Model-Checking Multi-Threaded Distributed Java Programs*

Scott D. Stoller
Computer Science Dept., Indiana University, Bloomington, IN 47405-7104 USA

26 September 2000

Abstract

Systematic state-space exploration is a powerful technique for verification of concurrent software systems. Most work in this area deals with manually-constructed models of those systems. We propose a framework for applying state-space exploration to multi-threaded distributed systems written in standard programming languages. It generalizes Godefroid's work on VeriSoft, which does not handle multi-threaded systems, and Bruening's work on ExitBlockRW, which does not handle distributed (multi-process) systems. Unlike ExitBlockRW, our search algorithms incorporate powerful partial-order methods, guarantee detection of deadlocks, and guarantee detection of violations of the locking discipline used to avoid race conditions in accesses to shared variables.

1 Introduction

Systematic state-space exploration (model-checking) is a powerful technique for verification of concurrent software systems. Most work in this area actually deals with manually-constructed models (abstractions) of those systems. The models are described using restricted languages, not general-purpose programming languages. Use of restricted modeling languages can greatly facilitate analysis and verification, leading to strong guarantees about the properties of the model. However, use of such models has two potentially significant disadvantages: first, the effort needed to construct the model (in addition to the actual implementation of the system), and second, possible discrepancies between the behavior of the model and the behavior of the original system. One approach to avoiding these disadvantages is automatic translation of general-purpose programming languages into modeling languages, as in [STMD96, HS99, DIS99, CDH+00]. This facilitates applying abstractions, but automatic translation that handles all language features (including dynamic memory allocation) and standard libraries and yields tractable models is very difficult.

Another approach is to apply state-space exploration directly to software written in general-purpose programming languages, such as C++ or Java. This approach is used in VeriSoft [God97, GHJ98]. Capturing and storing the state of a program written in a general-purpose programming language is difficult, so VeriSoft uses *state-less search*; this means that the search algorithm does not require storage of visited states. Stateless search might visit a state multiple times. VeriSoft uses partial-order methods—specifically, persistent sets and sleep sets (see Section 3)—to reduce this redundancy. VeriSoft is targeted at "distributed" systems, specifically, systems containing multiple single-threaded processes that do not share memory. Processes interact via *communication objects*, such as semaphores or sockets.

ExitBlock [Bru99a] is based on similar ideas as VeriSoft but targets a different class of systems. ExitBlock can test a single multi-threaded Java process that uses locks to avoid race conditions in accesses to variables shared by multiple threads. Specifically, ExitBlock assumes that the process satisfies the mutual-exclusion locking discipline (MLD) of Savage *et al.* [SBN+97]. ExitBlock exploits this assumption to reduce the

*Email: stoller@cs.indiana.edu Web: http://www.cs.indiana.edu/~stoller/ The author gratefully acknowledges the support of ONR under Grant N00014-99-1-0358 and the support of NSF under CAREER Award CCR-9876058.

number of explored interleavings of transitions of different threads. Bruening shows that if a system satisfies MLD, then for the purpose of determining reachability of control points and deadlocks, it suffices to consider schedules in which context switches between threads occur only when a lock is released, including the implicit release performed by Java's wait operation (java.lang.Object.wait).

This paper combines the ideas in VeriSoft and ExitBlock and extends them in several ways. Our framework targets systems of multi-threaded processes that interact via communication objects and that use locks to avoid race conditions in accesses to shared variables.[1] Thus, it handles a strict superset of the systems handled by VeriSoft or ExitBlock. A detailed comparison with related work appears in Section 11.

Our results fall into two categories: results in Sections 4–8 for systems known to satisfy MLD, and results in Section 9 for systems expected to satisfy MLD. Static analyses like Extended Static Checking [DLNS98], types for safe locking [FA99], and protected variable analysis [Cor00] can conservatively determine whether a system satisfies MLD. For such systems, MLD constrains the set of objects that may be accessed by a transition (based on the set of locks held by the thread performing the transition), and this information can be used to constrain dependency between transitions and thereby to compute smaller persistent sets. In the absence of such guarantees, MLD can be checked dynamically during the selective search, using a variant of the lockset algorithm [SBN+97]. Since MLD is expected to hold, we propose to still exploit MLD when computing persistent sets. This introduces a potentially dangerous circularity. If a transition t that violates MLD is incorrectly assumed to be independent of other transitions, this error might cause the persistent-set algorithm to return a set that is too small (*e.g.*, does not include t) and is not actually persistent. Since the explored set of transitions is not persistent, there is *a priori* no guarantee that the selective search will actually find a violation of MLD. Bruening does not address this issue. We show that this can happen with MLD but not with a slightly stricter variant MLD′.

2 System Model

We adopt Godefroid's model of concurrent systems [God96], except that we call the concurrent entities threads rather than processes, disallow transitions that affect the control state of multiple threads, and divide objects into three categories. A *concurrent system* is a tuple $\langle \Theta, \mathcal{O}, \mathcal{T}, s_{init}, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{mtx} \rangle$, where

Θ is a finite set of threads. A thread is a finite set of elements called *control points*. Threads are pairwise disjoint.

$\mathcal{O}$ is a finite set of objects. An *object* is characterized by a pair $\langle Dom, Op \rangle$, where Dom is the set of possible values of the object, and Op is the set of operations that can be performed on the object. An *operation* is a partial function that takes an input value and the current value of the object and returns a return value and an updated value for the object.

$\mathcal{T}$ is a finite set of transitions. A transition t is a tuple $\langle S, G, C, F \rangle$, where: S is a control point of some thread, which we denote by $thread(t)$; F is a control point of the same thread; G is a guard, *i.e.*, a boolean-valued expression built from read-only operations on objects and mathematical functions; and C is a command, *i.e.*, a sequence of expressions built from operations on objects and mathematical functions. We call S and F the *starting* and *final* control points of t.

s_{init} is the initial state. State is defined below.

[1] The division of the system into processes is inconsequential. It does not matter whether the threads sharing a variable are in the same or different processes. A variable shared only by threads in a single process can be regarded as a communication object; this is potentially useful if accesses to that variable do not satisfy MLD.

$\mathcal{O}_{unsh} \subseteq \mathcal{O}$ is the set of unshared objects, *i.e.*, objects accessed by at most one thread.

$\mathcal{O}_{syn} \subseteq \mathcal{O}$ is the set of synchronization objects, defined in Section 2.1.

$\mathcal{O}_{mtx} \subseteq \mathcal{O}$ is the set of objects for which MLD, defined in Section 2.2, is used.

A *state* is a pair $\langle L, V \rangle$, where L is a collection of control points, one from each thread, and V is a collection of values, one for each object. For a state s and object o, we abuse notation and write $s(o)$ to denote the value of o in s. Similarly, we write $s(\theta)$ to denote the control point of thread θ in state s.

A transition $\langle S, G, C, F \rangle$ of thread θ is *pending* in state s if $S = s(\theta)$, and it is *enabled* in state s if it is pending in s and G evaluates to true in s. For a concurrent system $\mathcal{S}$, let $pending_{\mathcal{S}}(s, \theta)$ and $enabled_{\mathcal{S}}(s, \theta)$ denote the sets of transitions of θ that are pending and enabled, respectively, in state s (in system $\mathcal{S}$). Let $enabled_{\mathcal{S}}(s)$ denote the set of transitions enabled in state s. When the system being discussed is clear from context, we elide the subscript. If a transition $\langle S, G, C, F \rangle$ is enabled in state $s = \langle L, V \rangle$, then it can be executed in s, leading to the state $\langle (L \setminus \{S\}) \cup \{F\}, C(V) \rangle$, where $C(V)$ represents the values obtained by using the operations in C to update the values in V. We write $s \xrightarrow{t} s'$ to indicate that transition t is enabled in state s and that executing t in s leads to state s'.

A *sequence* is a function whose domain is the natural numbers or a finite prefix of the natural numbers. Let $|\sigma|$ denote the length of a sequence σ. Let $\sigma(i..j)$ denote the subsequence of σ from index i to index j. Let $last(\sigma)$ denote $\sigma(|\sigma| - 1)$. Let $\langle a_0, a_1, \ldots \rangle$ denote a sequence containing the indicated elements; $\langle\rangle$ denotes the empty sequence. Let $\sigma_1 \cdot \sigma_2$ denote the concatenation of sequences σ_1 and σ_2.

An *execution* of a concurrent system $\mathcal{S}$ is a finite or infinite sequence σ of transitions of $\mathcal{S}$ such that there exist states $s_0, s_1, s_2, \ldots$ such that $s_0 \xrightarrow{\sigma(0)} s_1 \xrightarrow{\sigma(1)} s_2 \cdots$ and $s_0 = s_{init}$. Operations are deterministic, so the sequence of states $s_1, s_2, \ldots$ is completely determined by the sequence of transitions and s_{init}. When convenient, we regard that sequence of states as part of the execution. A state is *reachable* (in a system) if it appears in some execution (of that system). A control point is *reachable* if it appears in some reachable state.

Objects in $\mathcal{O} \setminus (\mathcal{O}_{unsh} \cup \mathcal{O}_{syn} \cup \mathcal{O}_{mtx})$ are called *communication objects*. For example, a system containing Java processes communicating over a socket involves some instances of java.net.Socket, which are in $\mathcal{O}_{mtx}$, and an underlying socket, which is a communication object.

We do not explicitly model allocation and de-allocation of threads or objects. We assume Θ and $\mathcal{O}$ are sufficiently large to accommodate all threads and objects that will be created.

2.1 Synchronization Objects

We plan to apply our framework to model-checking of Java programs, so we focus on the built-in synchronization operations in Java. In our framework, a synchronization object embodies the synchronization-related state that the JVM maintains for each Java object or class. (Java does not contain distinct synchronization objects; every Java object contains its own synchronization-related state. This difference is inconsequential.)

The fields of a synchronization object are: *owner* (name of a thread, or *free*), *depth* (number of unmatched acquire operations), and *wait* (list of waiting threads). We assume that the lock associated with each synchronization object is free in the initial state. The "operations" on synchronization objects are: acquire, release, wait, notify, and notifyAll. Each of these high-level "operations" is represented in a straightforward way as one or more transitions that use multiple (lower-level) operations on the synchronization object. For concreteness, we describe one such representation here. Other encodings are possible. Embedding this in a general semantics for Java bytecode would have little benefit, because wait, notify, and notifyAll are native

methods, and because synchronization-related state is not explicitly accessible at the bytecode level (it is encapsulated within the JVM).

Thread θ acquiring o's lock corresponds to a transition with guard $o.owner \in \{free, \theta\}$ and command $o.owner := \theta; o.depth{+}{+}$. Thread θ releasing o's lock corresponds to two transitions: one with guard contains $o.owner \neq \theta$ and a command that throws an IllegalMonitorStateException, and one with guard $o.owner = \theta$ and command $o.owner := (o.depth = 1)\,?\,free : \theta; o.depth{-}{-}$.[2]

Let *tmpDepth* denote an unshared natural-number-valued object used by θ. Thread θ waiting on o corresponds to three transitions: one with guard $o.owner \neq \theta$ and a command that throws an IllegalMonitorStateException, and one with guard $o.owner = \theta$ and command $o.wait.add(\theta); tmpDepth = o.depth; o.depth := 0; o.owner := free$ followed by one with guard $o.owner = free \wedge \theta \notin o.wait$ and command $o.owner := \theta; o.depth := tmpDepth$. Thread θ doing notify on o corresponds to $|\Theta| + 1$ transitions: one with guard $o.owner \neq \theta$ and a command that throws an IllegalMonitorStateException, one with guard $o.owner = \theta \wedge o.waitIsEmpty()$ and a command that does nothing, and, for each $\theta' \in \Theta \setminus \{\theta\}$, a transition with guard $o.owner = \theta \wedge \theta' \in o.wait$ and command $o.waitRemove(\theta')$, which removes θ' from $o.wait$; all of these transitions except the one that throws the exception have the same final control point. Thread θ doing notifyAll on o corresponds to two transitions: one with guard $o.owner \neq \theta$ and a command that throws an IllegalMonitorStateException, and one with guard $o.owner = \theta$ and command $o.waitClear()$, which makes $o.wait$ empty.

We informally refer to acquire, release, *etc.*, as operations on synchronization objects, when we actually mean the operations used by the corresponding transitions. An important observation is:

SyncWithoutLock: If a thread θ executes an operation *op* other than acquire on a synchronization object o in a state s in which θ does not hold o's lock, then (1) execution of *op* in s does not modify the state of o, and (2) execution of *op* in s has the same effect (*e.g.*, it throws IllegalMonitorStateException) regardless of other aspects of o's state (e.g., regardless of whether o's lock is held by another thread or free, and regardless of whether any threads are blocked waiting on o).

One might hope that synchronization objects could be included in $\mathcal{O}_{mtx}$ and not treated specially in the proofs below. Special consideration is needed for operations on synchronization objects, because they access *o.owner* in a way that violates MLD. Classifying synchronization objects as communication objects would mean that all operations on them are visible, which would increase the cost of the selective search.

Our results are sensitive to the operations on synchronization objects. For example, consider introducing a non-blocking operation Free? that returns true iff the object's lock is free. This operation violates SyncWithoutLock and would require that release be classified as visible (see Section 2.3).

2.2 Mutex Locking Discipline (MLD)

The MLD of [SBN+97] allows objects to be initialized without locking. Initialization is assumed to be completed before the object becomes shared (*i.e.*, accessed by two different threads). We formalize this as follows. Transition t *accesses* object o in state s if (1) t is pending in s and t's guard accesses (*i.e.*, contains an operation on) o or (2) t is enabled in s and t's command accesses o. Thread θ *accesses* object o in state s, denoted $access(s, \theta, o)$, if there exists a transition t in $pending(s, \theta)$ that accesses o in s. $startShared(\sigma, o)$ is the index of the first state in σ in which an access to o that is not part of initialization of o occurs; formally,

[2] The definition of command does not allow conditionals (*cf.* observations A1-A2 in the proof of Theorem 2′). The first assignment statement in this command is syntactic sugar for $o.ownerRelease(o.depth, \theta)$, which is an operation that has exactly the same effect as the assignment statement.

letting σ be $s_0 \stackrel{\sigma(0)}{\rightarrow} s_1 \stackrel{\sigma(1)}{\rightarrow} s_2 \cdots$, $startShared(\sigma, o)$ is the least value of i such that $(\exists i_1, i_2 \leq i : \exists \theta_1, \theta_2 \in \Theta : \theta_1 \neq \theta_2 \wedge access(s_{i_1}, \theta_1, o) \wedge access(s_{i_2}, \theta_2, o))$, or $|\sigma|$ if no such values exist.

Mutex Locking Discipline (MLD): A system $\langle \Theta, \mathcal{O}, \mathcal{T}, s_{init}, \mathcal{O}_{mtx}, \mathcal{O}_{syn} \rangle$ satisfies MLD iff for all executions $\sigma = s_0 \stackrel{\sigma(0)}{\rightarrow} s_1 \stackrel{\sigma(1)}{\rightarrow} s_2 \cdots$ of the system, for all objects $o \in \mathcal{O}_{mtx}$,

MLD-R: o is read-only after it becomes shared, *i.e.*, there exists a constant c such that for all $i \geq startShared(\sigma, o)$, $s_i(o) = c$.

MLD-L: o is properly locked after it becomes shared, *i.e.*, there exists a synchronization object $o_1 \in \mathcal{O}_{syn}$ such that, for all $i \geq startShared(\sigma, o)$, for all $\theta \in \Theta$, if $access(s_i, \theta, o)$, then θ owns o_1's lock in s_i.

We don't consider read/write locks, because Java does not provide built-in support for them.

Godefroid [God96] defines: transition t uses object o iff t's guard or command contains an operation on o. Thus, the command of a disabled transition uses o. Such uses cannot be detected by run-time monitoring, so we do not want the definition of MLD to depend on such uses. This motivates our definition of "accesses".

2.3 Visible and Invisible

Operations are classified into two categories: visible and invisible. Informally, visible operations are points in the computation at which the scheduler takes control and possibly causes a context switch between threads.

All operations on communication objects are visible, as in [God97]. The operations on synchronization objects that may block are visible; thus, acquire and wait (specifically, for wait, the operations in the transition that blocks, not the operations in the other two transitions) are visible. All other operations are invisible. A transition is *visible* if its command or guard contains a visible operation; otherwise, it is *invisible*. A control point S is *visible* if all transitions with starting control point S are visible; otherwise, it is *invisible*. A state s is *visible* if all control points in s are visible; otherwise, it is *invisible*. Visible states correspond to global states in [God97]. We define some conditions on systems:

Separation: Visible and invisible transitions are "separated", *i.e.*, for every thread θ, for every control point $S \in \theta$, all transitions with starting control point S are visible, or all of them are invisible.

Initial Control Locations are Visible (InitVis): For every thread θ, $s_{init}(\theta)$ is visible. This condition is inessential but convenient.

Bound on Invisible Transition Sequences (BoundedInvis): There exists a bound b on the length of contiguous sequences of invisible transitions by a single thread. Thus, in every execution, for every thread θ, every contiguous sequence of $b+1$ transitions executed by θ (ignoring interspersed transitions of other threads) contains at least one visible transition.

Determinism of Invisible Control Points (DetermInvis): In every reachable state, for every thread θ, θ has at most one enabled invisible transition.

Non-Blocking Invisible Control Points (NonBlockInvis): For every thread θ, for every invisible control point S of θ, for every reachable state s containing S, $enabled(s, \theta) \neq \emptyset$. The following condition might be easier to check and implies NonBlockInvis: for every thread θ, for every invisible control point S of θ, the disjunction of the guards of transitions with starting control point S is true.

In a system satisfying DetermInvis, non-determinism may still come from two sources: concurrency (*i.e.*, different interleavings of transitions) and visible transitions (*e.g.*, VeriSoft's VS_Toss operation [God97]).

A straightforward generalization (not considered further in this paper) is to allow conditional invisibility (*i.e.*, let operations be invisible in some states and visible in others) and to classify an acquire operation by θ as invisible in states where $owner = \theta$.

3 State-less Selective Search

The material in this section is paraphrased from [God97]. Two techniques used to make state-less search efficient are persistent sets and sleep sets. Both attempt to reduce the number of explored states and transitions. Persistent sets exploit the static structure of the system, while sleep sets exploit information about the history of the search. Informally, a set T of transitions enabled in a state s is persistent in s if, for every sequence of transitions starting from s and not containing any transitions in T, all transitions in that sequence are independent with all transitions in T.

Dependency Relation. Let $\mathcal{T}$ and *State* be the sets of transitions and states, respectively, of a concurrent system $\mathcal{S}$. $D \subseteq \mathcal{T} \times \mathcal{T}$ is an *unconditional dependency relation* for $\mathcal{S}$ iff D is reflexive and symmetric and for all $t_1, t_2 \in \mathcal{T}$, $\langle t_1, t_2 \rangle \notin D$ ("t_1 and t_2 are independent") implies that for all states $s \in State$,

1. if $t_1 \in enabled(s)$ and $s \xrightarrow{t_1} s'$, then $t_2 \in enabled(s)$ iff $t_2 \in enabled(s')$.
   (Independent transitions neither disable nor enable each other.)
2. if $\{t_1, t_2\} \subseteq enabled(s)$, then there is a unique state s' such that $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s'$ and $s \xrightarrow{t_2} s_2 \xrightarrow{t_1} s'$.
   (Enabled independent transitions commute.)

$D \subseteq \mathcal{T} \times \mathcal{T} \times State$ is a *conditional dependency relation* for $\mathcal{S}$ iff for all $t_1, t_2 \in \mathcal{T}$ and all $s \in State$, $\langle t_1, t_2, s \rangle \notin D$ ("t_1 and t_2 are independent in s") implies that $\langle t_2, t_1, s \rangle \notin D$ and conditions 1 and 2 above hold. This definition of conditional dependency assumes that commands of transitions satisfy the *no-access-after-update* restriction [God96, p. 21]: an operation that modifies the value of an object o cannot be followed by any other operations on o.

Persistent Set. A set $T \subseteq enabled(s)$ is *persistent* in s iff, for all nonempty sequences of transitions σ such that $s_0 \xrightarrow{\sigma(0)} s_1 \xrightarrow{\sigma(1)} s_2 \cdots \xrightarrow{\sigma(n-1)} s_n \xrightarrow{\sigma(n)} s_{n+1}$, if $s_0 = s$ and $(\forall i \in [0..n] : \sigma(i) \notin T)$, then $\sigma(n)$ is independent in s_n with all transitions in T.

Godefroid's state-less selective search (SSS) using persistent sets and sleep sets appears in Figure 1, where *exec* and *undo* are specified by: if $s \xrightarrow{t} s'$, then $exec(s, t) = s'$ and $undo(s', t) = s$. PS(s) returns a set of transitions that is persistent in s. D is an unconditional dependency relation. Because SSS is state-less, it may visit states multiple times; persistent sets and sleep sets help reduce the redundancy. SSS diverges if the state space contains cycles; in practice, divergence is avoided by limiting the search depth.

Following Godefroid [God96] but deviating from standard usage, a *deadlock* is a state s such that $enabled(s)$ is empty. We focus on determining reachability of deadlocks and control points. Reachability of control points can easily encode information about values of objects. For example, a Java program might assert that a condition e_1 holds using the statement `if (!`e_1`) throw` e_2; violation of this assertion corresponds to reachability of the control point at the beginning of `throw` e_2. If necessary (as in Section 5), assertion violations can easily be encoded as reachability of visible control points, by introducing a communication object with a single (visible) operation that is called when any assertion is violated. Questions about reachability of states can be encoded as questions about reachability of control points using standard techniques [God96, chapter 7].

```
Global variables: stack, curState;

SSS() {
  stack := empty;
  curState := s_init;
  DFS(∅);
}

DFS(sleep) {
  T := PS(curState) \ sleep;
  while (T is not empty)
    remove a transition t from T;
    push t onto stack;
    curState  := exec(curState, t);
    sleep' := {t' ∈ sleep | ⟨t, t'⟩ ∉ D};
    DFS(sleep')
    pop t from stack;
    curState  := undo(curState, t);
    sleep  := sleep ∪ {t};
}
```

Figure 1: State-less Selective Search (SSS) using persistent sets and sleep sets.

Theorem 1. Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space. A deadlock d is reachable in $\mathcal{S}$ iff SSS explores d. A control point S is reachable in $\mathcal{S}$ iff SSS explores a state containing S.

Proof: This is a paraphrase of Theorem 2 of [God97]. Assertion violations correspond to reachability of control points. ∎

4 Invisible-First Selective Search

Persistent sets can be used to justify not exploring all interleavings of invisible transitions.

Theorem 2. Let $\mathcal{S}$ be a concurrent system satisfying MLD and Separation. For all threads θ and all reachable states s, if $enabled(s, \theta)$ contains an invisible transition, then $enabled(s, \theta)$ is persistent in s.

Proof: See Appendix. ∎

Suppose the system satisfies MLD, Separation, BoundedInvis, and DetermInvis. If a thread θ has an enabled invisible transition in a state s, then Separation and DetermInvis imply that θ has exactly one enabled transition in s. Theorem 2 implies that it is sufficient to explore only that transition from s. This can be done repeatedly, until θ has an enabled visible transition. BoundedInvis implies that this iteration terminates. Let $execInvis_{\mathcal{S}}(s, \theta)$ be the unique state obtained by performing this procedure starting from state s; if θ has no enabled invisible transitions in state s, then we define $execInvis_{\mathcal{S}}(s, \theta) = s$. Specializing SSS to work in this way yields Invisible-First State-less Selective Search (IF-SSS), given in Figure 2. Note that IF-SSS applies PS to visible states only.

Theorem 3. Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space and satisfying MLD, Separation, BoundedInvis, and DetermInvis. A deadlock d is reachable in $\mathcal{S}$ iff IF-SSS explores d. A control point S is reachable in $\mathcal{S}$ iff IF-SSS explores a state containing S.

Proof: See Appendix. ∎

5 Composing Transitions

In some cases, a stronger partial-order reduction can be obtained by amalgamating a visible transition and the subsequent sequence of invisible transitions explored by IF-SSS into a single transition; an example

```
Global variables: stack, curState;

IF-SSS() {
  stack := empty;
  curState := s_init;
  DFS_if(∅);
}

DFS_if(sleep) {
  T := PS(curState) \ sleep;
  while (T is not empty)
    remove a transition t from T;
    push t onto stack;
    curState  := exec(curState, t);
    curState  := execInvis_S(curState, thread(t));
    sleep' := {t' ∈ sleep | ⟨t, t'⟩ ∉ D};
    DFS_if(sleep')
    pop t from stack;
    curState  := undo(curState, t);
    sleep  := sleep ∪ {t};
}
```

Figure 2: Invisible-First State-less Selective Search (IF-SSS) using persistent sets and sleep sets.

appears in Section 6. Transitions are amalgamated (composed) as follows. Given a sequence σ of transitions, let $cmd_{seq}(\sigma)$ be the sequential composition of the commands of the transitions in σ, and let $guard_{seq}(\sigma)$ be the weakest predicate ensuring that when each transition t in σ is executed, t's guard holds. $guard_{seq}$ can be expressed in terms of the weakest precondition predicate transformer wp [Gri81]:

$$guard_{seq}(\sigma) = guard(\sigma(0)) \wedge \bigwedge_{0<i<|\sigma|} wp(guard(\sigma(i)), cmd_{seq}(\sigma(0..i-1))), \tag{1}$$

where $guard(\langle S, G, C, F\rangle) = G$. We assume $guard_{seq}$ can be expressed in terms of the given operations on objects. Let $final(t)$ denote the final control point of transition t.

Given a concurrent system $\mathcal{S} = \langle \Theta, \mathcal{O}, \mathcal{T}, s_{init}, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{mtx}\rangle$ satisfying MLD, Separation, BoundedInvis, and DetermInvis, $\mathcal{C}(\mathcal{S})$ is $\langle \Theta, \mathcal{O}, \mathcal{T}', s_{init}, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{mtx}\rangle$, where $\mathcal{T}'$ is as follows. Let b be the bound in BoundedInvis for $\mathcal{S}$. For each visible transition $t = \langle S, G, C, F\rangle$ in $\mathcal{T}$, for each sequence σ of invisible transitions of length at most b such that $guard_{seq}(\langle t\rangle \cdot \sigma) \neq$ false and $final(last(\sigma))$ is visible, $\mathcal{T}'$ contains the transition $\langle S, guard_{seq}(\langle t\rangle \cdot \sigma), cmd_{seq}(\langle t\rangle \cdot \sigma), final(last(\sigma))\rangle$. Elements of $\mathcal{T}'$ are analogous to process transitions [God97].

Theorem 4. Let $\mathcal{S}$ be a concurrent system satisfying MLD, Separation, InitVis, BoundedInvis, and DetermInvis. s is a reachable visible state of $\mathcal{S}$ iff s is a reachable visible state of $\mathcal{C}(\mathcal{S})$.

Proof: See Appendix. ∎

Theorem 5. Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space and satisfying MLD, Separation, InitVis, BoundedInvis, and DetermInvis. A deadlock d is reachable in $\mathcal{S}$ iff SSS applied to $\mathcal{C}(\mathcal{S})$ explores d. A control point S is reachable in $\mathcal{S}$ iff SSS applied to $\mathcal{C}(\mathcal{S})$ explores a state containing S.

Proof: This follows directly from Theorems 1 and 4 and the observation that $\mathcal{S}$ and $\mathcal{C}(\mathcal{S})$ have the same set of reachable deadlocks, which follows easily from NonBlockInvis (which implies that all deadlocks of $\mathcal{S}$ are visible) and Theorem 4. ∎

6 Comparison of the Invisible-First and Composition Approaches

Sections 4 and 5 describe two approaches to achieving similar partial-order reductions. The invisible-first approach (Section 4) is worthwhile for three reasons. First, Theorem 2 shows that this reduction is a special case of persistent sets, thereby showing the relationship to existing partial-order methods. Second, Theorem 3 shows that, with IF-SSS, operations in invisible transitions do not need to be recorded (because they do not introduce dependencies that would cause transitions to be removed from sleep sets); we are investigating whether an analogous optimization is possible for SSS applied to $\mathcal{C}(\mathcal{S})$. Third, the guards of composed transitions sometimes introduce dependencies that cause SSS applied to $\mathcal{C}(\mathcal{S})$ to explore more interleavings than IF-SSS. For example, consider a thread θ that is ready to execute

$$\begin{array}{l}\textbf{if } (x_1) \ \{ \ \textbf{if } (x_2) \ c_1 \ \textbf{else } c_2 \ \} \\ \textbf{else } \{ \ \textbf{if } (x_3) \ c_3 \ \textbf{else } c_4 \ \}\end{array}$$

where $x_i \in \mathcal{O}_{mtx}$ and the c_i do not contain visible operations. Let S denote the starting control point of this statement. In the original system $\mathcal{S}$, θ accesses only x_1 at S. In the composed system, θ accesses x_1, x_2, and x_3 at S, because the composed transitions with starting control point S have guards like $x_1 \wedge x_2$ and $\neg x_1 \wedge \neg x_3$. In a state s with $s(\theta) = S$ and $s(x_1) = \text{false}$, the access by θ to x_2 in the composed system is an artifact of composition. Such accesses introduce dependencies that could cause persistent sets to be larger in $\mathcal{C}(\mathcal{S})$ than $\mathcal{S}$. Whether this occurs depends in part on how persistent sets are computed. If they are computed as described in Section 8, this would not occur, because $\mathit{pendInvisOps}(s, \theta)$ would be the same in $\mathcal{S}$ and $\mathcal{C}(\mathcal{S})$. If the calculation of *pendInvisOps* also exploited information from static analysis, this could occur.

The composition approach (Section 5) is worthwhile because it sometimes achieves a stronger partial-order reduction. For example, suppose two threads are both ready to acquire the lock that protects a shared variable v, copy v's value into an unshared variable, and then release the lock. In $\mathcal{C}(\mathcal{S})$, each thread can do this with a single transition, and those two transitions are independent, so SSS applied to $\mathcal{C}(\mathcal{S})$ could explore a single interleaving. In $\mathcal{S}$, each thread does this with a sequence of three transitions, and the transitions that manipulate the lock are not independent (*e.g.*, θ_1 acquiring the lock is dependent with θ_2 acquiring the lock), so IF-SSS applied to $\mathcal{S}$ explores multiple interleavings.

Here is a more detailed example. Consider a state s_1 from which, for $i = 1..2$, thread θ_i is ready to execute the following sequence of five transitions:

$t_{i,1}$**:** acquire the lock of the synchronization object o_i' that protects accesses to an object $o_i \in \mathcal{O}_{mtx}$,

$t_{i,2}$**:** acquire the lock of the synchronization object o_0 that protects accesses to an object $o \in \mathcal{O}_{mtx}$,

$t_{i,3}$**:** read from o and write to o_i,

$t_{i,4}$**:** release o_0's lock,

$t_{i,5}$**:** release o_i''s lock.

Suppose the locks mentioned above are free in s_1. If an "optimal" persistent-set algorithm is used (*i.e.*, one that returns a minimum-sized persistent set, though in general this criterion is only a greedy heuristic for optimizing the overall search), IF-SSS applied to *sys* starts by exploring $s_1 \xrightarrow{t_{1,1}} s_2 \xrightarrow{t_{2,1}} s_3$ (or a symmetric variant). $t_{1,2}$ and $t_{2,2}$ are dependent in s_3 , so the smallest set that is persistent in s_3 is $\{t_{1,2}, t_{2,2}\}$, so IF-SSS explores both of the following transition sequences from s_3 (invisible states are elided):

$$s_3 \xrightarrow{t_{1,2}} \xrightarrow{t_{1,3}} \xrightarrow{t_{1,4}} \xrightarrow{t_{1,5}} s_4 \xrightarrow{t_{2,2}} \xrightarrow{t_{2,3}} \xrightarrow{t_{2,4}} \xrightarrow{t_{2,5}} s_6 \quad \text{and} \quad s_3 \xrightarrow{t_{2,2}} \xrightarrow{t_{2,3}} \xrightarrow{t_{2,4}} \xrightarrow{t_{2,5}} s_5 \xrightarrow{t_{1,2}} \xrightarrow{t_{1,3}} \xrightarrow{t_{1,4}} \xrightarrow{t_{1,5}} s_6.$$

Let w_i denote the sequential composition of $t_{i,2}, t_{i,3}, t_{i,4}, t_{i,5}$. If an "optimal" persistent-set algorithm is used, SSS applied to $\mathcal{C}(\mathcal{S})$ explores the following sequence of transitions (or a symmetric variant):

$$s_1 \stackrel{t_{1,1}}{\rightarrow} s_2 \stackrel{t_{2,1}}{\rightarrow} s_3 \stackrel{w_1}{\rightarrow} s_4 \stackrel{w_2}{\rightarrow} s_6.$$

Note that w_1 and w_2 are independent in s_3.

7 Computing Sleep Sets

Consider refining DFS (in Figure 1) to use a conditional dependency relation when computing *sleep*′; this can produce larger sleep sets and hence more efficient search. Dependency of t and t' should be evaluated in the state prior to execution of t; thus, the line that computes *sleep*′ should be moved immediately above the line containing *exec*, and $\langle t, t' \rangle \notin D$ should be replaced with $\langle t, t', curState \rangle \notin D$. Theorem 1 holds for the modified algorithm, provided the transitions satisfy no-access-after-update. In VeriSoft [God97], this refinement works fine, because only visible operations affect dependency (invisible operations are on unshared objects), and visible operations can only appear as the first operation in a transition's command, so determining that visible operation (by intercepting it) before the transition actually executes is straightforward.

Our framework does not impose those restrictions, so operations on shared objects used by a transition t are not known until after t has been executed, so the calculation of *sleep*′ cannot easily be moved above the line containing *exec*. One solution is to execute and undo t in order to determine its guard and command, but this is expensive, because undo is expensive (especially if implemented using reset+replay or checkpointing). A more efficient approach is to observe that conditional dependency typically depends only on a relatively small and well-defined amount of information about the system's state; in such cases, we can record that information and use it to evaluate *sleep*′ after t is executed. Thus, to execute an operation in the guard or command of a transition t, we first record the relevant parts of the affected object's state, record that t performs this operation, and then perform the operation. For example, for a transition that manipulates a FIFO queue, one might use the conditional dependency relation in [God96, Section 3.4] and therefore record two booleans indicating whether the queue is empty or full.

Dependency relations for transitions are typically derived in a modular way from dependency relations for (the operations of) each object [God96, Definitions 3.15, 3.21]. For some types of objects, it might be difficult or expensive to record the parts of the state that affect conditional dependency, or conditional dependency might provide no benefit (*e.g.*, shared variables with only read and write operations). Also, conditional dependency (as defined in [God96]) cannot be used for objects that are accessed in a way that violates the no-access-after-update restriction. We simply use unconditional dependency for such objects; this is easy, because unconditional dependency is a special case of conditional dependency. As an exception, we can use conditional dependency for some transitions whose accesses to synchronization objects violate the no-access-after-update restriction, *e.g.*, transitions that acquire and then release a lock, as in the example at the end of Section 5. The remainder of this section describes computation of sleep sets in more detail.

Let $\mathcal{O}_u \subseteq \mathcal{O}$ be the set of objects for which unconditional dependency is used. For $o \in \mathcal{O} \setminus \mathcal{O}_u$, let *cDepInfo*$(s, o)$ ("conditional dependency info") denote the information about o's value in state s that is relevant to conditional dependency. For $o \in \mathcal{O}_u$, we define *cDepInfo*(s, o) to be some dummy value. Note that the third component of tuples in a conditional dependency relation for an object can sensibly be changed from the object's value to the conditional dependency info for that object. For $o \in \mathcal{O}$, let D_o denote a dependency relation for o. From these dependency relations for objects, we can derive a conditional

op_1 \ op_2	acquire	release	wait	notify	notifyAll
acquire	$owner \neq free$	$owner \neq \theta_2$	$owner \neq \theta_2$	$owner \neq \theta_2 \vee \theta_1 \notin wait$	$owner \neq \theta_2 \vee \theta_1 \notin wait$
release		true	true	true	true
wait			true	true	true
notify				true	true
notifyAll					true

Figure 3: Operations op_1 and op_2 on a synchronization object o are independent (i.e., do not cause dependency between the transitions in which they appear) if the predicate in the appropriate box holds. op_i is an operation performed by thread θ_i, with $\theta_1 \neq \theta_2$. *owner* is the owner of o's lock, or *free*. *wait* is o's wait set. A blank entry for op_1, op_2 means that is it a symmetric variant of the entry for op_2, op_1, obtained by interchanging θ_1 and θ_2.

dependency relation D^c on transitions [God96, Definition 3.21]:[3]

$$\begin{aligned}&\langle t_1, t_2, s\rangle \notin D^c = \\ &\quad \wedge\ thread(t_1) \neq thread(t_2) \\ &\quad \wedge \text{ for every operation } op_1 \text{ used by } t_1, \text{ for every operation } op_2 \text{ used by } t_2, \\ &\qquad \text{if } op_1 \text{ and } op_2 \text{ are operations on the same object } o, \text{ then} \\ &\qquad (o \in \mathcal{O}_u \wedge \langle op_1, op_2\rangle \notin D_o) \ \vee\ (o \in (\mathcal{O} \setminus \mathcal{O}_u) \wedge \langle op_1, op_2, cDepInfo(s, o)\rangle \notin D_o).\end{aligned} \tag{2}$$

It is easy to see that the third component of a tuple $\langle t_1, t_2, s\rangle$ in D^c can be changed from s to $cDepInfo(s, \{t_1, t_2\})$, where for a set T of transitions,

$$cDepInfo(s, T) = \bigcup_{t \in T} \{cDepInfo(s, o) \mid t \text{ accesses } o \text{ in } s\}. \tag{3}$$

Accurate analysis of dependencies between operations on synchronization objects involves the value of the object and the identities of the threads performing the operations. We assume that the latter can be inferred from the operation (or from a constant argument to the operation in the transition; such arguments can be considered as part of the operation). A conditional dependency relation for synchronization objects appears in Figure 3. The structure of (2) implies that the dependency relation is relevant only if $\theta_1 \neq \theta_2$; thus, the formulas in Figure 3 assume $\theta_1 \neq \theta_2$. The justifications for these formulas are straightforward. Consider the case in which op_1 and op_2 are both not acquire. At least one of these operations is executed by a thread that does not hold o's lock, so SyncWithoutLock implies that op_1 and op_2 can be executed in either order with the same results, so the corresponding predicate is true. Consider the case in which op_1 and op_2 are both acquire; the justifications for other entries involving acquire are similar. Let t_i be a transition containing op_i. For $i \in \{1, 2\}$, t_i's guard contains the conjunct $owner = free$. Consider cases based on the value of *owner*. If $owner = \theta_1$, then op_1 re-acquires the lock, and t_2 is disabled before and after op_2 executes, so these operations do not cause dependence between t_1 and t_2. A symmetric argument applies if $owner = \theta_2$. If $owner \in \Theta \setminus \{\theta_1, \theta_2\}$, then both transitions are disabled, so the conditions for independence of transitions are trivially satisfied. Thus, if $owner \neq free$, the acquire operations do not cause t_1 and t_2 to be dependent.

A version of DFS that records operations and uses conditional dependency appears in Figure 4; the other parts of SSS are unaffected.

[3] We use Lamport's bullet-style notation for lists of conjuncts or disjuncts [Lam93].

```
DFS_c(sleep) {
  T := PS(curState) \ sleep;
  while (T is not empty)
    remove a transition t from T;
    push t onto stack;
    S := control point of thread(t) in curState;
    curState, cdi, G, C, F := exec(curState, t), cDepInfo(s,t), guard(t), command(t), final(t);
    sleep' := {t' ∈ sleep | ⟨⟨S, G, C, F⟩, t', cdi⟩ ∉ D^c};
    DFS_c(sleep')
    pop t from stack;
    curState := undo(curState, t);
    sleep := sleep ∪ {⟨S, G, C, F⟩};
}
```

Figure 4: DFS_c: DFS using conditional dependency relation D^c. The multiple assignment statement indicates that the conditional dependency info, guard, command, and final control point of t are recorded as t executes.

We can use conditional dependency for synchronization objects even if some transitions access synchronization objects in a way that violates the no-access-after-update restriction. This generalization might be significant in practice, since a common pattern is for a single transition in $\mathcal{C}(S)$ to acquire a lock, access a shared object, and then release the lock. This can be accommodated by adding the following disjunct to the last line of (2):

$$o \in \mathcal{O}_{syn} \ \wedge \ (opsOn(o, t_1) = \langle \text{acquire}, \text{release} \rangle \ \vee \ opsOn(o, t_2) = \langle \text{acquire}, \text{release} \rangle), \tag{4}$$

where $opsOn(o, t)$ is the sequence of operations on o in t (recall that release is represented by two transitions; in (4), release denotes the operations in the transition with guard $o.owner = \theta$). To see that this is sound, note that the sequence of operations $\langle \text{acquire}, \text{release} \rangle$ has no net effect on the state of a synchronization object, regardless of the starting state.

8 Computing Persistent Sets

Computing persistent sets requires information about the future transitions of each thread. When model-checking standard languages, the exact set of transitions is not known, so statically determined upper bounds on the set of operations that each thread may perform (ignoring operations on unshared objects) are used to compute persistent sets. Let $allowedOps(\theta)$ denote such an upper bound for thread θ. Let $allowedInvisOps(\theta)$ be the set of invisible operations in $allowedOps(\theta)$. Let $usedVisOps(t)$ be the set of visible operations used by t. We assume that in each visible state s, for each thread θ, the following set is known:

$$pendVisOps(s, \theta) = \bigcup_{t \in pending(s, \theta)} usedVisOps(t) \tag{5}$$

To compute small persistent sets, it is important to have information about the set of invisible operations used by pending transitions of θ in s. A non-trivial upper bound $pendInvisOps(s, \theta)$ on that set can be obtained by exploiting MLD. For concreteness, we describe how to obtain such a bound based on the data structures maintained by the lockset algorithm [SBN+97]. We assume in this section that the system satisfies

MLD; the lockset algorithm is used here only to obtain information about which locks protect accesses to each object. If that information is available from whatever static analysis was used to ensure that MLD holds, then running the lockset algorithm during the search is unnecessary.

Briefly the data structures used by the lockset algorithm are as follows. For each object o, the following values are maintained: $o.mode$, which can be virgin (allocated but uninitialized), exclusive (accessed by only one thread), shared (accessed by multiple threads, but threads after the first did not modify the object), or shared-modified (none of the above conditions hold); $o.firstThread$, which is the first thread that accessed o (*i.e.*, the thread that initializes o; $o.firstThread$ is undefined when o is in virgin mode); and $o.candLockSet$ ("candidate lock set"), which is the set of locks that were held during all accesses to o after initialization (*i.e.*, starting with the access that changed $o.mode$ from exclusive to shared or shared-modified). We assume that $o.candLockSet$ contains all locks (*i.e.*, equals $\mathcal{O}_{syn}$) while o is in exclusive mode. For each thread θ, $held(s,\theta)$, the set of synchronization objects whose locks are held by θ in state s, is maintained, in order to efficiently update candidate lock sets. $acquiring(s,\theta)$ is the set of synchronization objects o such that $pendVisOps(s,\theta)$ contains an acquire operation on o.

$$pendInvisOps(s,\theta) = \bigcup_{o_1 \in held(s,\theta) \cup acquiring(s,\theta)} \{o.op \in allowedInvisOps(\theta) \mid MLDallows(s,\theta,o_1,o)\} \quad (6)$$

$$\begin{aligned} MLDallows(s,\theta,o_1,o.op) = {} & \vee\ o.mode = \text{virgin} \wedge mayInit(s,\theta,o) \quad (7)\\ & \vee\ o.mode = \text{exclusive} \wedge (\theta = o.firstThread \vee readOnly(op) \vee o_1 \in o.candLockSet)\\ & \vee\ o.mode = \text{shared} \wedge (readOnly(op) \vee o_1 \in o.candLockSet)\\ & \vee\ o.mode = \text{shared-modified} \wedge o_1 \in o.candLockSet \end{aligned}$$

where $readOnly(op)$ holds if op is read-only, and $mayInit(s,\theta,o)$ holds if θ can be the first thread to access a virgin object o in state s. For example, in Java, for non-static variables, one might require that θ be the thread that allocated o (for static variables of a class C, θ is the thread that caused class C to be loaded).

For systems that satisfy the following stricter version of MLD-L, we can modify how $o.candLockSet$ is computed in a way that can lead to smaller persistent sets: in every execution in which o is shared, the same lock protects accesses to o; formally, this corresponds to switching the order of the quantifications "for all executions of $\mathcal{S}$" and "there exists $o_1 \in \mathcal{O}_{syn}$". With this stricter requirement, we can modify undo so that it does not undo changes to the candidate lock set. This has the desired effect of possibly making $o.candLockSet$ smaller (hence possibly producing smaller persistent sets) without affecting whether a violation of the requirement is reported.

Persistent sets can be computed using the following variant of Algorithm 2 of [God96], which is based on Overman's Algorithm. We call this Algorithm 2-MLD.

1. Select one transition $t \in enabled(s)$. Let $T = \{thread(t)\}$.
2. For each $\theta \in T$, for each operation $op \in pendVisOps(s,\theta) \cup pendInvisOps(s,\theta)$, for each thread $\theta' \in \Theta \setminus T$, if $(\exists op' \in allowedOps(\theta') : op \triangleright_s op')$, then insert θ' in T.
3. Repeat step 2 until no more processes can be added. Return $\cup_{\theta \in T} enabled(s,\theta)$.

Theorem 6. Let $\mathcal{S}$ be a concurrent system satisfying MLD. In every state s of $\mathcal{S}$, Algorithm 2-MLD returns a set that is persistent in s.

Proof: This follows from correctness of Algorithm 2 of [God96]. ∎

9 Checking MLD During Selective Search

If the system is expected to satisfy MLD but no static guarantee is available, MLD can be checked during the selective search using the lockset algorithm [SBN+97]. As explained in Section 1, the results in Sections 4–8 do not directly apply in this case, because they compute persistent sets assuming that the system satisfies MLD. Here we extend those results to ensure that, if the system violates a slightly stronger variant of MLD, then the selective search finds a violation.

Savage *et al.* observe that their liberal treatment of initialization makes Eraser's checking undesirably dependent on the scheduler [SBN+97, p. 398]. For the same reason, IF-SSS might indeed miss violations of MLD. Consider a system in which θ_1 can perform the sequence of three transitions (control points are omitted in this informal shorthand) $\langle sem.up(), v := 1\rangle$, and θ_2 can perform the sequence of four transitions $\langle sem.down(), o.acquire(), v := 2, o.release()\rangle$, where $v \in \mathcal{O}_{mtx}$ is an integer variable, $o \in \mathcal{O}_{syn}$, and semaphore *sem* (a communication object) is initially zero. This system violates MLD, because $v := 1$ can occur after $v := 2$, and θ_1 holds no locks when it executes $v := 1$. IF-SSS does not find a violation, because after $sem.Up()$, *execInvis* immediately executes $v := 1$.

We strengthen the constraints on initialization by requiring that the thread (if any) that initializes each object be specified in advance and by allowing at most one initialization transition per object (a more flexible alternative is to allow multiple initialization transitions per object, but to require that the initializing thread not perform any visible operations between the first access to o and the last access to o that is part of initialization of o). Formally, we require that a partial function *initThread* from objects to threads be included as part of the system, and we define $startShared'(\sigma, o)$ to be: if o is not in the domain of *initThread*, then zero, otherwise the second smallest i such that $(\exists\theta \in \Theta : access(s_i, \theta, o))$, where σ is $s_0 \stackrel{\sigma(0)}{\rightarrow} s_1 \stackrel{\sigma(1)}{\rightarrow} s_2 \cdots$. Let MLD′ denote MLD with *startShared* replaced with *startShared′*, and extended with the requirement that for each object o in the domain of *initThread*, $initThread(o)$ is the first thread to access o.[4] The lockset algorithm can easily be modified to check MLD′. We assume that accesses to objects in $\mathcal{O}_{mtx}$ by the guard of a transition t are checked in each state in which t is pending (in other words, we assume that in each state, guards of all pending transitions are evaluated). It suffices to check accesses to objects in $\mathcal{O}_{mtx}$ by the command of a transition only when that transition is explored by the search algorithm; to see this, note that the following variant of MLD′-L is equivalent to MLD′-L, in the sense that it does not change the set of systems satisfying MLD′:

MLD′-L1: o is properly locked after it becomes shared, *i.e.*, there exists a synchronization object $o_1 \in \mathcal{O}_{syn}$ such that, for all $i \geq startShared'(\sigma, o)$, (1) if $access(s_i, \sigma(i), o)$, then $thread(\sigma(i))$ owns o_1's lock in s_i, and (2) for all $\theta \in \Theta$, if $pending(s_i, \theta)$ contains a transition whose guard accesses o, then θ owns o_1's lock in s_i.

For a state s, sequence σ of transitions, and transition t that is pending after execution of σ from s, let $s \stackrel{\sigma}{\Longrightarrow}$ denote execution of σ starting from s, and let $s \stackrel{\sigma;t}{\Longrightarrow}$ denote execution of σ starting from s followed by evaluation of t's guard and, if t is enabled, execution of t's command.

Theorem 2′. Let $\mathcal{S}$ be a concurrent system satisfying Separation. For all threads θ and all reachable states s, if $enabled(s, \theta)$ contains an invisible transition, then either $enabled(s, \theta)$ is persistent in s or $enabled(s, \theta)$ contains a transition t such that either $s \stackrel{\langle\rangle;t}{\Longrightarrow}$ violates MLD′ or $s \stackrel{t}{\rightarrow} s'$ and a violation of MLD′ is reachable from s'.

[4] It is easy to show that MLD′ is stricter than MLD (*i.e.*, a system that satisfies MLD′ also satisfies MLD). This observation does not enable one to easily prove the theorems in this section from the unprimed theorems in previous sections or *vice versa*.

Proof: See Appendix. ∎

Theorem 3′. Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space and satisfying Separation, BoundedInvis, and DetermInvis. $\mathcal{S}$ violates MLD′ iff IF-SSS finds a violation of MLD′.

Proof: See Appendix. ∎

Theorem 5′. Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space and satisfying Separation, InitVis, BoundedInvis, and DetermInvis. $\mathcal{S}$ violates MLD′ iff $\mathcal{C}(\mathcal{S})$ violates MLD′.

Proof: ($\Leftarrow$): Let σ be an execution of $\mathcal{C}(\mathcal{S})$ violating MLD′. Expanding each transition in σ into the sequence of transitions of $\mathcal{S}$ from which it is composed yields an execution of $\mathcal{S}$ that violates MLD′.

($\Rightarrow$): Theorem 3′ implies that IF-SSS explores an execution σ of $\mathcal{S}$ that violates MLD′. Composing sequences of transitions in $\mathcal{S}$ to form transitions of $\mathcal{C}(\mathcal{S})$ yields an execution of $\mathcal{C}(\mathcal{S})$ that violates MLD′. ∎

The stricter constraints on initialization in MLD′ allow the definition of *pendInvisOps* to be tightened. Let *pendInvisOps*′ denote that variant of *pendInvisOps*. Let Algorithm 2-MLD′ denote the variant of Algorithm 2-MLD that uses *pendInvisOps*′.

Theorem 6′. Let $\mathcal{S}$ be a concurrent system. In every state s of $\mathcal{S}$, Algorithm 2-MLD′ returns a set P such that either P is persistent in s or P contains a transition t such that t violates MLD′ in s.

Proof: In Algorithm 2-MLD′, only the calculation of *pendInvisOps*′ depends on MLD′, and *pendInvisOps*′(s, θ) is invoked only for threads θ that have already been added to T. Suppose for all threads θ in T, all transitions in *enabled*(s, θ) satisfy MLD′ in s. Then all invocations of *pendInvisOps*′ in this invocation of Algorithm 2-MLD′ returned accurate results, so P is persistent in s. Suppose there exists a thread θ in T such that some transition t in *enabled*(s, θ) violates MLD′ in s. Then P contains t, and t violates MLD′ in s. ∎

Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space and satisfying Separation, InitVis, BoundedInvis, and DetermInvis. Consider applying SSS with Algorithm 2-MLD′ to $\mathcal{C}(\mathcal{S})$ augmented with the lockset algorithm, modified slightly to check MLD′. Theorem 6′ implies that if no violation of MLD′ is found, then $\mathcal{C}(\mathcal{S})$ satisfies MLD′ and hence MLD. Theorem 3 Theorem 5′ then implies that $\mathcal{S}$ satisfies MLD. Theorem 5 can then be used to conclude that reachability of control points and deadlocks was correctly determined during the search.

Let $\mathcal{S}$ be a concurrent system with a finite and acyclic state space and satisfying Separation, InitVis, BoundedInvis, and DetermInvis. Consider applying IF-SSS with Algorithm 2-MLD′ to $\mathcal{S}$ augmented with the lockset algorithm, modified slightly to check MLD′. Theorems 3′ and 6′ imply that if no violation of MLD′ is found, then $\mathcal{S}$ satisfies MLD′ and hence MLD. Theorem 3 implies that reachability of control points and deadlocks was correctly determined during the search. Similarly, consider applying SSS to $\mathcal{C}(\mathcal{S})$ augmented to check MLD′. Theorem 6′ implies that if no violation of MLD′ is found, then $\mathcal{C}(\mathcal{S})$ satisfies MLD′, so Theorem 5′ implies that $\mathcal{S}$ satisfies MLD′ and hence MLD. Theorem 5 implies that reachability (in $\mathcal{S}$) of control points and deadlocks was correctly determined during the search.

10 Implementation

A prototype implementation for multi-threaded single-process systems is mostly complete, thanks to Gregory Alexander, Aseem Asthana, Sean Broadley, Sriram Krishnan, and Adam Wick. It transforms Java class files (application source code is not needed) by inserting calls to a scheduler at visible operations and inserting calls to a variant of the lockset algorithm at accesses to shared objects. The scheduler, written in Java, performs stateless selective search. The JavaClass toolkit [Dah99] greatly facilitated the implementation.

The scheduler runs in a separate thread. The scheduler gives a selected user thread permission to execute (by unblocking it) and then blocks itself. The selected user thread executes until it tries to perform a visible operation, at which point it unblocks the scheduler and then blocks itself (waiting for permission to continue). Thus, roughly speaking, only one thread is runnable at a time, so the JVM's built-in scheduler does not affect the execution.

The tool exploits annotations indicating which objects are (possibly) shared. Object creation commands (namely, the new instruction, and invocations of java.lang.Class.newInstance and java.lang.Object.clone) can be annotated as creating unshared objects, accesses to which are not intercepted, or as creating tentatively unshared objects, accesses to which are intercepted only to verify that the objects are indeed unshared. Objects created by unannotated commands are potentially shared; accesses to them are intercepted to check MLD and, if necessary, are recorded to determine dependencies. Currently, annotations are provided by the user; escape analysis, such as [WR99], could provide them automatically.

By default, classes have *field granularity*, *i.e.*, the intercepted operations are field accesses (getfield and putfield instructions). For some classes, it is desirable for operations to correspond to method calls for purposes of checking MLD and computing dependencies. We say that such classes have *method granularity*. For example, with semaphores, operations seen by the scheduler should be up (also called V or signal) and down (also called P or wait), not reads and writes of fields. Intercepting operations at method granularity reduces overhead and allows use of class-specific dependency relations. The annotation file indicates which classes have method granularity.

When methods are considered as operations, the boundaries of the operation must be defined carefully, because a method can invoke methods of and directly access fields of other objects. In our framework, by default, an intercepted method invocation i represents accesses to `this` performed by i but not accesses to `this` performed by methods invoked within i; it does not represent accesses to other objects. Accesses by i to instances of other classes are intercepted based on the granularities of those other classes; indicating that a class C has method granularity determines only how accesses to instances of C are intercepted. We require that methods of classes with method granularity perform no visible operations, except that the methods may be synchronized.

Ideally, for a class C with method granularity, *all* accesses to instances of C are intercepted at the level of method invocations. If C has non-private fields that are accessed directly by other classes, those field accesses would also need to be recorded. Therefore, we require that method granularity be used only for classes whose instance fields (including inherited ones) are all private or final (accesses to final fields are ignored). Similarly, an invocation of a method $C.m$ can access private fields of instances of C other than `this`. We disallow method granularity for classes that perform such accesses; a simple static analysis can conservatively check this requirement. If this turns out to be undesirably restrictive (e.g., for classes that use such accesses to implement comparisons, such as equals), we can deviate from the above ideal and explicitly record such field accesses; a simple static analysis can identify getfield and putfield instructions that possibly access instances other than `this`.

Classes may be annotated as having *atomic granularity*. An intercepted invocation i of a method (including, as always, inherited methods) of such a class represents all computations performed by i, including computations of other methods invoked from i except methods invoked on other instances of atomic classes. Requirements for atomic granularity include the three above requirements for method granularity. Furthermore, in order to ensure that invocations of atomic methods are dependent only with invocations of atomic methods on the same object, we require that an instance o_{na} of a non-atomic class accessed by a method of an instance o_a of an atomic class be "encapsulated" within o_a; specifically, if the computation represented by an intercepted invocation of a method of o_a accesses o_{na} in a way other than testing whether it is an instance

of an atomic class (this accommodates "equals" methods), then all accesses to o_{na} occur in computations represented by intercepted invocations of methods of o_a. We also require that methods of an atomic class C do not access static variables of classes other than C, and that all static fields of C are private. A proof that these conditions are sufficient is left for future work.

Currently, the user is responsible for checking the requirements for using method or atomic granularity; static analysis could provide conservative automatic checks. The Sun JDK 1.2.2 reference implementation of the java.util.Collection API mostly satisfies the requirements for atomic granularity, except for methods that return a collection backed by another collection, such as the keySet, values, and entrySet methods in java.util.AbstractMap. Atomic granularity can be used for Collection classes in programs that do not invoke such methods.

Synchronized methods and methods of classes with method or atomic granularity are intercepted using automatically generated wrapper classes. Unshared objects are instances of the original class C; shared objects are instances of C's wrapper class, which extends C. For each such method m, the wrapper class contains a wrapper method that overrides m. If m is synchronized, the wrapper indicates that it is trying to acquire a lock, yields control to the scheduler, waits for permission to proceed, invokes super.m, and then releases the lock. If the class has method or atomic granularity, the wrapper calls the lockset algorithm and possibly records the operation. An "invokevirtual $C.m$" instruction requires no explicit modification; the JVM's method lookup efficiently determines whether the instance is shared. For method invocations on unshared instances, the overhead is negligible. An obvious alternative approach, which we call Outside, is to insert near each invocation instruction a segment of bytecode that explicitly tests whether the instance is shared and, if so, performs the steps described above. With Outside, the overhead is non-negligible even for unshared instances. Another benefit of using wrappers to intercept invokevirtual is that, when generating a wrapper, it is easy to determine whether the method being wrapped is synchronized. With Outside, if the instance is shared, the inserted bytecode would need to explicitly check the class of the instance, because a synchronized method can override an unsynchronized method, and *vice versa*. Also, wrappers are convenient for intercepting RMIs on the server side. Wrappers for `run` methods of classes implementing Runnable are special: their first action is to block, waiting for permission to proceed.

Field accesses, array accesses, invokespecial instructions, and synchronization instructions (monitorenter and monitorexit) are intercepted using Outside. The bytecode inserted near these instructions must efficiently determine whether an object is shared. Looking for the object in a hash table, or using java.lang.Object.getClass to check whether it is an instance of a wrapper class, would be expensive. Inserting in java.lang.Object a boolean field would be a nice solution if it didn't give the JVM (Sun JDK 1.2.1 production implementation) a heart attack. Our solution is to insert in java.lang.Object a boolean-valued method, called isShared, whose body is "return false". This method is overridden in all wrapper classes by a method whose body is "return true". Using the aforementioned JVM, invoking isShared has approximately the same cost as a field access.

Certain calls to java.util.Random are intercepted and treated as non-deterministic: all possible return values are explored. This is similar to VS_Toss in VeriSoft [God97].

Java provides no direct way to check whether the lock associated with an object is currently held. This allows release to be invisible (*cf.* Section 2.1) but forces the scheduler to maintain this information itself, which complicates the implementation. For example, if a synchronized method invoked with invokespecial throws an exception, the inserted bytecode must catch this exception, record the release, and then re-throw the exception from within the scopes of the same exception handlers as the original invokespecial instruction.

The lockset algorithm requires associating some information (*e.g.*, a candidate lock set) with each potentially shared object. We encapsulate this information in a SharedInfo class and include in each wrapper class an instance field of type SharedInfo. Since there is a single instance of SharedInfo per instance, not per

field, a violation of MLD′ is reported if different locks protect different fields of an instance. Maintaining a separate instance of SharedInfo for each field would be straightforward. For each static field of a class C, the wrapper class for C contains a corresponding static field of type SharedInfo.

$undo(s,t)$, as used in SSS or IF-SSS, can be implemented in three ways: reverse computation, reset+replay, and checkpointing. Reverse computation is theoretically the most attractive but is difficult to implement. Our current prototype uses reset+replay (like VeriSoft), mainly because it is the easiest approach to implement. ExitBlock [Bru99b] and Java PathFinder [BHPV00] use checkpointing, which requires a customized JVM. Checkpointing is more efficient than reset+replay for CPU-intensive programs. Experiments comparing the efficiency of checkpointing and reset+replay for typical applications of these testing tools are needed. These tools, like other model checkers, are typically applied to small problem instances, which consume relatively little CPU time. Also, CPU-intensive code not directly relevant to the properties being checked can be manually removed during testing; program slicing and other static analyses can facilitate such abstractions.

One inevitable difference between an execution and a replay is that different object references are created during replay. To facilitate calculation of dependencies, we put a sequence number in each shared object, by adding an appropriate field in all wrapper classes. The sequence number provides an object identifier that is unchanged by reset+replay.

Constructors (which officially are not methods, though we treat them with the same granularity as methods) are intercepted by inserting bytecode at the beginning of the code for the constructor in the original class C; this bytecode checks whether the object is shared and, if so, updates the sequence number of the object and calls the lockset algorithm. Each constructor in the wrapper class for C simply invokes the corresponding constructor in C. If we instead inserted this bytecode in the constructor in the wrapper class, when an instance of C is created, invocations of constructors of superclasses of C would not be intercepted (unless additional effort was made), because wrapper classes do not extend each other.

RMI requires special treatment in the lockset algorithm, because when running an RMI, the JVM is free to use a new thread or (for efficiency) re-use a thread that executed a previous RMI. To be safe, the lockset algorithm should regard each RMI as being executed by a different thread. Note that a remotely-invokable method can be invoked locally as well. Java does not provide a convenient way to determine from within a method whether the current invocation was initiated locally or remotely, but this can be determined indirectly, by examining the thread group or the call stack (obtained using Throwable.printStackTrace).

Our prototype has been applied to some simple programs (e.g., dining philosophers) but is under construction and currently has some limitations: array accesses are not intercepted; support for notifyAll, communication objects, and RMI are unimplemented; Algorithm 2-MLD′ and dependency relations for semaphores, queues, *etc.*, are unimplemented, so $enabled(s)$ is used as the persistent set, and a simple read/write dependency relation is used to compute sleep sets. These limitations can be overcome with a modest amount of implementation effort; none reflects a limitation of the underlying framework.

11 Related Work

The framework in [God97] can be regarded as the special case of ours that handles systems with $\mathcal{O}_{syn} = \emptyset$ and $\mathcal{O}_{mtx} = \emptyset$.

Java PathFinder [BHPV00] incorporates a custom JVM, written in Java, that supports traditional (as opposed to state-less) selective search. It ensures that each state is explored at most once but probably has more overhead than our bytecode-rewriting approach. It incorporates partial-order reductions, as in Spin [HP94], but does not exploit MLD, so in principle, every access to a shared variable needs to be intercepted

to check for dependencies.

Corbett's protected variable reduction [Cor00] exploits MLD to make state-space exploration more efficient. Corbett proposes a static analysis that conservatively checks whether objects are accessed in a way that satisfies MLD. In [Cor00], Corbett does not provide results on checking MLD during state-space exploration and does not consider making release, notify, and notifyAll invisible (except for releases that do not make the lock free).

In [Bru99a], Bruening considers only threads interacting via shared variables; the partial-order methods used in our framework also accommodate arbitrary communication objects. ExitBlock corresponds roughly to IF-SSS with $\mathrm{PS}(s) = \mathit{enabled}(s)$ and, for the calculation of sleep sets, the trivial dependency relation $\mathcal{T} \times \mathcal{T}$ (*i.e.*, every transition is dependent with every transition). This is equivalent to SSS applied to $\mathcal{C}(\mathcal{S})$ with $\mathrm{PS}(s) = \mathit{enabled}(s)$ and, for the calculation of sleep sets, the trivial dependency relation $\mathcal{T}' \times \mathcal{T}'$. (To see that the delayed thread in ExitBlock is not a sleep set, note that ExitBlock explores *every* exit-block schedule [Bru99a, Theorem 2]. If sleep sets were used, some exit-block schedules would not be explored.) ExitBlockRW corresponds roughly to IF-SSS with $\mathrm{PS}(s) = \mathit{enabled}(s)$ and, for the calculation of sleep sets, the unconditional dependency relation that recognizes the independence of operations on different objects and of read operations on the same object.

IF-SSS (or SSS applied to $\mathcal{C}(\mathcal{S})$) allows the use of any persistent-set algorithm and any dependency relation. This flexibility allows properties of common synchronization constructs to be exploited. For example, for threads interacting via a shared FIFO queue, IF-SSS can exploit the fact that in states where the queue is non-empty, an insertion and a removal are independent. Similarly, for interaction involving a semaphore, IF-SSS can exploit the fact that in states where the semaphore's value is positive, an up operation is independent with a down operation. Accesses to fields of synchronization objects (*e.g.*, *owner* and *wait*) and to fields of other synchronization constructs (*e.g.*, the value field of a semaphore) are included in ExitBlockRW's read and write sets and therefore cause dependencies based on the simple read/write dependency relation.

ExitBlock treats release as visible and acquire as invisible. This complicates deadlock detection in ExitBlock, and ExitBlockRW might miss deadlocks. IF-SSS and SSS find all reachable deadlocks.

ExitBlock and ExitBlockRW backtrack when a thread θ tries to acquire a lock held by another thread. The work done in executing θ from the previous visible operation to the "failed" acquire is wasted. Our approach does not involve any such wasted effort.

ExitBlockRW requires recording information about invisible operations—specifically, it records the sets of objects read and written by each block. With IF-SSS, invisible operations do not need to be recorded; they do need to be intercepted, mainly to check MLD$'$, unless the system is known to satisfy MLD.

Bruening's proof that ExitBlock finds all assertion violations [Bru99a, Theorem 3, pp. 47-48] is incomplete, because the proof implicitly assumes that all accesses satisfy MLD. Accesses to synchronization-related state (*e.g.*, $o.\mathit{owner}$) need not follow MLD and therefore require special consideration in the proof.

Bruening does not prove that ExitBlock (or ExitBlockRW) is guaranteed to find a violation of MLD for systems that violate MLD. Even if violations of MLD are manifested as assertion violations, the (incomplete) proof that ExitBlock finds all assertion violations [Bru99a, Theorem 3, pp. 47-48] does not imply that ExitBlock finds all violations of MLD, because that proof presupposes that the system satisfies MLD.

Bruening's proof that ExitBlockRW finds all assertion violations is dubious [Bru99a, pp. 53-54].[5] Our Theorem 2 clearly shows how the idea of exploiting MLD is related to persistent sets. Bruening does not relate ExitBlock or ExitBlockRW to existing partial-order methods.

[5] In the first sentence of the second paragraph of the proof, the requirements on s'_i and s'_{i-1} are symmetric; thus, after swapping two segments, one can immediately swap them again, thereby returning to the original schedule. Thus, the meaning of the next sentence of the proof ("Move each segment in this way as far as possible to the left.") is unclear.

References

[BHPV00] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

[Bru99a] Derek L. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999. Available via http://sdg.lcs.mit.edu/rivet.html.

[Bru99b] Derek L. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[CDH+00] James C. Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[Cor00] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *ACM Transactions on Software Engingeering and Methodology*, 9(1):51–93, January 2000.

[Dah99] Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin, 1999. Available via http://www.inf.fu-berlin.de/~dahm/JavaClass/.

[DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, July 1999.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq SRC, 1998.

[FA99] Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proc. European Symposium on Programming (ESOP)*, volume 1576 of *LNCS*, pages 91–108. Springer-Verlag, March 1999.

[GHJ98] Patrice Godefroid, Robert S. Hanmer, and Lalita Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA'98)*, pages 124–133, 1998.

[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.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[HP94] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. International Conference on Formal Description Techniques (FORTE)*, 1994.

[HS99] Klaus Havelund and Jens U. Skakkebæk. Applying model checking in Java verification. In *Proc. 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*, pages 216–231. Springer-Verlag, September 1999.

[Lam93] Leslie Lamport. How to write a long formula. Technical Report SRC-119, Digital Equipment Corporation, Systems Research Center, 1993. Available via http://www.research.digital.com/SRC/personal/Leslie_Lamport/proofs/proofs.html.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[STMD96] S. M. Shatz, S. Tu, T. Murata, and S. Duri. An application of Petri net reduction for Ada tasking deadlock analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1307–1322, December 1996.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 187–206. ACM Press, October 1999. Appeared in *ACM SIGPLAN Notices* 34(10).

Appendix

Proof of Theorem 2: Let σ be a sequence of transitions such that $s_0 \stackrel{\sigma(0)}{\rightarrow} s_1 \stackrel{\sigma(1)}{\rightarrow} s_2 \cdots \stackrel{\sigma(n-1)}{\rightarrow} s_n \stackrel{\sigma(n)}{\rightarrow} s_{n+1}$ with $s_0 = s$ and $(\forall i \in [0..n] : \sigma(i) \notin \mathit{enabled}(s, \theta))$. It suffices to show that $\sigma(n)$ is independent in s_n with all transitions $t = \langle S, C, G, F \rangle$ in $\mathit{enabled}(s, \theta)$. By hypothesis, $\mathit{enabled}(s, \theta)$ contains an invisible transition, so Separation implies that t is invisible. Note that the control point of θ in s is S.

We first show that σ does not contain transitions of θ. The proof is by induction. Base case: $\sigma(0)$ is taken from state s, and $\sigma(0) \notin \mathit{enabled}(s, \theta)$, so $\sigma(0)$ is not a transition of θ. Step case: The induction hypothesis is that $\sigma(0..i)$ does not contain transitions of θ, and we need to show that $\sigma(i+1)$ is not a transition of θ. We assume that $\sigma(i+1)$ is a transition $t_d = \langle S_d, G_d, C_d, F_d \rangle$ of θ and show a contradiction. By hypothesis, $\sigma(i+1) \notin \mathit{enabled}(s, \theta)$, *i.e.*, t_d is disabled in s, so to reach a contradiction, it suffices to show that $\sigma(0..i)$ does not cause any transition of θ that is disabled in s to become enabled in s_{i+1}. By the induction hypothesis, $\sigma(0..i)$ does not contain transitions of θ, so it does not change θ's control point, so $S_d = S$. By hypothesis, $\mathit{enabled}(s, \theta)$ contains an invisible transition. The starting control point of that transition must be S. Thus, Separation implies that t_d is invisible. t_d can become enabled by $\sigma(0..i)$ only through updates to objects accessed by t_d. t_d is invisible, so it does not access communication objects or perform acquire or wait on synchronization objects. All other operations on synchronization objects are non-blocking and therefore do not affect whether t_d is enabled, even if t_d uses some of those operations. By hypothesis, $\sigma(0..i)$ and t_d are transitions of different threads, so accesses by $\sigma(0..i)$ to unshared objects cannot enable t_d. Finally, consider accesses by a transition $\sigma(j)$ to an object o in $\mathcal{O}_{mtx}$, where $0 < j \leq i$.

case: $\sigma(j)$ is part of initialization of o. We show by contradiction that t_d's guard does not access o in s_{i+1}, hence $\sigma(j)$'s access to o does not affect t_d's enabledness. Suppose t_d's guard accesses o in s_{i+1}. Since $S_d = S$, t_d's guard also accesses o in s, so $\mathit{access}(s, \theta, o)$ holds. But s appears before s_j in the execution and $\mathit{thread}(\sigma(j)) \neq \theta$, so o becomes shared at or before s_j, so $\sigma(j)$ cannot be part of initialization of o.

case: $\sigma(j)$ is not part of initialization of o.

- **case:** MLD-R holds for o. $\sigma(j)$ is not part of initialization of o, so $\sigma(j)$ does not update o, so $\sigma(j)$'s access to o cannot affect t_d's enabledness.
- **case:** MLD-L holds for o. We show that t_d's guard does not access o in s_{i+1}, hence $\sigma(j)$'s access to o does not affect t_d's enabledness. Let o_1 be the synchronization object whose lock protects accesses to o. MLD-L implies $\mathit{thread}(\sigma(j))$ holds o_1's lock in state s_j, so θ does not hold o_1's lock in s_j. Since $\sigma(j..i)$ does not contain transitions of θ, θ does not hold o_1's lock in s_{i+1}, so MLD-L implies that t_d does not access o in s_{i+1}.

This completes the proof that σ does not contain transitions of θ.

Suppose $\sigma(n)$ accesses an object o in s_n; thus, $\sigma(n)$ contains an operation op_n on o. We show that the presence of this operation in $\sigma(n)$ does not cause dependence between t and $\sigma(n)$ in s_n. If t does not access o in s_n, this is obvious. Suppose t accesses o in s_n; thus, t contains an operation op on o. Consider four cases.

case: $o \in \mathcal{O}_{unsh}$. This is impossible, since $\sigma(n)$ and t are transitions of different threads and both access o.

case: $o \in \mathcal{O}_{syn}$. t is invisible, so op is not acquire or wait, so op is release, notify, or notifyAll.

case: θ holds o's lock in s_n. As shown above, $thread(\sigma(n)) \neq \theta$, so $thread(\sigma(n))$ does not hold o's lock in s_n. $\sigma(n)$ is enabled in s_n, so op_n is not acquire. SyncWithoutLock-1 implies that op_n does not modify the state of o, so op_n does not affect execution of op. op cannot cause $thread(\sigma(n))$ to hold o's lock, so SyncWithoutLock-2 implies that execution of op_n is unaffected by execution of op.

case: θ does not hold o's lock in s_n. SyncWithoutLock-1 implies that op does not modify the state of o, so op does not affect execution of op_n. op_n cannot cause θ to hold o's lock, so SyncWithoutLock-2 implies that execution of op is unaffected by execution of op_n.

case: $o \in \mathcal{O}_{mtx}$. By hypothesis, θ and $thread(\sigma(n))$ both access o in s_n, and $\theta \neq thread(\sigma(n))$ (as shown above), so o is shared in s_n, *i.e.*, $\sigma(n)$ and t are not part of initialization of o. Consider two cases, corresponding to the cases in the definition of MLD.

case: MLD-R holds for o. MLD-R implies that $\sigma(n)$ and t do not update o in s_n, so op_n and op are independent in s_n.

case: MLD-L holds for o. This case is impossible. Let o_1 be the synchronization object whose lock protects accesses to o. By hypothesis, t accesses o in s_n; since $s_n(\theta) = s(\theta)$ and $t \in enabled(s)$, it follows that t accesses o in s. Thus, MLD-L implies that θ holds o_1's lock in s. $\sigma(0..n-1)$ does not contain transitions of θ and therefore does not cause θ to lose o_1's lock, so θ holds o_1's lock in s_n. By hypothesis, $sigma(n)$ accesses o in s_n, so MLD-L implies that $thread(\sigma(n))$ holds o_1's lock in s_n. Since $\theta \neq thread(\sigma(n))$, the conclusions of the two preceding sentences form a contradiction.

case: $o \notin (\mathcal{O}_{unsh} \cup \mathcal{O}_{syn} \cup \mathcal{O}_{mtx})$. This case is impossible. o is a communication object, so op is visible. This contradicts the hypothesis that t is invisible. ∎

Proof of Theorem 3: This follows directly from Theorems 1 and 2, by comparing an execution of IF-SSS with an execution of SSS using a persistent set function PS that returns a singleton set containing an invisible transition whenever possible. The only significant difference between the two executions is in the calculation of sleep sets. SSS inserts invisible transitions in sleep sets, and IF-SSS does not, but using smaller sleep sets is clearly safe. Actually, inserting invisible transitions in sleep sets does not reduce the number of transitions explored by SSS, because Separation and DetermInvis imply that the argument of DFS never contains invisible transitions. IF-SSS does not explicitly check whether transitions in *sleep* are independent with invisible transitions executed by *execInvis*. This is safe because the former and the latter are always independent, because (1) if $enabled(s)$ contains an invisible transition t of a thread θ, then Separation and DetermInvis imply that t is the only transition of θ in $enabled(s)$, and Theorem 2 implies that t is independent in s with all transitions of other threads in $enabled(s)$; (2) in IF-SSS, when each invisible transition is executed, $sleep \subseteq enabled(curState)$ (similarly, in SSS, when *exec* is called, $sleep \subseteq enabled(curState)$). ∎

Proof of Theorem 4: ($\Leftarrow$): This direction follows immediately from the observation that for every execution σ of $\mathcal{C}(\mathcal{S})$, $expand_{\mathcal{S}}(\sigma)$ is an execution of $\mathcal{S}$, where $expand_{\mathcal{S}}$ is the function that takes an execution σ' of $\mathcal{C}(\mathcal{S})$ and returns the sequence of transitions obtained from σ' by expanding each transition t' in σ' into the sequence of transitions of $\mathcal{S}$ from which t' is composed.

($\Rightarrow$): Let s be a reachable visible state of $\mathcal{S}$; thus, there exists an execution σ of $\mathcal{S}$ such that $s_0 \stackrel{\sigma(0)}{\rightarrow} s_1 \stackrel{\sigma(1)}{\rightarrow} s_2 \cdots \stackrel{\sigma(n-1)}{\rightarrow} s_n \stackrel{\sigma(n)}{\rightarrow} s_{n+1}$ with $s_0 = s_{init}$ and $s_{n+1} = s$.

Let $\langle t_0, t_1, \ldots, t_m \rangle$ be the subsequence of invisible transitions in σ. We re-arrange σ using the following procedure, which moves the invisible transitions of θ that appear between the i'th and $(i+1)$'th visible transitions of θ backwards so that those invisible transitions form a contiguous subsequence of σ starting immediately after the i'th visible transition of θ.

for $i = 0$ to m
  while the transition t immediately preceding t_i in σ has $thread(t) \neq thread(t_i)$
    swap t_i and t in σ;

We show that each swap preserves the fact that σ is an execution of $\mathcal{S}$. Suppose a fragment $s \xrightarrow{t} s' \xrightarrow{t_i}$ of σ is modified by a swap, *i.e.*, t and t_i get swapped. Note that $thread(t) \neq thread(t_i)$. It suffices to show that t_i is enabled in s, and that t and t_i are independent in s. For the former, since t_i is enabled in s', it suffices to show that t cannot possibly change t_i's status from disabled to enabled. t and t_i are transitions of different threads, so accesses by t to unshared objects cannot enable t_i. t_i is invisible and hence cannot access communication objects or perform acquire or wait on synchronization objects. The other operations on synchronization objects are non-blocking, so even if t_i uses them, they do not affect whether t_i is enabled. Suppose t_i's guard contains some operation *op* on some object $o \in \mathcal{O}_{mtx}$. We prove by contradiction that t's command does not update o, which implies that t does not affect t_i's enabledness via *op*. Suppose t's command updates o. t is enabled in s, so t accesses o in s. t_i's guard accesses o, and t_i is pending in s (because t_i is pending in s', and t does not change $thread(t_i)$'s control location), so t_i accesses o in s. Thus, neither t nor t_i is part of initialization of o in σ.

case: MLD-R holds for o. MLD-R implies that t does not update o, a contradiction.

case: MLD-L holds for o. Let o_1 be the synchronization object whose lock protects accesses to o. MLD-L implies that $thread(t)$ holds o_1's lock in s, and $thread(t_i)$ holds o_1's lock in s. This is impossible, because $thread(t) \neq thread(t_i)$.

This completes the proof that t_i is enabled in s. t_i is invisible, so Theorem 2 implies that $enabled(s, thread(t_i))$ is persistent in s. By hypothesis, $thread(t) \neq thread(t_i)$, so $t \notin enabled(s, thread(t_i))$. Since $t \in enabled(s)$, the definition of persistent set implies that t and t_i are independent in s.

Let v and m be such that $\langle \sigma(v(0)), \sigma(v(1)), \ldots, \sigma(v(m)) \rangle$ is the subsequence of visible transitions in σ. InitVis implies that s_{init} is visible, so $v(0) = 0$. Let $w_i = \sigma(v(i)..v(i+1)-1)$. Each w_i leads to a visible state, denoted s'_{i+1}. If w_i contains the last transition of $thread(w_i(0))$ in σ, then visibility of s'_{i+1} follows from visibility of s; otherwise, it follows from the observation that the next transition of $\Theta(w_i(0))$ after w_i in σ is the first transition in some w_j and hence is visible. By definition of $\mathcal{T}'$, for each w_i, $\mathcal{T}'$ contains a transition $\sigma'(i)$ that is the sequential composition of the transitions in w_i. Thus, $s'_0 \xrightarrow{\sigma'(0)} s'_1 \xrightarrow{\sigma'(1)} s'_2 \cdots \xrightarrow{\sigma'(m-1)} s'_m \xrightarrow{\sigma'(m)} s'_{m+1}$ with $s'_0 = s_{init}$ and $s'_{m+1} = s$, so s is a reachable visible state of $\mathcal{C}(\mathcal{S})$. ∎

Proof of Theorem 2′: Some observations about accesses: (A1) in all states in which a transition t is enabled, t accesses the same set of objects, namely, those used in its guard or command; (A2) in all states in which a transition t is pending and disabled, t accesses the same set of objects, namely those used in its guard. Some observations about MLD′: (M1) a transition t that is pending in a state s can violate MLD′ in s even if t is disabled in s; (M2) after initialization, whether an object satisfies MLD′-R depends on the set of accesses (specifically, whether it contains a write) but not on their order; (M3) after initialization, whether an object satisfies MLD′-L depends on the set of accesses that occur (and the associated sets of held locks) but not on their order, because set intersection is commutative and associative. Let s and σ be as in the proof of Therorem 2. Consider cases corresponding to the places in which a violation of MLD′ could affect the proof of Theorem 2.

case 1: for all $j \in [0..|\sigma|-1]$, $s \stackrel{\sigma(0..j)}{\Longrightarrow}$ and $s \stackrel{\sigma(0..j-1);t}{\Longrightarrow}$ satisfy MLD′. In this case, the proof of Theorem 2 goes through, and $enabled(s, \theta)$ is persistent in s.

case 2: there exists $j \in [0..|\sigma|-1]$ such that $s \stackrel{\sigma(0..j)}{\Longrightarrow}$ or $s \stackrel{\sigma(0..j-1);t}{\Longrightarrow}$ violates MLD′. Let j denote the least such j. The proof of Theorem 2 goes through for $\sigma(0..j-1)$; specifically, for $i < j$, $thread(t) \neq thread(\sigma(i))$ and t is independent with $\sigma(i)$ in s_i. Independence of t with transitions in $\sigma(0..j-1)$ and the definitions of t and σ together imply that $\langle t \rangle \cdot \sigma(0..j-1)$ can be executed from s, and t can be executed from s_j.

case 2.1: $s \stackrel{\sigma(0..j-1)\cdot\langle t \rangle}{\Longrightarrow}$ violates MLD′. The violation occurs when t accesses some object o.

case 2.1.1: $\sigma(0..j-1)$ contains an initialization transition $\sigma(i)$ for o. MLD′ requires that $thread(\sigma(i))$ be the first thread to access o, so $s_0 \stackrel{\langle\rangle;t}{\Longrightarrow}$ violates MLD′, because $thread(\sigma(i)) \neq thread(t)$, and because t is enabled in s_0 and hence accesses in s_0 a superset (not necessarily proper) of the objects that it accesses in s_j. Thus, $enabled(s, \theta)$ contains a transition t such that $s \stackrel{\langle\rangle;t}{\Longrightarrow}$ violates MLD′. (For $j > 0$, we could conclude that this case is impossible, since it contradicts the definition of j.)

case 2.1.2: $\sigma(0..j-1)$ does not contain an initialization transition for o. Observations A1-A2 and M1-M3 and invisibility of t imply that $s \stackrel{\langle t \rangle \cdot \sigma(0..j-1)}{\Longrightarrow}$ also violates MLD′, since the same set of accesses with the same sets of held locks occur in $\sigma(0..j-1) \cdot \langle t \rangle$ and $\langle t \rangle \cdot \sigma(0..j-1)$. The violation might occur at any point in $\langle t \rangle \cdot \sigma(0..j-1)$; regardless of when it occurs, the requirements of the theorem are satisfied.

case 2.2: $s \stackrel{\sigma(0..j-1)\cdot\langle t \rangle}{\Longrightarrow}$ satisfies MLD′. In this case, $s \stackrel{\sigma(0..j)}{\Longrightarrow}$ violates MLD′.

case 2.2.1: after $s \stackrel{\sigma(0..j-1)}{\Longrightarrow}$, some access by $\sigma(j)$'s guard violates MLD′. Observations A1-A2 and M1-M3 and invisibility of t imply that $s \stackrel{\langle t \rangle \cdot \sigma(0..j-1);\sigma(j)}{\Longrightarrow}$ also violates MLD′ (even though $\sigma(j)$ might be disabled after $s \stackrel{\langle t \rangle \cdot \sigma(0..j-1)}{\Longrightarrow}$).

case 2.2.2: after $s \stackrel{\sigma(0..j-1)}{\Longrightarrow}$, all accesses by $\sigma(j)$'s guard satisfy MLD′. In this case, some access by $\sigma(j)$'s command violates MLD′. As in the proof of Theorem 2, t does not affect $\sigma(j)$'s enabledness, so $\sigma(j)$ is enabled after $s \stackrel{\langle t \rangle \cdot \sigma(0..j-1)}{\Longrightarrow}$. Observations A1-A2 and M1-M3 and invisibility of t imply that $s \stackrel{\langle t \rangle \cdot \sigma(0..j)}{\Longrightarrow}$ also violates MLD′. ∎

Proof of Theorem 3′: We suppose $\mathcal{S}$ violates MLD′, and IF-SSS does not find a violation, and reach a contradiction. The basic idea is to prove an invariant that a violation of MLD′ is reachable from some state in the "to do" stack of the depth-first search. DFS is written in recursive, not iterative, form, so it does not have an explicit "to do" stack. We introduce one as an auxiliary variable, by inserting the following statements: in IF-SSS, immediately after the assignment to *curState*, insert "*toDo* := singleton stack containing *curState*"; in DFS_{if}, immediately after the statement containing *execInvis*, insert "push *curState* onto *toDo*", and immediately after the end of the **while** loop, insert "pop from *toDo*".[6] We show below that the following predicate I holds starting after initialization of *toDo*: ($\exists s \in toDo$: a violation of MLD′ is reachable from s). At the end of the search, *toDo* is empty, so I does not hold, a contradiction.

We assume the system is augmented with a variant of the lockset algorithm, so that violations of MLD′ correspond to reachability of a control point. Theorem 1 implies that if persistent sets and sleep sets are

[6] *toDo* cannot be defined as a function of *stack* and *curState* alone, because those variables have the same values at the beginning and end of the search; in contrast, *toDo* has initial value $\{s_{init}\}$ and final value $\emptyset$. This is why we add statements to IF-SSS.

computed correctly, then reachability of control points and hence violations of MLD′ are determined correctly, and I is preserved. Consider an invisible transition t explored by *execInvis*. A violation of MLD′ can cause two problems: (1) $\{t\}$ might not be persistent; (2) t might be dependent with a transition t' in *sleep*, in which case t' incorrectly remains in *sleep*. We show that these problems do not falsify I.

(1) Suppose $\{t\}$ is not persistent. Theorem 2′ implies that either (a) $s \stackrel{\langle\rangle;t}{\Longrightarrow}$ violates MLD′, or (b) $s \stackrel{t}{\rightarrow} s'$ and a violation of MLD′ is reachable from s'. Case (a) contradicts the hypothesis that IF-SSS does not find a violation of MLD′. Case (b) implies that I still holds, because s' or an appropriate state reachable from s' will be added to *toDo* (more precisely, a simple induction on the length of the explored sequence of invisible transitions is needed).

(2) Suppose t is dependent with t'. A property of sleep sets is $sleep \subseteq enabled(s)$, so $t' \in enabled(s)$. t is invisible, so Separation and DetermInvis imply $enabled(s,\theta) = \{t\}$, so dependency of t and t' implies that $enabled(s,\theta)$ is not persistent, so Theorem 2′ implies that either (a) $s \stackrel{\langle\rangle;t}{\Longrightarrow}$ violates MLD′, or (b) $s \stackrel{t}{\rightarrow} s'$ and a violation of MLD′ is reachable from s'. The rest of the proof is the same as in the previous paragraph. ∎