# Model-Checking Multi-Threaded Distributed Java Programs*

Scott D. Stoller

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

**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

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

require storage of previously-visited states. State-less 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 [Bru99] 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 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.

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 use locks to avoid race conditions in accesses to shared variables. Thus, it handles a strict superset of the systems handled by VeriSoft or ExitBlock. Related work is discussed further 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 check whether a system satisfies MLD. If it does, 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

- $\Theta$ 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.
- $\mathcal{O}_{unsh} \subseteq \mathcal{O}$ 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 $\theta$ in state $s$.

A transition $\langle S, G, C, F \rangle$ of thread $\theta$ 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 $\mathcal{A}$, let $pending_{\mathcal{A}}(s, \theta)$ and $enabled_{\mathcal{A}}(s, \theta)$ denote the sets of transitions of $\theta$ that are pending and enabled, respectively, in state $s$ (in system $\mathcal{A}$). Let $enabled_{\mathcal{A}}(s)$ 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 $V$. $s \xrightarrow{t} s'$ means 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 $\sigma$. Let $\sigma(i..j)$ denote the subsequence of $\sigma$ 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 $\sigma_1$ and $\sigma_2$.

An *execution* of a concurrent system $\mathcal{A}$ is a finite or infinite sequence $\sigma$ of transitions of $\mathcal{A}$ 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.

### 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.

Thread $\theta$ acquiring $o$'s lock corresponds to a transition with guard $o.owner \in \{free, \theta\}$ and command $o.owner := \theta; o.depth{+}{+}$. Thread $\theta$ 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 $o.owner := (o.depth = 1)\,?\,free : \theta; o.depth{-}{-}$.[1]

Let *tmpDepth* denote an unshared natural-number-valued object used by $\theta$. Thread $\theta$ 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 $\theta$ doing notify 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 $o.waitRemove()$, which removes an arbitrary element of the set. notifyAll is similar, except that *Remove* is replaced with *RemoveAll*.

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:

[1] The definition of command does not allow conditionals. 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.

**SyncWithoutLock:** If a thread $\theta$ executes an operation *op* other than acquire on a synchronization object $o$ in a state $s$ in which $\theta$ 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). The guard or command of a transition *accesses* object $o$ if it contains an operation on $o$. Transition $t$ *accesses* object $o$ in state $s$ if (1) $t$ is pending in $s$ and $t$'s guard accesses $o$ or (2) $t$ is enabled in $s$ and $t$'s command accesses $o$. Thread $\theta$ *accesses* object $o$ in state $s$, denoted $access(s, \theta, o)$, if there exists a transition in $pending(s, \theta)$ that accesses $o$ in $s$. $startShared(\sigma, o)$ is the index of the first state in $\sigma$ in which an access to $o$ that is not part of initialization of $o$ occurs; formally, letting $\sigma$ 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 $\theta$ owns $o_1$'s lock in $s_i$.

Godefroid [God96] defines: transition $t$ uses object $o$ iff $t$'s guard or command contains an operation on $o$. A use of $o$ by the command of a disabled transition 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 $\theta$, 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 $\theta$, $s_{init}(\theta)$ is visible.

**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 $\theta$, every contiguous sequence of $b+1$ transitions executed by $\theta$ (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 $\theta$, $\theta$ has at most one enabled invisible transition.

**Non-Blocking Invisible Control Points (NonBlockInvis):** For every thread $\theta$, for every invisible control point $S$ of $\theta$, for every reachable state $s$ containing $S$, $enabled(s, \theta) \neq \emptyset$.

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 $\theta$ 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

```
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};
}
```

**Fig. 1.** State-less Selective Search (SSS) using persistent sets and sleep sets.

$s$ 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 $\mathcal{A}$. $D \subseteq \mathcal{T} \times \mathcal{T}$ is an *unconditional dependency relation* for $\mathcal{A}$ 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), and (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 $\mathcal{A}$ 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 $\sigma$ 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. 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.

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

**Proof**: This is a paraphrase of Theorem 2 of [God97]. □

## 4 Invisible-First Selective Search

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

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

**Proof**: Let $\sigma$ be a sequence of transitions as in the definition of persistent set. Let $t \in enabled(s, \theta)$. Separation implies that $t$ is invisible. It suffices to show that $\sigma(n)$ is independent in $s_n$ with $t$. First, we show that $\sigma$ does not contain transitions of $thread(t)$; second, we show the desired independence. Roughly, MLD implies that accesses to objects in $\mathcal{O}_{mtx}$ do not cause dependence; invisibility of $t$ implies that accesses to communication objects do not cause dependence; and SyncWithoutLock implies that accesses to synchronization objects do not cause dependence. For details, see [Sto00]. □

Suppose the system satisfies MLD, Separation, BoundedInvis, and DetermInvis. If a thread $\theta$ has an enabled invisible transition in a state $s$, then Separation and DetermInvis imply that $\theta$ 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 $\theta$ has an enabled visible transition. BoundedInvis implies that this iteration terminates. Let $execInvis_{\mathcal{A}}(s, \theta)$ be the unique state obtained by performing this procedure starting from state $s$; if $\theta$ has no enabled invisible transitions in state $s$, then we define $execInvis_{\mathcal{A}}(s, \theta) = s$. Specializing SSS to work in this way yields Invisible-First State-less Selective Search (IF-SSS), given in Figure 2.

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

**Proof**: This follows from Theorems 1 and 2, and the fact that transitions in *sleep* are independent with invisible transitions executed by *execInvis* (this follows

```
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_A(curState, thread(t));
    sleep' := {t' ∈ sleep | ⟨t, t'⟩ ∉ D};
    DFS_if(sleep')
    pop t from stack;
    curState  := undo(curState, t);
    sleep  := sleep ∪ {t};
}
```

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

from Separation, DetermInvis, and Theorem 2), so it is safe not to explicitly check that independence when computing $sleep'$. For details, see [Sto00]. □

## 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 appears in Section 6. Transitions are amalgamated (composed) as follows. Given a sequence $\sigma$ of transitions, let $cmd_{seq}(\sigma)$ be the sequential composition of the commands of the transitions in $\sigma$, and let $guard_{seq}(\sigma)$ be the weakest predicate ensuring that when each transition $t$ in $\sigma$ is executed, $t$'s guard holds. Let $final(t)$ denote the final control point of transition $t$.

Given a concurrent system $\mathcal{A} = \langle \Theta, \mathcal{O}, \mathcal{T}, s_{init}, \mathcal{O}_{unsh}, \mathcal{O}_{syn}, \mathcal{O}_{mtx} \rangle$ satisfying MLD, Separation, BoundedInvis, and DetermInvis, $\mathcal{C}(\mathcal{A})$ 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 $\mathcal{A}$. For each visible transition $t = \langle S, G, C, F \rangle$ in $\mathcal{T}$, for each sequence $\sigma$ 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 $\mathcal{A}$ be a concurrent system satisfying MLD, Separation, InitVis, BoundedInvis, and DetermInvis. $s$ is a reachable visible state of $\mathcal{A}$ iff $s$ is a reachable visible state of $\mathcal{C}(\mathcal{A})$.*

**Proof**: A proof sketch follows; for details, see [Sto00].

($\Leftarrow$): Let $s$ be a reachable visible state of $\mathcal{C}(\mathcal{A})$. Let $\sigma$ be an execution of $\mathcal{C}(\mathcal{A})$ containing $s$. Expanding each transition in $\sigma$ into the sequence of transitions of $\mathcal{A}$ from which it is composed yields an execution of $\mathcal{A}$ that contains $s$.

($\Rightarrow$): Let $s$ be a reachable visible state of $\mathcal{A}$. Let $\sigma$ be an execution of $\mathcal{A}$ containing $s$. We re-arrange $\sigma$ as follows: for each thread $\theta$, move the invisible transitions of $\theta$ that appear between the $i$'th and $(i+1)$'th visible transitions of $\theta$ backwards so that those invisible transitions form a contiguous subsequence of the re-arranged execution starting immediately after the $i$'th visible transition of $\theta$. We use MLD and Theorem 2 to show that this can be achieved by interchanging independent transitions. From the re-arranged execution of $\mathcal{A}$, we can easily form an execution of $\mathcal{C}(\mathcal{A})$ containing $s$. □

**Theorem 5.** *Let $\mathcal{A}$ 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 $\mathcal{A}$ iff SSS applied to $\mathcal{C}(\mathcal{A})$ explores $d$. A control point $S$ is reachable in $\mathcal{A}$ iff SSS applied to $\mathcal{C}(\mathcal{A})$ explores a state containing $S$.*

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

## 6 Comparison of Invisible-First and Composition

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 $\mathcal{C}(\mathcal{A})$. Third, the guards of composed transitions sometimes introduce dependencies that cause SSS applied to $\mathcal{C}(\mathcal{A})$ to explore more interleavings than IF-SSS. For example, consider a thread $\theta$ that is ready to execute **if** $(x_1)$ { **if** $(x_2)$ $c_1$ **else** $c_2$ } **else** { **if** $(x_3)$ $c_3$ **else** $c_4$ }, 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 $\mathcal{A}$, $\theta$ accesses only $x_1$ at $S$. In the composed system, $\theta$ 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 $x_1 \wedge \neg x_3$. In a state $s$ with $s(\theta) = S$ and $s(x_1) = \text{false}$, the access by $\theta$ to $x_3$ in the composed system is an artifact of composition. Such accesses introduce dependencies that could cause persistent sets to be larger in $\mathcal{C}(\mathcal{A})$ than $\mathcal{A}$, if the calculation of persistent sets—specifically, the calculation of *pendInvisOps*, defined in Section 8—exploits information from static analysis.

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 $\mathcal{C}(\mathcal{A})$, each thread can do this with a single transition, and those two transitions are independent, so SSS applied to $\mathcal{C}(\mathcal{A})$ could explore a single interleaving. In $\mathcal{A}$, each thread does

this with a sequence of three transitions, and the transitions that manipulate the lock are not independent, so IF-SSS applied to $\mathcal{A}$ explores multiple interleavings. A more detailed example appears in [Sto00].

## 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 $\langle t, t', \mathit{curState} \rangle \notin D$. 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. 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. 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.

## 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 transi-

tions 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 $\theta$. 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 $\theta$, the following set is known:

$$pendVisOps(s,\theta) = \bigcup_{t \in pending(s,\theta)} usedVisOps(t) \tag{1}$$

To compute small persistent sets, it is important to have information about the set of invisible operations used by pending transitions of $\theta$ 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.

The lockset algorithm uses the following data structures. For each object $o$, the following values are maintained: $o.mode$, which is 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 $\theta$, $held(s,\theta)$, the set of synchronization objects whose locks are held by $\theta$ 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 $o$.

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

$$\begin{aligned} MLDallows(s,\theta,o_1,o.op) = \\ &\lor o.mode = \text{virgin} \land mayInit(s,\theta,o) \\ &\lor o.mode = \text{exclusive} \land (\theta = o.firstThread \lor rdOnly(op) \lor o_1 \in o.candLockSet) \\ &\lor o.mode = \text{shared} \land (rdOnly(op) \lor o_1 \in o.candLockSet) \\ &\lor o.mode = \text{shared-modified} \land o_1 \in o.candLockSet \end{aligned}$$

where $rdOnly(op)$ holds if $op$ is read-only, and $mayInit(s,\theta,o)$ holds if $\theta$ 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 $\theta$ be the thread that allocated $o$ (for static variables of a class $C$, $\theta$ 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 $\mathcal{A}$" 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 $\theta'$ in $T$.
3. Repeat step 2 until no more processes can be added. Return $\cup_{\theta \in T} enabled(s,\theta)$.

**Theorem 6.** *Let $\mathcal{A}$ be a concurrent system satisfying MLD. In every state $s$ of $\mathcal{A}$, 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 $\theta_1$ can perform the sequence of three transitions (control points are omitted in this informal shorthand) $\langle v := 0, sem.up(), v := 1\rangle$, and $\theta_2$ 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 $\theta_1$ 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 $\sigma$ 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 $o$.[2] 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 $\theta$ owns $o_1$'s lock in $s_i$.

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

**Theorem 2$'$.** *Let $\mathcal{A}$ be a concurrent system satisfying Separation. For all threads $\theta$ 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 $s'$.*

**Proof**: MLD$'$ is relatively insensitive to the order in which accesses occur. Let $\sigma$ be a sequence of transitions as in the definition of persistent set. One can show (roughly) that if a violation of MLD$'$ occurs when $\sigma$ is executed before $t$, then a violation also occurs if $t$ is executed before $\sigma$. For details, see [Sto00]. □

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

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

**Proof**: An invariant $I$ is used to show that violations of MLD$'$ are eventually detected, even if they cause persistent sets or sleep sets to be computed incorrectly. The proof of invariance of $I$ is based on Theorem 2$'$. For details, see [Sto00]. □

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

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

($\Rightarrow$): Theorem 3$'$ implies that IF-SSS explores an execution $\sigma$ of $\mathcal{A}$ that violates MLD$'$. Composing sequences of transitions in $\mathcal{A}$ to form transitions of $\mathcal{C}(\mathcal{A})$ yields an execution of $\mathcal{C}(\mathcal{A})$ 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 $\mathcal{A}$ be a concurrent system. In every state $s$ of $\mathcal{A}$, 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*$'(s, \theta)$ is invoked only for threads $\theta$ that have already been added to $T$. Suppose for all threads $\theta$ in $T$, all transitions in $enabled(s, \theta)$ 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 $\theta$ in $T$ such that some transition $t$ in $enabled(s, \theta)$ violates MLD$'$ in $s$. Then $P$ contains $t$, and $t$ violates MLD$'$ in $s$. □

Let $\mathcal{A}$ 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 $\mathcal{C}(\mathcal{A})$ augmented with the lockset algorithm, modified slightly to check MLD$'$. Theorem 6$'$ implies that if no violation of MLD$'$ is found, then $\mathcal{C}(\mathcal{A})$ satisfies MLD$'$ and hence MLD. Theorem 3 Theorem 5$'$ then implies that $\mathcal{A}$ satisfies MLD. Theorem 5 can then be used to conclude that reachability of control points and deadlocks was correctly determined during the search.

Let $\mathcal{A}$ 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 $\mathcal{A}$ augmented with the lockset algorithm, modified slightly to check MLD$'$. Theorems 3$'$ and 6$'$ imply that if no violation of MLD$'$ is found, then $\mathcal{A}$ 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 $\mathcal{C}(\mathcal{A})$ augmented to check MLD$'$. Theorem 6$'$ implies that if no violation of MLD$'$ is found, then $\mathcal{C}(\mathcal{A})$ satisfies MLD$'$, so Theorem 5$'$ implies that $\mathcal{A}$ satisfies MLD$'$ and hence MLD. Theorem 5 implies that reachability (in $\mathcal{A}$) 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 state-less 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 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. 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*.

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. 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) a heart attack. We insert in java.lang.Object a boolean-valued method with body "return false". It is overridden in all wrapper classes by a method with body "return true".

Certain calls to java.util.Random are treated as non-deterministic, *i.e.*, all possible return values are explored; this is like VS_Toss in VeriSoft [God97].

$undo(s,t)$ can be implemented by reverse computation, reset+replay, or checkpointing. Reverse computation is attractive in theory but difficult to implement. Our prototype uses reset+replay (like VeriSoft), mainly because it is easy to implement. ExitBlock [Bru99] and Java PathFinder [BHPV00] use checkpointing, which requires a custom 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. Such tools are typically applied to small problem instances that consume relatively little CPU time.

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.

## 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 selective search. It ensures that each state is explored at most once but probably has more overhead than our bytecode rewriting. It uses partial-order reductions 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 [Bru99], 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 $\mathcal{T} \times \mathcal{T}$. 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 $\mathcal{C}(\mathcal{A})$) 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.

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 [Bru99, 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.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 [Bru99, Theorem 3, pp. 47-48] does not imply that ExitBlock finds all violations of MLD, because that proof presupposes that the system satisfies MLD.

In Bruening's proof that ExitBlockRW finds all assertion violations [Bru99, pp. 53-54], the requirements on $s'_i$ and $s'_{i-1}$ are symmetric, so two swapped segments can be swapped again, so the meaning of "Move each segment in this way as far as possible to the left" is unclear. 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.

## 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.

[Bru99] 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.

[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.

[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.

[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.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. Technical Report 536, Computer Science Dept., Indiana University, 2000.

[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, October 1999.