \text{definition of } \star \} \\
M_i\ (u_i\ \overline{q})^*\ (p_i + u_i\ q)\ N^*\ R & \leq & \{\overline{q} \leq 1 \} \\
M_i\ u_i^*\ (p_i + u_i\ q)\ N^*\ R & \leq & \{M_i\ u_i^* = M_i\ (31) \} \\
M_i\ (p_i + u_i\ q)\ N^*\ R & = & \{(\text{distributivity}) \} \\
(M_i\ p_i + M_i\ u_i\ q)\ N^*\ R & \leq & \{M_i\ u_i \leq M_i\ u_i^* \leq M_i\ (31) \} \\
(M_i\ p_i + M_i\ q)\ N^*\ R & = & \{1 = \overline{q} + q \} \\
(M_i\ p_i\ (\overline{q} + q) + M_i\ q)\ N^*\ R & = & \{(\text{distributivity}) \} \\
(M_i\ p_i\ \overline{q} + M_i\ p_i\ q + M_i\ q)\ N^*\ R & \leq & \{p_i \leq 1 \} \\
(M_i\ p_i\ \overline{q} + M_i\ q)\ N^*\ R & \leq & \{M_i\ p_i\ \overline{q} = N_i\ (6);\ N_i \leq N(10) \} \\
(N + M_i\ q)\ N^*\ R & = & \{(\text{distributivity}) \} \\
N\ N^*\ R + M_i\ q\ N^*\ R & \leq & \{N\ N^* \leq N^*;\ N^*\ R \leq \top \} \\
N^*\ R + M_i\ q\ \top & \leq & \{M_i\ q\ \top \leq R\ (11) \} \\
N^*\ R + R & \leq & \{R \leq N^*\ R \} \\
N^*\ R & &
\end{array}$$

□

M_i actions swallow u_i actions to their right:

(31) $M_i\ u_i^* \leq M_i$

$$\begin{array}{lcl}
M_i\ u_i^* & = & \{M_i = v_i\ u_i^*\ (5)\} \\
v_i\ u_i^*\ u_i^* & = & \{u_i^*\ u_i^* = u_i^* \quad\} \\
v_i\ u_i^* & = & \{v_i\ u_i^* = M_i\ (5)\} \\
M_i & &
\end{array}$$

□

$N^*\ R$ can be used to generate (to its left) either a u_i or a p_i (indicating that u_i no longer has invisible operations to perform):

(32) $N^*\ R \le (u_i\ \overline{q})\ N^*\ R + (p_i + u_i\ q)\ N^*\ R$

$$\begin{array}{lcl}
N^*\ R & = & \{R = (1+M)\ q\ \top\ (11) \} \\
N^*\ (1+M)\ q\ \top & \le & \{1 \le p_i + u_i\ \top\ (19) \} \\
N^*\ (1+M)\ q\ (p_i+u_i)\ \top & = & \{(\text{distributivity}) \} \\
N^*\ (1+M)\ (q\ p_i + q\ u_i)\ \top & \le & \{q\ p_i = p_i\ q\ (\text{Boolean algebra}) \} \\
N^*\ (1+M)\ (p_i\ q + q\ u_i)\ \top & \le & \{q\ u_i \le u_i\ q\ (21) \} \\
N^*\ (1+M)\ (p_i\ q + u_i\ q)\ \top & = & \{(\text{distributivity}) \} \\
N^*\ (1+M)\ (p_i+u_i)\ q\ \top & = & \{(\text{distributivity}) \} \\
N^*\ ((p_i+u_i) + M\ (p_i+u_i))\ q\ \top & \le & \{M\ (p_i+u_i) \le (p_i+u_i)\ (M+R)\ (38) \} \\
N^*\ ((p_i+u_i) + (p_i+u_i)\ (M+R))\ q\ \top & = & \{(\text{distributivity}) \} \\
N^*\ (p_i+u_i)\ (1+M+R)\ q\ \top & \le & \{(1+M+R)\ q\ \top \le R\ (33) \} \\
N^*\ (p_i+u_i)\ R & \le & \{N\ (p_i+u_i) \le (p_i+u_i)\ (N+R)\ (35); \} \\
 & & \{(1)\ \text{ with } x := N,\ y := p_i+u_i,\ z := N+R\} \\
(p_i+u_i)\ (N+R)^*\ R & \le & \{R\ N \le R\ (34) \le (1+N)\ R;\ (3) \} \\
(p_i+u_i)\ N^*\ R^*\ R & = & \{R\ R \le R\ (34);\ (\text{* ind R}) \} \\
(p_i+u_i)\ N^*\ R & \le & \{1 = \overline{q} + q \} \\
(p_i + u_i\ (\overline{q} +\ q))\ N^*\ R & = & \{(\text{distributivity}) \} \\
(u_i\ \overline{q})\ N^*\ R + (p_i + u_i\ q)\ N^*\ R & &
\end{array}$$

□

(33) $(1+M+R)\ q\ \top \le R$

$$\begin{array}{lcl}
(1+M+R)\ q\ \top & = & \{(\text{distributivity}) \} \\
(1+M)\ q\ \top + R\ q\ \top & \le & \{R\ q\ \top \le R\ (34) \} \\
(1+M)\ q\ \top + R & \le & \{(1+M)\ q\ \top = R\ (11)\} \\
R & &
\end{array}$$

□

R's swallow up z's to the right:

(34) $R\ z \le R$

$$\begin{array}{lcl}
R\ z & = & \{R = (1+M)\ q\ \top\ (11)\} \\
(1+M)\ q\ \top\ z & = & \{\top\ z \le \top \} \\
(1+M)\ q\ \top & = & \{(1+M)\ q\ \top = R\ (11)\} \\
R & &
\end{array}$$

□

N commutes right past $(p_i + u_i)$ (possibly changing into an R):

(35) $N_j\ (p_i + u_i) \le (p_i + u_i)\ (N + R)$

$$\begin{array}{lll}
N_j\ (p_i + u_i) & \le & \{N_j = M_j\ p_j\ \overline{q}\ (6) \} \\
M_j\ p_j\ \overline{q}\ (p_i + u_i) & \le & \{\overline{q} \le 1 \text{ (Boolean algebra)} \} \\
M_j\ p_j (p_i + u_i) & = & \{\text{(distributivity)} \} \\
M_j\ (p_j\ p_i + p_j\ u_i) & \le & \{p_j\ p_i = p_i\ p_j \text{ (Boolean algebra)} \} \\
M_j\ (p_i\ p_j + p_j\ u_i) & \le & \{p_j\ u_i \le u_i\ p_j\ (37) \} \\
M_j\ (p_i\ p_j + u_i\ p_j) & = & \{\text{(distributivity)} \} \\
M_j\ (p_i + u_i)\ p_j & \le & \{M_j\ (p_i + u_i) \le (p_i + u_i)\ (M_j + R)\ (38)\} \\
(p_i + u_i)\ (M_j + R)\ p_j & = & \{\text{(distributivity)} \} \\
(p_i + u_i)\ (M_j\ p_j + R\ p_j) & \le & \{M_j\ p_j \le N + R\ (36) \} \\
(p_i + u_i)\ (N + R + R\ p_j) & \le & \{p_j \le 1 \text{ (Boolean algebra)} \} \\
(p_i + u_i)\ (N + R) & &
\end{array}$$

□

(36) $M_j\ p_j \le N + R$

$$\begin{array}{lll}
M_j\ p_j & = & \{1 = q + \overline{q} \text{ (Boolean algebra)} \} \\
M_j\ p_j\ (q + \overline{q}) & = & \{\text{(distributivity)} \} \\
M_j\ p_j\ q + M_j\ p_j\ \overline{q} & = & \{M_j\ p_j\ \overline{q} = N_j\ (6) \} \\
M_j\ p_j\ q + N_j & \le & \{N_j \le N\ (10) \} \\
M_j\ p_j\ q + N & \le & \{M_j \le M\ (9);\ p_j \le 1 \text{ (Boolean algebra)}\} \\
M\ q + N & \le & \{M\ q \le (1 + M)\ q\ \top = R\ (11) \} \\
R + N & &
\end{array}$$

□

u_i commutes left past p_j:

(37) $p_j\ u_i \le u_i\ p_j$

$i = j:\ (18);\ i \ne j : (14)$

□

M_j commutes to the right past $(p_i + u_i)$ (possibly changing into an R):

(38) $M_j\ (p_i + u_i) \le (p_i + u_i)\ (M_j + R)$

$$\begin{array}{lll}
i = j : & & \\
M_i\ (p_i + u_i) & \le & \{p_i \le 1 \text{ (Boolean algebra)} \} \\
M_i\ (1 + u_i) & \le & \{M_i = v_i\ {u_i}^*\ (5) \} \\
v_i\ {u_i}^*\ (1 + u_i) & = & \{{u_i}^*\ (1 + u_i) = {u_i}^* \} \\
v_i\ {u_i}^* & = & \{1 = p_i + \overline{p}_i \text{ (Boolean algebra)}\} \\
(p_i + \overline{p}_i) v_i\ {u_i}^* & = & \{\text{(distributivity)} \} \\
(p_i\ v_i + \overline{p}_i\ v_i)\ {u_i}^* & = & \{\overline{p}_i\ v_i = 0\ (20) \} \\
p_i\ v_i\ {u_i}^* & = & \{v_i\ {u_i}^* = M_i\ (5) \} \\
p_i\ M_i & = & \{p_i \le p_i + u_i;\ M_i \le M_i + R \} \\
(p_i + u_i)\ (M_i + R) & &
\end{array}$$

$i \neq j$:

$$\begin{array}{lll}
M_j\ (p_i + u_i) & = & \{M_j = v_j\ u_j^*\ (5) \} \\
v_j\ u_j^*\ (p_i + u_i) & \leq & \{i \neq j\ \Rightarrow\ u_j\ (p_i + u_i) \leq (p_i + u_i)\ [u_j]\ (39);\ (1)\} \\
v_j\ (p_i + u_i)\ [u_j]^* & = & \{[u_j]^* = [u_j^*]\ (41) \} \\
v_j\ (p_i + u_i)\ [u_j^*] & = & \{(\text{distributivity}) \} \\
(v_j\ p_i + v_j\ u_i)\ [u_j^*] & \leq & \{i \neq j\ \Rightarrow\ v_j\ p_i \leq p_i\ v_j\ (15) \} \\
(p_i\ v_j + v_j\ u_i)\ [u_j^*] & \leq & \{i \neq j\ \Rightarrow\ v_j\ u_i \leq u_i\ [v_j]\ (17) \} \\
(p_i\ v_j + u_i\ [v_j])\ [u_j^*] & \leq & \{v_j \leq [v_j]\ (12) \} \\
(p_i\ [v_j] + u_i\ [v_j])\ [u_j^*] & = & \{(\text{distributivity}) \} \\
(p_i + u_i)\ [v_j]\ [u_j^*] & \leq & \{[v_j]\ [u_j^*] \leq [v_j\ u_j^*]\ (40) \} \\
(p_i + u_i)\ [v_j\ u_j^*] & = & \{v_j\ u_j^* = M_j\ (5) \} \\
(p_i + u_i)\ [M_j] & = & \{[M_j] = M_j + q\ \top + M_j\ q\ \top\ (12) \} \\
(p_i + u_i)\ (M_j + q\ \top + M_j\ q\ \top) & = & \{(\text{distributivity}) \} \\
(p_i + u_i)\ (M_j + (1 + M_j)\ q\ \top) & \leq & \{M_j \leq M\ (9) \} \\
(p_i + u_i)\ (M_j + (1 + M)\ q\ \top) & = & \{(1 + M)\ q\ \top = R\ (11) \} \\
(p_i + u_i)\ (M_j + R) & &
\end{array}$$

□

(39) $i \neq j\ \Rightarrow\ u_j\ (p_i + u_i) \leq (p_i + u_i)\ [u_j]$

$$\begin{array}{lll}
u_j\ (p_i + u_i) & = & \{(\text{distributivity}) \} \\
u_j\ p_i + u_j\ u_i & = & \{i \neq j\ \Rightarrow\ u_j\ p_i = p_i\ u_j\ (14) \} \\
p_i\ u_j + u_j\ u_i & = & \{i \neq j\ \Rightarrow\ u_j\ u_i \leq u_i\ [u_j]\ (16) \} \\
p_i\ u_j + u_i\ [u_j] & \leq & \{u_j \leq u_j + q\ \top + u_j\ q\ \top = [u_j]\ (12)\} \\
p_i\ [u_j] + u_i\ [u_j] & = & \{(\text{distributivity}) \} \\
(p_i + u_i)\ [u_j] & &
\end{array}$$

□

(40) $[x]\ [y^*] \leq [x\ y^*]$

$$\begin{array}{lll}
[x]\ [y^*] & = & \{[x] = x + q\ \top + x\ q\ \top\ (12) \} \\
(x + q\ \top + x\ q\ \top)\ [y^*] & = & \{(\text{distributivity}) \} \\
x\ [y^*] + q\ \top\ [y^*] + x\ q\ \top\ [y^*] & \leq & \{\top\ [y^*] \leq \top \} \\
x\ [y^*] + q\ \top + x\ q\ \top & = & \{[y^*] = y^* + q\ \top + y^*\ q\ \top\ (12) \} \\
x\ (y^* + q\ \top + y^*\ q\ \top) + q\ \top + x\ q\ \top & = & \{(\text{distributivity}) \} \\
x\ y^* + x\ q\ \top + x\ y^*\ q\ \top + q\ \top & \leq & \{1 \leq y^*,\ \text{ so }\ x\ q\ \top \leq x\ y^*\ q\ \top\} \\
x\ y^* + x\ y^*\ q\ \top + q\ \top & = & \{(12) \} \\
[x\ y^*] & &
\end{array}$$

□

(41) $[x]^* = [x^*]$

$$\begin{array}{lll}
[x^*] & = & \{[x^*] = x^* + q\ \top + x^*\ q\ \top\ (12)\} \\
x^* + q\ \top + x^*\ q\ \top & = & \{x \le [x]\ (12);\ q\ \top \le [x]\ (12)\ \} \\
[x]^* + [x] + [x]^*\ [x] & \le & \{\text{(regular algebra)}\ \} \\
[x]^* & \le & \{1 \le x^* \le [x^*]\ (12)\ \} \\
[x^*]\ [x]^* & = & \{[x^*]\ [x] \le [x^*]\ (42);\ (1)\ \} \\
[x^*] & &
\end{array}$$

Since the first and last terms are equal, the first and fourth terms are equal.
□

(42) $[x^*]\ [x] \le [x^*]$

$$\begin{array}{lll}
[x^*]\ [x] & = & \{[x^*] = x^* + q\ \top + x^*\ q\ \top\ (12)\} \\
(x^* + q\ \top + x^*\ q\ \top)\ [x] & = & \{\text{(distributivity)}\ \} \\
x^*\ [x] + q\ \top\ [x] + x^*\ q\ \top\ [x] & \le & \{\top\ z \le \top\ \} \\
x^*\ [x] + q\ \top + x^*\ q\ \top & = & \{[x] = x + q\ \top + x\ q\ \top\ (12)\ \} \\
x^*\ (x + q\ \top + x\ q\ \top) + q\ \top + x^*\ q\ \top & = & \{\text{(distributivity)}\ \} \\
x^*\ x + x^*\ q\ \top + x^*\ x\ q\ \top + q\ \top + x^*\ q\ \top & = & \{x^*\ x \le x^*\ \} \\
x^* + x^*\ q\ \top + q\ \top & = & \{(12)\ \} \\
[x^*] & &
\end{array}$$

□

5 System Model

We define a simple model of concurrent systems that use mutual exclusion for access to selected variables, and we prove that our reduction theorem applies to these systems. This model is intended to be the simplest one that retains all relevant aspects of concurrent programming languages, such as Java. It can be modified and generalized in numerous ways with little effect on our results.

Each shared variable is classified as *protected* or *unprotected.* There are no constraints on how unprotected variables are accessed. The synchronization discipline requires that mutual exclusion be used for access to protected variables. Any combination of synchronization mechanisms (locks, condition variables, semaphores, barriers, *etc.*) can be used to provide the mutual exclusion, provided the scheme can be captured by *exclusive access predicates*, as described in Section 1.

Formally, a system is a tuple $\langle \Theta, V_{unsh}, V_{prot}, V_{unprot}, T, I, e \rangle$ where

Θ is a set of threads (thread identifiers). i and j range over Θ.

V_{unsh} is a set of unshared variables, *i.e.*, variables that appear in transitions of at most one thread.

V_{prot} is a set of variables declared (possibly incorrectly) to be protected, *i.e.*, there are synchronization mechanisms that ensure mutual exclusion for accesses to these variables. For each variable $x \in V_{prot}$ and each thread i, there is an exclusive access predicate e_i^x.

V_{unprot} is a set of (possibly shared) variables, called unprotected variables. No assumptions are made regarding synchronization for accesses to them.

$T = \bigcup_i T_i$ is a set of transitions, where T_i is the set of transitions of thread i. Let $V = V_{unsh} \cup V_{prot} \cup V_{unprot}$ and $V_{guard} = V_{unsh} \cup V_{unprot}$. A transition t is a guarded command $g \to c$, where the guard g is a predicate over V_{guard}, and c is built from assignments over V, sequential composition, and conditionals (if-then and if-then-else).

I is a predicate over V. I characterizes the initial states.

e is a family of (possibly incorrect) exclusive access predicates e_i^x over V.

Our results do not depend on the details of the expression language. The expression language may support references (pointers) to variables. Expressions in commands may be non-deterministic, *e.g.*, expressions like choose(S), which returns a non-deterministically selected element of the non-empty set S.

Guards are used for synchronization (blocking). Conditionals in commands are used for sequential control flow. For convenience of analysis, protected variables cannot appear in guards. This is reasonable because the synchronization mechanisms that protect the protected variables, not the protected variables themselves, should be used to achieve the necessary synchronization. If necessary, the value of a protected variable v can be copied into an unshared or unprotected variable, and the latter variable can be used in a guard, or v can be moved from V_{prot} to V_{unprot} and used in a guard directly (this last approach may make more transitions visible, as defined below, and thereby decrease the benefit of the reduction).

A *state* is a mapping from variables to values. Let Σ be the set of states. We also use states as maps from expressions to values, with the usual meaning (homomorphic extension).

A transition t is *enabled* in state s if its guard is true in s. We interpret a transition t as a binary relation $[\![t]\!]$ over Σ in the usual way: $\langle s, s' \rangle \in [\![t]\!]$ iff t is enabled in s and execution of t from s can lead to s'. An *execution* is a finite or infinite sequence σ of states such that $\sigma(0)$ satisfies I and every pair of consecutive states in σ is in $[\![t]\!]$ for some transition t.

A transition t of thread i is *invisible* if (i) for each unprotected variable x, each operation op on x in t left-commutes with all operations op' on x in transitions of other threads ($i.e.$, $op'\,op \le op\,op'$), and (ii) for each protected variable x, either (ii-a) t cannot change the value of any exclusive access predicate for x, or (ii-b) t cannot truthify any exclusive access predicate for x, and the exclusive access predicates for x are disjoint in all reachable states. Other transitions are *visible*. Let $T_i^{\rm vis}$ and $T_i^{\rm invis}$ be the sets of visible and invisible transitions in T_i, respectively. The visible and invisible transition relations and the predicate p_i are

$$(43)\quad v_i = \bigcup_{t \in T_i^{\rm vis}} [\![t]\!] \qquad u_i = \bigcup_{t \in T_i^{\rm invis}} [\![t]\!] \qquad v = \bigcup_i v_i \qquad u = \bigcup_i u_i$$

$$(44)\quad p_i = \text{all transitions in } T_i^{\rm invis} \text{ are disabled}$$

A system is well-formed if the following conditions hold.

WF-disjoint. V_{unsh}, V_{prot}, and V_{unprot} are disjoint.

WF-initVis. The initial transitions of each thread are visible, *i.e.*, $I \Rightarrow p$. This ensures that the conclusion of the reduction theorem applies to all reachable states of the original system.

WF-sep. Visible and invisible transitions of each thread are separate, *i.e.*, cannot be executed from the same state. Formally, $(\forall i : \text{domain}(u_i) \cap \text{domain}(v_i) = \emptyset)$.

WF-acc. Internal non-determinism in a transition (*i.e.*, non-deterministic choices that do not affect the ending state) does not affect the set of variables accessed by the transition or the order in which those variables are first accessed. This ensures well-definedness of *acc* in Section 6 and of x in case 2 of the proof of (17) in Section 7.

WF-finiteInvis. No thread has an infinite execution sequence containing only invisible transitions. Formally, $(\forall i : u_i^{\omega} = \emptyset)$.

WF-initExcl. For each protected variable x, the exclusive access predicates for x are initially disjoint, *i.e.*, $I \Rightarrow \text{disjoint}(e^x)$, where $\text{disjoint}(e^x) = \neg(\exists i, j : i \neq j \wedge e_i^x \wedge e_j^x)$.

WF-endExcl. A thread cannot take away another thread's exclusive access to a variable. Formally, for an exclusive access predicate e_i^x and $j \neq i$, transitions of thread j cannot falsify e_i^x.

Assuming V_{unsh} and the exclusive access predicates are chosen appropriately, all transition systems that are reasonable models of Java programs satisfy these conditions, except possibly WF-finiteInvis, WF-initExcl, and WF-endExcl. One approach to choosing V_{unsh} is to use an automatic and conservative static analysis, such as [WR99]. Another approach is to allow the user to specify V_{unsh} and check correctness of the classification during state-space exploration of the reduced system; as mentioned in Section 1, a similar idea is used in Spin [HP95] and can easily be proved sound. In practice, WF-finiteInvis is approximated by aborting state-space exploration if a thread consumes an excessive amount of CPU time without executing a visible transition. WF-initExcl and WF-endExcl are typically easy to check.

6 Mutual-Exclusion Synchronization Discipline

The synchronization discipline requires that, for every variable $x \in V_{prot}$, (*i*) a transition of thread i executed from a state s may access x only if $s \models e_i^x$, and (*ii*) $\text{disjoint}(e^x)$ holds in every reachable state.

Let $acc(s_1, t, s_2)$ denote the set of protected variables accessed by execution of transition t from state s_1 to s_2. The set of accessed variables may depend on which branches of conditionals are taken. The ending state s_2 is included as an argument to *acc* because t may be non-deterministic. WF-acc ensures that *acc* is well-defined. Since guards do not contain protected variables, $acc(s_1, t, s_2) = \emptyset$ if t is disabled in s_1 (otherwise, $acc(s_1, t, s_2)$ would be the set of protected variables in t's guard).

We augment the system with a predicate q that holds iff the synchronization discipline has been violated. Formally, q is the least predicate that satisfies

$$(45)\quad \forall i : \forall t \in T_i : \forall \langle s_1, s_2 \rangle \in [\![t_i]\!] : s_2 \models q \iff \\ (\exists x \in V_{prot} : (x \in acc(s_1, t, s_2) \ \wedge \ s_1 \not\models e^x_i) \ \vee \ s_2 \not\models \text{disjoint}(e^x)) \ \vee \ s_1 \models q.$$

The third disjunct in (45) implies that q is monotonic, *i.e.*, it can be truthified but not falsified.

Maintaining q involves accesses to q and accesses to variables that occur in exclusive access predicates. These accesses are ignored when determining $acc(s_1, t, s_2)$.

7 Proof that the Reduction Theorem Applies to the Mutual-Exclusion Synchronization Discipline

Section 1 explains informally why the main hypotheses of the reduction theorem hold for the mutual-exclusion synchronization discipline; in particular, hypotheses (14)-(15), (16)-(17), and (21) correspond to the informal hypotheses (*i*), (*ii*), and (*iii*) discussed in Section 1. Detailed proofs of these hypotheses appear below.

The other formal hypotheses of the reduction theorem follow directly from the definitions and well-formedness conditions in Section 5. In particular, hypothesis (13) follows directly from the well-formedness condition WF-finiteInvis. Hypotheses (18)-(19) say that p_i holds iff u_i is disabled; this follows directly from the definitions of p_i and u_i. Hypothesis (20) follows directly from WF-sep.

Proof of (14) and (15). These formulas say that a (invisible or visible) transition t_j of thread j cannot change the truth value of p_i. t_j could change the truth value of p_i only by updating some variable that is used in t_j's command and in the guard g_i of an invisible transition t_i of thread i. We show that no such variable exists. A protected variable cannot appear in a guard. An unshared variable cannot appear in both t_j and g_i. An unprotected variable cannot appear in g_i, because t_i is invisible.

Proof of (16). This is a corollary of (17), because every invisible transition could be classified as visible: an invisible transition is just a visible transition that satisfies some additional restrictions.

Proof of (17). Let t_i be an invisible transition of thread i, and let t_j be a visible transition t_j of thread j, and let s_1, s_2, and s_3 be states such that $\langle s_1, s_2 \rangle \in t_j$ and $\langle s_2, s_3 \rangle \in t_i$. Let $t_i = g_i \rightarrow c_i$ and $t_j = g_j \rightarrow c_j$. t_j does not enable t_i, because each variable x accessed by both t_j and the guard of t_i must be unprotected (recall that protected variables cannot appear in guards) and operations on x in t_i left-commute with operations on x in t_j. t_i does not disable t_j, for analogous reasons. Thus, there exist states s'_2 and s'_3 such that $\langle s_1, s'_2 \rangle \in t_i$ and $\langle s'_2, s'_3 \rangle \in t_j$. Transitions may be non-deterministic, so s'_2 and s'_3 are not uniquely determined by these conditions. It suffices to show that s'_2 and s'_3 can be chosen so that one of the following conditions (which correspond to the summands in (17)) holds: (*i*) $s'_2 \models q$, (*ii*) $s'_3 = s_3$ (*i.e.*, c_i left-commutes with c_j), or (*iii*) $s'_3 \models q$. Let $A = acc(s_1, t_j, s_2) \cap acc(s_1, t_i, s'_2)$.

case 1: $A = \emptyset$. This implies that

(46) $acc(s_1, t_j, s_2) = acc(s'_2, t_j, s'_3) \wedge acc(s_2, t_i, s_3) = acc(s_1, t_i, s'_2)$,

because the same branches of conditionals will be executed from either source state. This and $A = \emptyset$ imply that $(\forall x \in acc(s_2, t_i, s_3) : s_1(x) = s_2(x))$ and $(\forall x \in acc(s_1, t_j, s_2) : s_1(x) = s'_2(x))$. Thus, by resolving non-determinism (if any) in the transitions in the same way when executing t_i followed by t_j as when executing t_j followed by t_i to reach s_3, we obtain $s'_3(v) = s_3(v)$ for all variables $v \in V \setminus \{q\}$. We must exclude q here because *acc* does not reflect accesses used to update q, as stated in Section 6.

case 1.1: $s_3 \models \bar{q}$. If $s'_3 \models \bar{q}$, then $s'_3 = s_3$, *i.e.*, condition (*ii*) holds. If $s'_3 \models q$, then condition (*iii*) holds.

case 1.2: $s_3 \models q$. We show that $s'_2 \models q$ or $s'_3 \models q$.

case 1.2.1: $s_1 \models q$. This and monotonicity of q imply $s'_3 \models q$.

case 1.2.2: $s_1 \models \bar{q}$. This and $s_3 \models q$ imply that the synchronization discipline is violated either by execution of t_j from s_1 or by execution of t_i from s_2. The violation corresponds to the first or second disjunct in (45) being true (the third disjunct just makes q monotonic). Thus, there are 2×2 cases to consider.

case 1.2.2.1: $(\exists x \in V_{prot} : x \in acc(s_1, t_j, s_2) \wedge s_1 \not\models e^x_j)$. (46) implies $x \in acc(s'_2, t_j, s'_3)$. t_i is invisible, so it cannot truthify e^x_j, so $s'_2 \not\models e^x_j$. Thus, the definition of q implies $s'_3 \models q$.

case 1.2.2.2: $(\exists x \in V_{prot} : x \in acc(s_2, t_i, s_3) \wedge s_2 \not\models e^x_i)$. (46) implies $x \in acc(s_1, t_i, s'_2)$. WF-endExcl implies t_j did not falsify e^x_i, so $s_1 \not\models e^x_i$. Thus, the definition of q implies $s'_2 \models q$.

case 1.2.2.3: $(\exists x \in V_{prot} : s_2 \not\models \text{disjoint}(e^x))$. t_i is invisible, and the exclusive access predicates for x are not disjoint in some reachable state, so t_i must satisfy condition *ii-a* in the definition of invisible transition, so t_i cannot falsify any exclusive access predicate for x, so $s_3 \not\models \text{disjoint}(e^x)$. s_3 and s'_3 have the same values for all variables except q, so $s'_3 \not\models \text{disjoint}(e^x)$. Thus, the definition of q implies $s'_3 \models q$.

case 1.2.2.4: $(\exists x \in V_{prot} : s_3 \not\models \text{disjoint}(e^x))$. s_3 and s'_3 have the same values for all variables except q, so $s'_3 \not\models \text{disjoint}(e^x)$. Thus, the definition of q implies $s'_3 \models q$.

case 2: $A \neq \emptyset$. Note that $A \subseteq V_{prot}$, because t_i does not access unprotected variables, and because t_i and t_j do not access each other's unshared variables. Let x be the variable in A first accessed by execution of t_j from s_1 to s_2.

case 2.1: $s_1 \models e^x_j$. By definition of A, $x \in acc(s_1, t_i, s'_2)$.

case 2.1.1: $s_1 \models \text{disjoint}(e^x)$. The hypotheses of cases 2.1 and 2.1.1, together with $i \neq j$, imply $s_1 \not\models e^x_i$. This and $x \in acc(s_1, t_i, s'_2)$ imply $s'_2 \models q$.

case 2.1.2: $s_1 \not\models \text{disjoint}(e^x)$. This and the definition of q imply $s_1 \models q$. This and monotonicity of q imply $s'_2 \models q$.

case 2.2: $s_1 \not\models e^x_j$. The definitions of A and x imply that we can choose s'_2 and s'_3 such that $x \in acc(s'_2, t_j, s'_3)$, because the first access to x by t_j precedes execution of conditionals in t_j whose conditions could be affected by execution of t_i from s_1. The only variables that can be accessed by both t_i and those conditionals are unprotected variables (in A), and operations on those variables in t_i left-commute with operations on them in t_j, because t_i is invisible. t_i cannot truthify e^x_j, again because t_i is invisible, so $s'_2 \not\models e^x_j$. Thus, the definition of q implies $s'_3 \models q$.

Proof of (21): This follows directly from monotonicity of q, discussed in Section 6.

8 How to Use the Reduction

A methodology for using the reduction is as follows.

1. Guess the set V_{prot} of protected variables and the exclusive access predicates e^x_i. These guesses determine visibility of transitions and hence define a reduced system, in which the transition relation of thread i is N_i, defined in (6).

2. Augment the reduced system with a predicate q, as described in Section 6.

3. Check whether $\bar{q}$ holds in all reachable states of the reduced system. Check this using your favorite technique: model checking, theorem proving, hand waving, *etc.*

4. If so, then the reduction theorem implies that $\bar{q}$ holds in all reachable states of the original system, *i.e.*, the guesses in Step 1 are correct. Traditional reduction theorems can now be used to infer other properties of the original system from properties of the reduced system.

5. If not, then for some variable x in V_{prot}, the reduced system has a reachable state in which the mutual-exclusion synchronization discipline for x is violated. Revise the guess for e^x (using the path to the violation as a guide) or re-classify x as unprotected, and then return to Step 1.

A user who is uncertain in Step 1 whether a variable is protected or what its exclusive access predicates are, has two options. One option is to go ahead and guess. In the best case, the guess will be correct; in the worst case, a violation will be reported, and the incorrect guess will be revised or eliminated in Step 5. Another option is to declare the variable to be unprotected; this makes the reduction less effective (because more transitions are visible), but eliminates the need to guess the exclusive access predicates.

9 How to Use the Reduction Automatically for Systems with Monitors

The methodology in Section 8 is automatic except that the user must guess V_{prot} and the exclusive access predicates. For systems that use monitors for synchronization, this step, too, can be automated. We give a model of Java's built-in monitors and then describe how to automatically guess exclusive access predicates for variables protected by monitors.

9.1 Monitors

Java provides five built-in synchronization operations based on the classic operations on monitors: `acquire`, `release`, `wait`, `notify`, and `notifyAll`. A recursive lock and a condition variable are implicitly associated with each object. A *recursive lock* can be repeatedly acquired by a thread that holds it; the lock is free when each call to `acquire` has been matched by a call to `release`. In Java, `acquire` and `release` are implicitly invoked by synchronized methods and synchronized statements.

Every object has methods `wait()`, `notify()`, and `notifyAll()`. These methods throw `Illegal-MonitorStateException` if invoked by a thread that does not own the target object's lock; otherwise, they behave as follows. o`.wait()` adds the calling thread i to o's wait set (*i.e.*, the set of threads waiting on o), releases o's lock, and suspends i. When another thread notifies i (by invoking `notify` or `notifyAll`), i contends to re-acquire o's lock. When i acquires the lock, the invocation of o`.wait()` returns. o`.notify()` non-deterministically selects a thread i in o's wait set, removes i from the set, and notifies i. If o's wait set is empty, o`.notify()` has no effect. o`.notifyAll()` removes all threads from o's wait set and notifies each of them. A waiting thread i can also be awoken by a call to i`.interrupt`.

We deal only with untimed systems, so we do not consider bounded-time variants of `wait`. Also, we do not consider Java's controversial weakly consistent memory model.

Java's monitors can be represented in our system model by transitions with the forms in Figure 1, where g, c_1, and c_2 are not part of the monitor operations but rather are fragments of the surrounding program. Figure 1 does not model `Thread.interrupt` but can easily be extended to do so. Let m be a variable containing the state of a monitor, which is a record with four fields. $m.owner$ is *free* or the identity of the thread that owns m's lock. $m.depth$ is the number of unmatched acquire operations on m. $m.waiters$ is the set of threads waiting on m. $m.notified$ is the set of threads that were waiting on m, have been notified, and have not executed any transitions since the notification. In addition, $ldepth_i$ is an unshared variable of thread i, which holds the old value of $m.depth$ while thread i is waiting on m.

For an acquire, release, wait, notify, notifyAll, or waitResume transition of the form in Figure 1, we refer to the fragment not containing containing g, c_1, and c_2 as an acquire, release, wait, notify, notifyAll, or waitResume operation, respectively.

In order to classify transitions containing a release, notify, or notifyAll operation as invisible, we need to show that each such operation left-commutes with operations of other threads, and that the exclusive access predicates for a variable x protected by a monitor m are disjoint in all reachable states. The latter typically follows immediately from the structure of the predicates; for example, if $e_i^x = (m.owner = i)$, the predicates are clearly disjoint. For the former, let op_i be a release, notify, or notifyAll operation on monitor m by thread i, and let op_j be an operation on m by another thread j. We show that $op_j op_i \leq op_i op_j$, *i.e.*, if execution of $op_j op_i$ can lead from a state s to a state s', then execution of $op_i op_j$ can also lead from state s to s'. Note that the error command c_e in op_i accesses only unshared variables and therefore commutes with op_j. Each of the

```
acquire: $m.owner \in \{free, i\} \wedge g \ \rightarrow\ m.owner := i; m.depth := m.depth + 1; c_2$
release: $g \ \rightarrow\ c_1$;
              if $m.owner = i$ then
                  if $m.depth = 1$ then $m.owner := free$;
                  $m.depth := m.depth - 1$
              else $c_e$;
              $c_2$
wait: $g \ \rightarrow\ c_1$;
          if $m.owner = i$ then
              $m.waiters := m.waiters \cup \{i\}; ldepth_i := m.depth$;
              $m.depth := 0; m.owner := free$;
          else $c_e$;
          $c_2$
notify: $g \ \rightarrow\ c_1$;
            if $m.owner = i$ then
                if $m.waiters \neq \emptyset$ then
                    $j :=$ choose($m.waiters$);
                    $m.notified := m.notified \cup \{j\}$;
                    $m.waiters := m.waiters \setminus \{j\}$;
            else $c_e$;
            $c_2$
notifyAll: $g \ \rightarrow\ c_1$;
               if $m.owner = i$ then
                   $m.notified := m.notified \cup m.waiters$;
                   $m.waiters := \emptyset$;
               else $c_e$;
               $c_2$
waitResume: $i \in m.notified \wedge m.owner = free \wedge g \ \rightarrow\ m.notified := m.notified \setminus \{i\}$;
                                                        $m.owner := i; m.depth := ldepth_i; c_2$
```

Figure 1: Forms of transitions for monitor operations, where m is a monitor and $i, j \in \Theta$. m and $ldepth_i$ may not appear in g, c_1, c_2, or c_e or in transitions with other forms. The command c_e (e is mnemonic for "error") throws an `IllegalMonitorStateException`; c_e may access only unshared variables.

three monitor operations that op_i could be has the property that, when executed from any state in which $m.owner \neq i$, it behaves in exactly the same way: it is a no-op on m, and it executes its error command c_e.

Suppose $m.owner \neq i$ in s. Then $m.owner \neq i$ also holds after execution of op_j from s, because no monitor operation sets the owner of a monitor to be the identifier of a thread other than the one executing the operation. Thus, whether op_i executes before or after op_j, op_i behaves exactly the same way and does not update m. This implies that op_i left-commutes with op_j.

Suppose $m.owner = i$ in s. Thus, $m.owner \neq j$ in s. By the same reasoning as above,

$m.owner \neq j$ also holds after execution of op_i from s. Note that op_j cannot be an acquire or waitResume operation, because op_j would be blocked in s, so $op_j\, op_i$ would not lead to any state s'. Each of the four monitor operations that op_j could be (namely, release$_j$, wait$_j$, notify$_j$, or notifyAll$_j$) has the property that, when executed from any state in which $m.owner \neq j$, it behaves in exactly the same way: it is a no-op on m, and it executes its error command c_e. Thus, whether op_j executes before or after op_i, it behaves exactly the same way and does not update m. This implies that op_i left-commutes with op_j.

9.2 Exclusive Access Predicates for Variables Protected By Monitors

Exclusive access predicates for variables protected by monitors typically have the form $e_i^x = eap_i^{x,m}$, where

$$(47)\quad eap_i^{x,m} \;=\; init_i^x \vee (i = m.owner \;\wedge\; \neg init^x)$$

$$(48)\quad init^x \;=\; (\exists i \in \Theta : init_i^x).$$

and where the *initialization predicate* $init_i^x$ holds while thread i is executing code that initializes x. Note that the lock protecting a variable does not need to be held while the variable is being initialized.

Initialization predicates for variables in systems that correspond to Java programs can be guessed automatically: $init_i^x$ holds when thread i's program counter is in the appropriate class initializer (for static fields) or the appropriate constructor invocation (for instance fields).

To use (47), we need to identify, for each variable x in V_{prot}, a monitor m that protects x. This can be done automatically by running a variant of the lockset algorithm [SBN+97] during state-space exploration of the reduced system.

10 Comparison to Traditional Partial-Order Methods

This section demonstrates that our method can outperform traditional partial-order methods even on simple systems that do not involve pointers, object references, *etc.* Consider a system with two threads that use monitors m_0 and m_1 as locks and use an integer variable y to implement a barrier. Let uppercase letters denote control points. Let *guard* $\rightarrow$ *stmt* denote a transition that blocks when *guard* is false and can execute *stmt* when *guard* is true. For $i \in \{0, 1\}$, the code for thread i is

$$(49)\quad {}^{A}m_0.\text{acquire}();\ {}^{B}x_0 := i;\ {}^{C}m_0.\text{release}();{}^{D}m_1.\text{acquire}();\ {}^{E}x_1 := i;\ {}^{F}m_1.\text{release}();$$
$$\quad {}^{G}y{+}{+};\ {}^{H}y = 2 \rightarrow \text{skip};\ {}^{I}x_i = i\ {}^{J}$$

In the initial state, $x_j = j$ and $y = 0$, and both threads are at control point A. x_j is a protected variable, with exclusive access predicate $e_i^{x_j} = (m_j.owner = i) \vee (y = 2 \wedge i = j)$. y is not protected.

This system has 106 reachable states. With the reduction in this paper, transitions that release locks or update x_0 or x_1 are invisible; other transitions are visible. The reachable states of the reduced system are the reachable states of the original system in which every thread is ready to perform a visible transition or is at its final control point. There are 38 such states.

Traditional partial-order methods based on persistent sets [God96] (or ample sets [CGP99]) can also significantly reduce the number of explored states but do not achieve the same benefits as our reduction. To illustrate this, we compare our method to selective search using persistent sets. To ensure an unbiased comparison, we compute persistent sets using the most precise algorithm in [God96], namely, the conditional stubborn set algorithm (CSSA). CSSA is a non-deterministic algorithm for computing persistent sets. It is parameterized by a dependence relation, called might-be-the-first-to-interfere-with (MBTFTIW), on operations. To further ensure an unbiased comparison, we maximize the effectiveness of CSSA by always resolving non-determinism in a way that yields a minimum-size persistent set, and by using the most precise MBTFTIW relations. For acquire and release, this is the relation in [Sto02, Fig. 3]. For accesses to y, this is the MBTFTIW relation derived from the dependence relation on operations in which increments to y are independent of each other, and an increment to y is dependent with the condition $y = 2$ only in states in which the increment changes the truth value of the condition.

The CSSA-based selective search explores 77 states. To illustrate why it explores more than the 38 states explored by our method, consider the reachable state s in which thread 0 is at control point D and thread 1 is at control point B. With the reduction in this paper, the transitions that update x_0 or x_1 are invisible, so the system passes through this invisible state by executing the enabled transition of thread 1; the enabled transition of thread 0 is not executed in s. In contrast, the selective search explores both enabled transitions in s.

Selective search using persistent sets can be improved by incorporating sleep sets, a partial-order reduction technique that exploits information about the history of the search [God96]. For this example, incorporating sleep sets reduces the number of explored states from 77 to 72. This is still significantly more than the 38 states explored using our reduction.

This example can be generalized to show our method outperforming selective search by an arbitrary amount: simply insert additional transitions that access x_0 before the transition m_0.release() in thread 0.

If the barrier were implemented using two separate variables y_0 and y_1, such that only thread i increments y_i, then CSSA would be able to determine that $\{1B\}$ is persistent in state s. This illustrates that CSSA is a static analysis that is more effective for systems with more explicit static structure. Our method works equally well with either implementation of barriers.

11 Experimental Results

We implemented the reduction, specialized for monitor-based synchronization with exclusive access predicates of the form (47), in Java PathFinder (JPF) [BHPV00]. Specifically, we augmented the scheduler with an inner loop that iterates over invisible transitions without allowing context switches before them, and the system uses an appropriate variant of the lockset algorithm [SBN$^+$97]

Program	None	LocalFinal	Optimistic	Optimistic/LocalFinal
Clean	5.03	2.43	1.79	0.74
HaltException	6.74	2.18	1.69	0.78
Elevator	> 1260	31.2	7.24	0.23
TSP	> 1600	> 1600	32.5	0.02

Table 1: Experimental results. The middle three columns show memory usage, in MB, of JPF with the indicated reduction. The rightmost column shows the ratio of memory usage with the two reductions.

to automatically identify which monitors, if any, protect each protected variable. The user supplies configuration files indicating which variables are unshared and which are unprotected; other variables are implicitly classified as protected.

Traditional partial-order reductions are not supported in the version of JPF that we modified, so a direct experimental comparison to them is not readily possible. Nevertheless, it is clear that they would be less effective for these programs than our reduction. Suppose Spin's partial-order reduction [HP95, Hol97], described briefly in Section 1, were used. For reads and writes to shared variables (excluding special variables such as the state of a lock or thread), that reduction will be effective only when it can determine that a shared variable will not be written by any thread in the future (because reads are independent with reads), which is non-trivial in the presence of dynamic thread creation. Even if an effective static analysis is used to determine this, it will have relatively little benefit in the above examples, because most shared variables, with the exception of final variables, are both read and written by all threads that access them. Thus, Spin's reduction would be about as effective as one that treats all accesses to unshared variables and all accesses to final variables as invisible, and other operations as visible. We implemented the latter reduction in JPF, with all stack-allocated (*i.e.*, method-local) variables classified as unshared (identifying unshared heap-allocated variables would require a static analysis that is significantly more complicated than the static analyses used in Spin's partial-order reduction). We refer to this reduction as the LocalFinal reduction.

We measured the benefit of the reduction for four programs. Clean [BHPV00, Figure 1] and HaltException [HP00] are small synchronization skeletons (i.e., they were obtained from real programs by deleting everything except the statements related to synchronization) supplied by the developers of JPF; they are about 50 and 100 lines of code (LOC), respectively. The other two programs, Elevator and TSP, were developed at ETH Zürich and used as benchmarks in [vPG01]. Elevator (350 LOC) is a simple discrete event simulator. TSP (590 LOC) is a parallel program to solve the traveling salesman problem; we ran it on the data file map4 that accompanies it. The lockset algorithm was used in all experiments. All of the configuration files were empty; thus, all variables were classified, without error, as protected.

Table 1 shows the memory usage in MB for JPF to explore the state space, looking for assertion violations. We enabled JPF's hash compaction, which is similar to Spin's bitstate hashing [Hol97], for all experiments. Elevator with no reduction was manually terminated after 24 hours (of CPU time); Elevator with either reduction finishes in less than half an hour. For TSP with no reduction or LocalFinal reduction, the JVM reported an out-of-memory error. The JVM's maximum heap size was set to 1600 MB. Experiments were run on a Sun Blade 1500 with 2 GB RAM.

For Clean and HaltException, our reduction has modest benefit (average 24% reduction in memory usage) over the LocalFinal reduction, because these toy programs perform little real concurrent computation, and most instructions are accesses to unshared variables. For Elevator, a slightly larger and more realistic program, our reduction has noticeably more benefit, reducing memory usage by 77% compared to LocalFinal. For TSP, an even larger and more computationally interesting program, our reduction reduces memory usage by 98% compared to LocalFinal.

More elaborate traditional partial-order reductions, such as persistent sets computed using CSSA combined with sleep sets, can achieve more reduction than Spin's algorithm, *e.g.*, by exploiting a given dependence relation on monitor operations, but still would not recognize or exploit properties of protected variables. Accesses to protected variables are much more frequent than monitor operations in Elevator, TSP, and most other Java programs, so the performance of these more elaborate traditional partial-order reductions would be much closer to the performance of the LocalFinal reduction than to the performance of our optimistic reduction.

Acknowledgments. We thank Shaz Qadeer for telling us about exclusive access predicates, Liqiang Wang for doing the experiments with JPF, and Patrice Godefroid for insightful comments about partial-order methods.

References

[BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE Int'l. Conference on Automated Software Engineering (ASE)*, pages 3–12, September 2000.

[CGP99] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[CL98] E. Cohen and L. Lamport. Reduction in TLA. In *Proc. 9th Int'l. Conference on Concurrency Theory (CONCUR)*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[Coh00] E. Cohen. Separation and reduction. In *Proc. 5th Int'l. Conference on Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

[FF01] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96. ACM Press, June 2001.

[FFQ02] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. European Symposium on Programming (ESOP)*, pages 262–277, 2002.

[FQS02] C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *Proc. 14th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 180–194. Springer-Verlag, 2002.

[FQ03] Cormac Flanagan and Shaz Qadeer. Transactions for software model checking. In *Proc. 2nd Workshop on Software Model Checking*, volume 89(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[Hol97] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HP95] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. 7th Int'l. Conference on Formal Description Techniques (FORTE '94)*, pages 197–211. Chapman & Hall, 1995.

[Koz94] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[Lip75] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[SBN$^+$97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[SC02] S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. Technical Report DAR-02-8, SUNY at Stony Brook, Computer Science Dept., August 2002. Available at www.cs.sunysb.edu/~stoller/optimistic.html.

[SC03] S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of Lecture Notes in Computer Science, pages 489–504. Springer-Verlag, April 2003.

[Sto02] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.

[Val97] A. Valmari. Stubborn set methods for process algebras. In D. Peled, V. R. Pratt, and G. J. Holzmann, editors, *Proc. Workshop on Partial Order Methods in Verification*, volume 29 of *DIMACS Series*, pages 213–231. American Mathematical Society, 1997.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, October 2001.

[WR99] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. ACM Conf. on Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 187–206. ACM Press, October 1999.