Optimistic Synchronization-Based State-Space Reduction

Scott D. Stoller*[1] and Ernie Cohen**[2]

[1] State University of New York at Stony Brook
[2] Microsoft Research, Cambridge, UK

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 within sequences of 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 within sequences of transitions that access only unshared variables and protected variables. The model-checking experiments reported in [Sto02] are

* The author gratefully acknowledges the support of NSF under Grant CCR-9876058 and the support of ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Address: Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400. Email: stoller@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~stoller/

** Email: ernie.cohen@acm.org

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 significantly decrease the computational cost of the automated theorem-proving needed for thread-modular verification [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 [BR01,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 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. If this were the case, then the reduction would be 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.

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.

The reduction in [Sto02] is similar in spirit to the one in this paper. The main contributions of this paper relative to [Sto02] are: (1) 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); (2) separation of a general reduction theorem that justifies checking hypotheses of traditional reduction theorems in the reduced system from the application of this technique to mutual-exclusion synchronization disciplines; (3) allowing non-determinism in invisible transitions (in the notation of Section 3, [Sto02] requires that u be deterministic); (4) significantly shorter and cleaner proofs, based on ω-algebra. The first author initially tried to prove similar results in a transition-system framework, like the one in [God96]; that should be possible, but our experience suggests that the algebraic framework facilitates the task.

Operations on monitors are not analyzed specially in this paper. As a result, for systems that mainly use monitors for synchronization, this reduction is not as effective as the one in [Sto02]. It should be possible to integrate the specialized treatment of monitor operations in [Sto02] into this paper's broader framework.

Our method and traditional partial-order methods (*e.g.*, ample sets [CGP99], stubborn sets [Val97], 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 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 also exploits 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.

Traditional partial-order methods rely on static analysis to conservatively determine dependencies between transitions. As a result, those methods are less effective for programs that contain references (or pointers) and arrays, because static analysis cannot in general determine exactly which locations are accessed by each transition, and the static analysis of dependencies between transitions is correspondingly imprecise. Our method does not rely on conservative static analysis of dependencies and has no difficulty with references, *etc.*

2 Omega Algebra

An *omega algebra* is an algebraic structure over the operators (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[3]:

$$
\begin{array}{rl}
(x+y)+z = x+(y+z) & x \le y \Leftrightarrow x+y = y \\
x+y = y+x & \\
x+x = x & x^* = 1 + x + x^*\ x^* \\
0+x = x & x\ y \le x \Rightarrow x\ y^* = x \qquad (^* \text{ ind}) \\
x\ (y\ z) = (x\ y)\ z & x\ y \le y \Rightarrow x^*\ y = y \qquad (^* \text{ ind}) \\
0\ x = x\ 0 = 0 & \\
1\ x = x\ 1 = x & x \star y = x^\omega + x^*\ y \\
x\ (y+z) = x\ y + x\ z & x^\omega = x\ x^\omega \\
(x+y)\ z = x\ z + y\ z & x \le y\ x+z \Rightarrow x \le y \star z \qquad (\star \text{ ind})
\end{array}
$$

In parsing formulas, $\cdot$ and $\star$ associate to the right; e.g., $u\ v \star x \star y$ parses to $(u \cdot (v^\omega + v^* \cdot (x^\omega + x^* \cdot y)))$. In proofs, we use the hint "(dist)" to indicate application of the distributivity laws, and the hint "(hyp)" to indicate the use of hypotheses. If x_i is a finite collection of terms, 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. (Completeness holds only for *standard* terms, where the first arguments to $\cdot$, $^\omega$, and $\star$ are regular.) Thus, we make free use, without

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

proof, of familiar equations from the theory of (omega-)regular languages (e.g., $x^*\ x^* = x^*$).

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 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. Common properties of Boolean algebras (e.g., $p\ q = q\ p$) are used silently in proofs, as is the fact $x\ p\ y = 0 \implies x\ y = x\ \overline{p}\ y$.

The omega algebra axioms support several interesting programming models, where (intuitively) 0 is magic[4], 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 set of states in their domain (i.e., the states from which they can be executed). We define $\top = 1^\omega$; it is easy to see that $\top$ is the maximal element under $\leq$, and in the relational model, it relates all pairs of states.

In addition to equational identities of regular languages, we will use the following two standard theorems (proofs of these theorems and more sophisticated theorems of this type appear in [Coh00]):

(1) $\quad x\ y \leq y\ z \implies x^*\ y \leq y\ z^*$

(2) $\quad y\ x \leq x\ y \implies (x + y)^* \leq x^*\ y^*$

3 A Reduction Theorem

We consider systems composed of a fixed, finite 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[5], 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

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

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

of process i might modify variables that are either local to process i or protected by locks held by process i, 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, *i.e.*, $p = (\cdot i : p_i)$. 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 (3)–(9) 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.)

(3) $M_i = v_i \; u_i^*$

(4) $N_i = M_i \; p_i \; \overline{q}$

(5) $u = (+i : u_i)$

(6) $v = (+i : v_i)$

(7) $M = (+i : M_i)$

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

(9) $R = (1 + M) \; q \; \top$

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, (19), is

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

The hypotheses of our reduction theorem are as follows, formalized in formulas 11–(18) below. It is impossible to execute invisible actions of a single

process forever without violating the discipline (11). An action cannot enable or disable an invisible action of another process (12),(13), and in the absence of a discipline violation, it commutes to the right of such an action (14),(15). Visible and invisible actions of a process cannot be simultaneously enabled (16). u_i is enabled whenever p_i is *false* (17). Invisible actions cannot hide violations of the discipline (18).

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

(12) $\quad i \neq j \implies u_j\ p_i = p_i\ u_j$

(13) $\quad i \neq j \implies v_j\ p_i = p_i\ v_j$

(14) $\quad i \neq j \implies u_j\ u_i \leq u_i\ (q\ \top + u_j + u_j\ q\ \top)$

(15) $\quad i \neq j \implies v_j\ u_i \leq u_i\ (q\ \top + v_j + v_j\ q\ \top)$

(16) $\quad \overline{p}_i\ v_i = 0 = p_i\ u_i$

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

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

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 (18), the conditions above are all monotonic in q. Thus, if all the conditions above (including (18)) 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 (12) and (13) imply N_j does not affect enabledness of u_i, and (16) 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 (11), 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.

We now turn to the formal proof of the reduction theorem (19). We 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):

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

$$
\begin{array}{lcl}
p\ (u+v)^*\ q & \leq & \{v \leq M \leq M+R \} \\
p\ (u+M+R)^*\ q & \leq & \{(M+R)\ u \leq (1+u)\ (M+R)\ (20); (2)\} \\
p\ u^*\ (M+R)^*\ q & \leq & \{p\ u = 0\ (16); p \leq 1 \} \\
(M+R)^*\ q & \leq & \{R\ M \leq R;\ (2) \} \\
M^*\ R^*\ q & \leq & \{R\ R \leq R,\ \text{ so } R^* = (1+R) \} \\
M^*\ (1+R)\ q & \leq & \{(1+R)\ q = R \} \\
M^*\ R & \leq & \{1 \leq N^* \} \\
M^*\ N^*\ R & = & \{M\ N^*\ R \leq N^*\ R\ (21);\ (^* \text{ ind}) \} \\
N^*\ R &&
\end{array}
$$

(20) says that a u moves to the left of an M or R (but may disappear in the process):

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

$$
\begin{array}{lcl}
(M+R)\ u & \leq & \{(\text{dist}) \} \\
M\ u + R\ u & \leq & \{R = R\ \top,\ \text{so } R\ u = R\ \top\ u \leq R\ \top = R\} \\
M\ u + R & = & \{(7), (5), (\text{dist}) \} \\
(+i, j : M_j\ u_i) + R & \leq & \{M_j\ u_i \leq (1+u_i)\ (M_j + R))\ (25) \} \\
(+i, j : (1+u_i)\ (M_j + R)) + R & \leq & \{(7), (5), (\text{dist}) \} \\
(1+u)\ (M+R) &&
\end{array}
$$

(21) shows that $N^*\ R$ actions act as a factory for u_i actions until they either produce a discipline violation (q) or until they produce enough u_i's to turn the M_i to their left into an N.

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

$$
\begin{array}{lcl}
M_i\ N^*\ R & \leq & \{N^*\ R \leq (u_i\ \overline{q}) N^*\ R + (p_i + u_i\ q) N^*\ R \} \\
 & & \{(22); (\star \text{ ind}) \} \\
M_i\ (u_i\ \overline{q}) \star (p_i + u_i\ q)\ N^*\ R & \leq & \{(u_i\ \overline{q})^\omega = 0\ (11) \} \\
M_i\ (u_i\ \overline{q})^*\ (p_i + u_i\ q)\ N^*\ R & \leq & \{\overline{q} \leq 1;\ M_i\ u_i^* = M_i\ (3) \} \\
M_i\ (p_i + u_i\ q)\ N^*\ R & = & \{(\text{dist}) \} \\
(M_i\ p_i + M_i\ u_i\ q)\ N^*\ R & \leq & \{M_i\ u_i \leq M_i\ (3);\ M_i\ p_i \leq M_i\ p_i\ \overline{q} + M_i\ q\} \\
(M_i\ p_i\ \overline{q} + M_i\ q)\ N^*\ R & \leq & \{M_i\ p_i\ \overline{q} = N_i\ (4);\ N_i \leq N (8) \} \\
(N + M_i\ q)\ N^*\ R & \leq & \{(\text{dist}) \} \\
N\ N^*\ R + M_i\ q\ N^*\ R & \leq & \{M_i\ q\ N^*\ R \leq M_i\ q\ \top \leq R\ (9) \} \\
N\ N^*\ R + R & \leq & \{N\ N^* \leq N^*,\ 1 \leq N^* \} \\
N^*\ R &&
\end{array}
$$

(22) and (23) show that $N^*\ R$ generates a u_i (unless p_i already holds, or the discipline has already been violated).

(22) $N^*\ R \leq (u_i\ \overline{q})\ N^*\ R + (p_i + u_i\ q)\ N^*\ R$

$$
\begin{array}{lcl}
N^*\ R & \leq & \{N^*\ R \leq (p_i + u_i)\ N^*\ R (23)\} \\
(p_i + u_i)\ N^*\ R & \leq & \{u_i = u_i\ \overline{q} + u_i\ q \quad\} \\
(p_i + u_i\ \overline{q} + u_i\ q)\ N^*\ R & \leq & \{(\text{dist}) \quad\} \\
(u_i\ \overline{q})\ N^*\ R + (p_i + u_i\ q)\ N^*\ R & &
\end{array}
$$

(23) $N^*\ R \leq (p_i + u_i)\ N^*\ R$

$$
\begin{array}{lcl}
N^*\ R & = & \{(9) \quad\} \\
N^*\ (1 + M)\ q\ \top & \leq & \{1 \leq p_i + u_i\ \top\ (17) \quad\} \\
N^*\ (1 + M)\ q\ (p_i + u_i)\ \top & \leq & \{q\ p_i = p_i\ q;\ q\ u_i \leq u_i\ q\ (18) \quad\} \\
N^*\ (1 + M)\ (p_i + u_i)\ q\ \top & \leq & \{M\ (p_i + u_i) \leq (p_i + u_i)\ (M + R),\ (25)\} \\
N^*\ (p_i + u_i)\ (1 + M + R)\ q\ \top & \leq & \{(1 + M + R)\ q\ \top \leq R\ (9) \quad\} \\
N^*\ (p_i + u_i)\ R & \leq & \{N\ (p_i + u_i) \leq (p_i + u_i)\ (N + R)\ (24);\ \} \\
 & & \{(1) \quad\} \\
(p_i + u_i)\ (N + R)^*\ R & \leq & \{R\ N \leq R;\ (2) \quad\} \\
(p_i + u_i)\ N^*\ R^*\ R & = & \{R\ R \leq R;\ (*\ \text{ind}) \quad\} \\
(p_i + u_i)\ N^*\ R & &
\end{array}
$$

Finally, (24) and (25) show that $(p_i + u_i)$ commutes left past an N or M (possibly changing them into R's).

(24) $N_j\ (p_i + u_i) \leq (p_i + u_i)\ (N + R)$

$$
\begin{array}{lcl}
N_j\ (p_i + u_i) & \leq & \{N_j \leq M_j\ p_j\ (4) \quad\} \\
M_j\ p_j (p_i + u_i) & \leq & \{p_j\ p_i = p_i\ p_j;\ p_j\ u_i \leq u_i\ p_j\ (12)(16) \quad\} \\
M_j\ (p_i + u_i)\ p_j & \leq & \{M_j\ (p_i + u_i) \leq (p_i + u_i)\ (M_j + R),\ (25)\} \\
(p_i + u_i)\ (M_j + R)\ p_j & \leq & \{1 = q + \overline{q} \quad\} \\
(p_i + u_i)\ (M_j + R)\ p_j\ (\overline{q} + q) & \leq & \{M_j\ p_j\ \overline{q} \leq N\ (4)(8);\ M_j\ q \leq R\ (9) \quad\} \\
(p_i + u_i)\ (N + R) & &
\end{array}
$$

(25) $M_j\ (p_i + u_i) \leq (p_i + u_i)\ (M_j + R)$

$i = j$:

$$
\begin{array}{lcl}
M_i\ (p_i + u_i) & \leq & \{p_i \leq 1;\ M_i\ u_i \leq M_i\ (3) \quad\} \\
M_i & \leq & \{v_i = p_i\ v_i\ (16),\ \text{ so }\ M_i = p_i\ M_i\ (3)\} \\
(p_i + u_i)\ (M_i + R) & &
\end{array}
$$

$i \neq j$:

Let $[x] = x + x\ q\ \top + q\ \top$; then

$$
\begin{array}{lcl}
M_j\ (p_i + u_i) & = & \{(3) \quad\} \\
v_j\ u_j^*\ (p_i + u_i) & \leq & \{u_j\ (p_i + u_i) \leq (p_i + u_i)\ [u_j]\ (12)(14);\ (1)\} \\
v_j\ (p_i + u_i)\ [u_j]^* & = & \{[u_j]^* = [u_j^*]\ (2) \quad\} \\
v_j\ (p_i + u_i)\ [u_j^*] & \leq & \{v_j\ (p_i + u_i) \leq (p_i + u_i)\ [v_j]\ (13)(15) \quad\} \\
(p_i + u_i)\ [v_j]\ [u_j^*] & \leq & \{[v_j]\ [u_j]^* \leq M_j + R \quad\} \\
(p_i + u_i)\ (M_j + R) & &
\end{array}
$$

4 System Model and Synchronization Discipline

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 various 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*. 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).

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.

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 variables, not the protected variables themselves, should be used to achieve the necessary synchronization. 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 then used in a guard directly.

Fix a system. 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. 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 is *visible* if it (*i*) contains an occurrence of a variable in V_{unprot} or (*ii*) might change the value of an exclusive access predicate. Other transitions are *invisible*. This classification of transitions determines the transition relations u_i and v_i and the predicates p_i.

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

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 4.1 and of x in case 2 of the proof of (15) in Section 5.)

WF-finiteInvis. No thread has an infinite execution sequence containing only invisible transitions. Formally, $(\forall i : u_i^\omega = 0)$.

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.

4.1 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 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

$$(26)\quad \forall i : \forall x \in V_{prot} : \forall t \in T_i : \forall \langle s_1, s_2 \rangle \in [\![t_i]\!] : s_2 \models q \iff \\ ((x \in acc(s_1, t, s_2) \ \wedge \ s_1 \not\models e_i^x) \ \vee \ s_2 \not\models \text{disjoint}(e^x) \ \vee \ s_1 \models q).$$

The third disjunct in (26) 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 $acc(s_1, t, s_2)$.

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

We prove in [SC02] that well-formed systems satisfy the hypotheses (11)–(18) of the reduction theorem. Most of the proofs are straightforward. Here we consider only the most interesting one.

Proof of (15). 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 $t_i = g_i \to c_i$ and $t_j = g_j \to c_j$. t_j does not enable t_i, because c_j and g_i access disjoint sets of variables (because t_i is invisible and hence does not access unprotected variables, and protected variables do not appear in guards). 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 (15)) 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

(27) $acc(s_1, t_j, s_2) = acc(s_2', t_j, s_3') \ \wedge \ acc(s_2, t_i, s_3) = acc(s_1, t_i, s_2')$,

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 4.1.

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 (26) being true (the third disjunct just makes q monotonic). Thus, there are 2×2 cases to consider.

case 1.2.2.1: $(\exists x \in V_{prot} : x \in acc(s_1, t_j, s_2) \wedge s_1 \not\models e_j^x)$. (27) implies $x \in acc(s_2', t_j, s_3')$. t_i is invisible, so it cannot truthify e_j^x, so $s_2' \not\models e_j^x$. Thus, the definition of q implies $s_3' \models q$.

case 1.2.2.2: $(\exists x \in V_{prot} : x \in acc(s_2, t_i, s_3) \wedge s_2 \not\models e_i^x)$. (27) implies $x \in acc(s_1, t_i, s_2')$. WF-endExcl implies t_j did not falsify e_i^x, so $s_1 \not\models e_i^x$. 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, so it cannot falsify any exclusive access predicate, 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: $A \neq \emptyset$. 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 $x \in acc(s'_2, t_j, s'_3)$, 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. 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 $s'_3 \models q$.

6 Examples

This section contains examples of systems for which the current reduction is effective (*i.e.*, it reduces the number of reachable states) and the reduction in [Sto02] is not effective. In general, our method is effective whenever some variables can be classified as protected. These examples are based mainly on descriptions in [SBN+97] of code in real systems.

Semaphores. A user thread sends a request to a device driver thread, asking the device driver to store data in a buffer b, and then waits for the result by invoking *sem*.down(), where *sem* is a semaphore, initialized to zero. The device driver thread receives the request, waits for the device to supply the data, stores the data in b, and then calls *sem*.up(). The buffer b can be classified as protected. For example, e^b_{user} holds when the program counter of the user thread points to a statement after the call to *sem*.down(), and e^b_{driver} holds when the program counter of the device driver thread points to a statement before the call to *sem*.up(). The semaphore ensures disjointness of e^b_{user} and e^b_{driver}.

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^b_i for a block b might be

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

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 $w.\text{task}$ that refers to the worker's task. For a task object x, the exclusive access predicate e^x_{master} holds before x has been passed to a worker thread, and e^x_w holds when $w.\text{task} = x$.

7 Comparison to Traditional Partial-Order Methods

This section demonstrates that our method has advantages over traditional partial-order methods even for some simple systems for which precise static analysis of transition dependencies is feasible. 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

$$(28)\quad \begin{array}{l} {}^A m_0.\text{acquire}();\ {}^B x_0 := i;\ {}^C m_0.\text{release}();\ {}^D m_1.\text{acquire}();\ {}^E x_1 := i; \\ {}^F m_1.\text{release}();\ {}^G y{+}{+};\ {}^H y = 2 \rightarrow \text{skip};\ {}^I x_i = i\ {}^J \end{array}$$

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 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 62 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. For concreteness, we compare our method to selective search using the conditional stubborn set algorithm (CSSA) [God96]. We always resolve non-determinism in CSSA in a way that yields a minimum-size persistent set. CSSA is parameterized by dependency relations on operations. For acquire and release, we use the might-be-the-first-to-interfere-with relation in [Sto02, Fig. 3]. For accesses to y, we use the minimal might-be-the-first-to-interfere-with relation, based on the dependency relation on operations in which 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 selective search (using CSSA) explores 77 states. To illustrate why it explores more than the 62 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, as explained in detail in [SC02].

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

The selective search exploits some independence that our method does not, in particular, independence of release with acquire and release, and independence of acquire with acquire in some states. One way to obtain the benefits of both methods is to apply selective search to the reduced system. This works for systems for which sufficiently precise static analysis of dependencies between transitions is feasible (*cf.* Section 1). Another approach is to extend our method, *e.g.*, to incorporate the specialized treatment of monitor operations in [Sto02] that allows release to be classified as invisible.

8 How to Use the Reduction

The intended methodology for using the reduction is as follows.

1. Guess the set V_{prot} of protected variables and the exclusive access predicates e_i^x. 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 (4).

2. Augment the reduced system with a predicate q, as described in Section 4.1.

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.

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, based on the observation

that the exclusive access predicates typically have the form $e_i^x = eap_i^{x,m}$, where

$$(29) \quad eap_i^{x,m} = init_i^x \vee (i = m.owner \ \wedge \ \neg init^x)$$

$$(30) \quad init^x = (\exists i \in \Theta : init_i^x).$$

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: the initialization predicate holds when the thread's program counter is in the appropriate class initializer (for static fields) or the appropriate constructor invocation (for instance fields).

To use (29), 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 Experimental Results

We implemented the similar reduction of [Sto02] in Java PathFinder (JPF) [BHPV00] and measured the benefit of the reduction for several programs with monitor-based synchronization. HaltException and Clean [BHPV00, Figure 1] are small "synchronization skeletons" supplied by the developers of JPF. Xtango-DP and Xtango-QS are animations of a dining philosophers algorithm and quicksort, respectively, from http://www.mcs.drexel.edu/~shartley/; we replaced java.awt methods with methods having empty bodies, due to limitations of JPF. The lockset algorithm was used in all experiments. With negligible manual effort (to write a few lines of config files), the reduction decreases memory usage by a factor of 1.4MB/0.77MB $\approx$ 1.8 for HaltException, 4.3MB/2.2MB $\approx$ 2.0 for Clean, 609MB/236MB $\approx$ 2.6 for Xtango-DP, and 344MB/101MB $\approx$ 3.4 for Xtango-QS, compared to model checking with JPF'S default granularity, which executes each line of source code atomically. In a real JVM, bytecode instructions execute atomically. Our reduction preserves that semantics. JPF's source-line granularity does not: it can miss errors. Compared to bytecode granularity, our reduction decreases memory usage by a factor of 13.9MB/0.77MB $\approx$ 18 for HaltException and at least 1800MB/101MB $\approx$ 18 for Xtango-QS ("at least" reflects an out-of-memory exception).

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

References

[BR01] C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, November 2001.

[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] Edmund M. Clarke, Jr., Orna Grumberg, and Doron 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] Cormac Flanagan and Stephen 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.

[FQS02] Cormac Flanagan, Shaz Qadeer, and Sanjit 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.

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

[God97] Patrice 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.

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

[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.

[Sto02] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, to appear.

[Val97] Antti 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.

[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.